Combinatorial Pattern Matching

27th Annual Symposium on Combinatorial Pattern Matching,
June 27-29, 2016, Tel Aviv, Israel

Edited by

Roberto Grossi

Moshe Lewenstein

Editors

Roberto Grossi
Dipartimento di Informatica
Università di Pisa
grossi@di.unipi.it

Moshe Lewenstein
Department of Computer Science
Bar Ilan University
moshe@cs.biu.ac.il

ACM Classification 1998
E.1 Data Structures, E.2 Data Storage Representations, E.4 Coding and Information Theory, F. Theory of Computation G.2 Discrete Mathematics, H. Information Systems, I.7 Document and Text Processing

ISBN 978-3-95977-012-5

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-012-5.

Publication date
June 2016

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

Digital Object Identifier: 10.4230/LIPIcs.CPM.2016.0

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

http://www.dagstuhl.de/lipics

Contents

27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016).
Editors: Roberto Grossi and Moshe Lewenstein
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The objective of the Annual Symposium on Combinatorial Pattern Matching is to provide an international forum for research in combinatorial pattern matching and related applications. It addresses issues of searching and matching strings and more complicated patterns such as trees, regular expressions, graphs, point sets, and arrays. The goal is to derive combinatorial properties of such structures and to exploit these properties in order to achieve a superior performance for the corresponding computational problems. The meeting also deals with problems in bioinformatics and computational biology, coding and data compression, combinatorics on words, data mining, information retrieval, natural language processing, pattern discovery, string algorithms, string processing in databases, symbolic computing, and text searching.

This volume contains the papers presented at the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016) held during June 27-29, 2016, in Tel Aviv, Israel

The conference program included 29 contributed papers and three invited talks by Gregory Kucherov, University Paris-Est Marne-la-Vallée, France, on "Stringology in action: bioinformatics examples", Moni Naor, Weizmann Institute, Israel, on "How to share a secret, infinitely", and Yoram Louzon, Bar-Ilan University, Israel, on "Node classification based on local and global sub-graph patterns".

The contributed papers were selected out of 52 submissions, corresponding to an acceptance ratio of 55.8%. Each submission received at least three reviews. We thank the members of the Program Committee and all the additional external reviewers for their hard and invaluable work that resulted in an excellent scientific program.

The Annual Symposium on Combinatorial Pattern Matching started in 1990, and has since taken place every year. Previous CPM meetings were held in Paris, London (UK), Tucson, Padova, Asilomar, Helsinki, Laguna Beach, Aarhus, Piscataway, Warwick, Montreal, Jerusalem, Fukuoka, Morelia, Istanbul, Jeju Island, Barcelona, London (Ontario, Canada), Pisa, Lille, New York, Palermo, Helsinki, Bad Herrenalb, Moscow, and Ischia. From the 3rd to the 26th meeting, proceedings of all meetings have been published in the LNCS series, as volumes 644, 684, 807, 937, 1075, 1264, 1448, 1645, 1848, 2089, 2373, 2676, 3109, 3537, 4009, 4580, 5029, 5577, 6129, 6661, 7354, 7922, 8486, and 9133, respectively. From the curent meeting, the proceedings will appear in the LIPIcs (Leibniz International Proceedings in Informatics) series.

Selected papers from the 1st meeting appeared in vol. 92 of Theoretical Computer Science, from the 11th meeting in vol. 2 of the Journal of Discrete Algorithms, from the 12th meeting in vol. 146 of Discrete Applied Mathematics, from the 14th meeting in vol. 3 of the Journal of Discrete Algorithms, from the 15th meeting in vol. 368 of Theoretical Computer Science, from the 16th meeting in vol. 5 of the Journal of Discrete Algorithms, from the 19th meeting in vol. 410 of Theoretical Computer Science, from the 20th meeting in vol. 9 of the Journal of Discrete Algorithms, from the 21st meeting in vol. 213 of Information and Computation, from the 22nd meeting in vol. 483 of Theoretical Computer Science, and from the 23rd meeting in vol. 25 of the Journal of Discrete Algorithms. Selected papers from this meeting will appear in a special issue of Algorithmica.

The whole submission and review process was carried out with the help of the EasyChair conference system. We thank the CPM Steering Committee for supporting Tel Aviv as the site for CPM 2016, and for their advice and help in different issues. We thank Shay Golan, Avivit Levy, Noa Lewenstein and Ely Porat for the local arrangements. The conference was

sponsored by the I-CORE Program of the planning and budgeting committee and The Israel Science Foundation (grant number 4/11), Shenkar College of Engineering and Design, and Bar Ilan University. We thank them for their financial support.

List of Authors

27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016).
Editors: Roberto Grossi and Moshe Lewenstein
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Deterministic Sub-Linear Space LCE Data Structures With Efficient Construction*

Yuka Tanimura[1], Tomohiro I[2], Hideo Bannai[3], Shunsuke Inenaga[4], Simon J. Puglisi[5], and Masayuki Takeda[6]

1 **Department of Informatics, Kyushu University, Japan**
`yuka.tanimura@inf.kyushu-u.ac.jp`
2 **Kyushu Institute of Technology, Japan**
`tomohiro@ai.kyutech.ac.jp`
3 **Department of Informatics, Kyushu University, Japan**
`bannai@inf.kyushu-u.ac.jp`
4 **Department of Informatics, Kyushu University, Japan**
`inenaga@inf.kyushu-u.ac.jp`
5 **Department of Computer Science, University of Helsinki, Finland**
`puglisi@cs.helsinki.fi`
6 **Department of Informatics, Kyushu University, Japan**
`takeda@inf.kyushu-u.ac.jp`

Abstract

Given a string S of n symbols, a longest common extension query $\mathsf{LCE}(i, j)$ asks for the length of the longest common prefix of the ith and jth suffixes of S. LCE queries have several important applications in string processing, perhaps most notably to suffix sorting. Recently, Bille et al. (J. Discrete Algorithms 25:42–50, 2014, Proc. CPM 2015:65–76) described several data structures for answering LCE queries that offers a trade-off between data structure size and query time. In particular, for a parameter $1 \leq \tau \leq n$, their best deterministic solution is a data structure of size $O(\frac{n}{\tau})$ which allows LCE queries to be answered in $O(\tau)$ time. However, the construction time for all deterministic versions of their data structure is quadratic in n. In this paper, we propose a deterministic solution that achieves a similar space-time trade-off of $O(\tau \min\{\log \tau, \log \frac{n}{\tau}\})$ query time using $O(\frac{n}{\tau})$ space, but we significantly improve the construction time to $O(n\tau)$.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases longest common extension, longest common prefix, sparse suffix array

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.1

1 Introduction

Given a string S of n symbols, a *longest common extension query* $\mathsf{LCE}(i, j)$ asks for the length of the longest common prefix of the ith and jth suffixes of S.

The ability to efficiently answer LCE queries allows optimal solutions to many string processing problems. Gusfield's book [4], for example, lists several applications of LCEs to basic pattern matching and discovery problems, including: pattern matching with wildcards, mismatches and errors; the detection of various types of palindromes (maximal, complimented, separated, approximate); and the detection of repetitions and approximate repetitions.

* HB, SI, MT were supported by JSPS KAKENHI Grant Numbers 25280086, 26280003, 25240003.

Table 1 Deterministic solutions to LCE.

Data Structure		Preprocessing		Trade-off range	Reference
Space	Query	Space	Time		
1	n	1	1	-	naïve computation
n	1	n	n	-	suffix array + RMQ
$\frac{n}{\tau}$	τ^2	$\frac{n}{\tau}$	$\frac{n^2}{\tau}$	$1 \leq \tau \leq \sqrt{n}$	[3]
$\frac{n}{\tau}$	$\tau \log^2 \frac{n}{\tau}$	$\frac{n}{\tau}$	n^2	$1 \leq \tau \leq n$	[2], Section 2
$\frac{n}{\tau}$	τ	$\frac{n}{\tau}$	$n^{2+\epsilon}$	$1 \leq \tau \leq n$	[2], Section 4
$\frac{n}{\tau}$	$\tau \log^2 \frac{n}{\tau}$	$\frac{n}{\tau}$	$n\tau + n \log \frac{n}{\tau}$	$1 \leq \tau \leq n$	This work, Theorem 9
$\frac{n}{\tau}$	$\tau \log \tau$	$\frac{n}{\tau}$	$n\tau$	$1 \leq \tau \leq \frac{n}{\log n}$	This work, Theorem 10
$\frac{n}{\tau}$	$\tau \min\{\log \tau, \log \frac{n}{\tau}\}$	$\frac{n}{\tau}$	$n\tau$	$1 \leq \tau \leq n$	This work, Corollary 12

Lempel-Ziv parsing [6] and suffix sorting [7, 5] are two more fundamental string processing problems to which LCEs are key.

Without preprocessing, answering an arbitrary query $\mathsf{LCE}(i, j)$ requires $O(n)$ time: we simply compare the suffixes starting at positions i and j character by character until we find a mismatch. To answer queries faster we could build the suffix tree and preprocess it for lowest-common-ancestor queries. This well-known solution answers queries in $O(1)$ time and the data structure is of $O(n)$ size and takes $O(n)$ time to construct.

In recent years, motivated by scenarios where $O(n)$ space is prohibitive, several authors have sought data structures that achieve a trade-off between data structure size and query time [13, 3, 2]. The best trade-off to date is due to Bille et al. [2], where they describe a data structure of size $O(n/\tau)$ which allow LCE queries to be answered in $O(\tau)$ time.

However, as described in [2], their deterministic data structure requires $O(n^2)$ time to construct if only $O(n/\tau)$ working space is allowed. This is a major drawback, because it does not allow the space-query time trade-off to be passed on to applications—indeed, construction of the data structure would become a time bottleneck in all the applications listed above. We note that Bille et al. [2] also proposed randomized solutions which achieve the same space-query time trade-off with subquadratic preprocessing time. In this paper, we focus on determinstic solutions.

The main contributions of this article are as follows:

1. We describe a new data structure for LCEs that has size $O(\frac{n}{\tau})$, query time $O(\tau \log \tau)$, and, critically, can be constructed in $O(n\tau)$ time.
2. We show how to combine the new data structure with one of Bille et al. to derive a structure that has $O(\tau \min\{\log \tau, \log \frac{n}{\tau}\})$ query time and the same space and construction bounds as the new structure. As a side result, we also show how this particular structure of Bille et al. can be constructed efficiently.

Table 1 summarizes our results and previous work on the deterministic version of the problem.

In the next section we lay down notation and some basic algorithmic and data structural tools. Then, in Section 3, we introduce our new LCE data structures, beginning with a a slightly modified version of one of Bille et al.'s data structures, followed by the new and combined data structures. Section 4 deals with efficient construction. We finish, in Section 5, by noting that our new structures lead directly to improved (deterministic) bounds for the sparse suffix sorting problem.

2 Preliminaries

Let $\Sigma = \{1, \ldots, \sigma\}$ denote the alphabet, and Σ^* the set of strings. If $w = xyz$ for some strings w, x, y, z, then x, y, and z are respectively called a *prefix*, *substring*, and *suffix* of w. For any string w, let $|w|$ denote the length of w, and for any $0 \leq i < |w|$, let $w[i]$ denote the ith character of w, i.e., $w = w[0] \cdots w[|w| - 1]$. For convenience, let $w[i] = 0$ when $i \geq |w|$. For any $0 \leq i \leq j$, let $w[i..j] = w[i] \cdots w[j]$, and for any $0 \leq i < |w|$, let $w[i..] = w[i..|w| - 1]$. We denote $x \prec y$ if a string x is lexicographically smaller than a string y.

For any string w, let $\mathsf{lcp}_w(i, j)$ denote the length of the longest common prefix of $w[i..]$ and $w[j..]$. We will write $\mathsf{lcp}(i, j)$ when w is clear from the context. Since $\mathsf{lcp}_w(i, i) = |w| - i$, we will only consider the case when $i \neq j$. Note that answering an LCE query $\mathsf{LCE}(i, j)$ is equivalent to computing $\mathsf{lcp}_w(i, j)$.

For any integers $i \leq j$, let $[i..j]$ denote the set of integers from i to j (including i and j), and for $0 \leq p < \tau$, let $[i..j]^\tau_p = \{k \mid k \in [i..j], k \bmod \tau = p\}$.

For any string w of length n and $0 \leq p < \tau$, let $\hat{w}_{\tau,p}$ denote a string of length $\lceil (|w| - p)/\tau \rceil$ over the alphabet $\{1, \ldots, \sigma^\tau\}$ such that $\hat{w}_{\tau,p}[i] = w[p + \tau i..p + \tau(i + 1) - 1]$ for any $i \geq 0$. We call $\hat{w}_{\tau,p}$ the *meta-string* of w wrt. sampling rate τ and offset p, and each character of $\hat{w}_{\tau,p}$ is called a *meta-character*.

In the rest of the paper, we assume a polynomially bounded integer alphabet, i.e., for some constant $c \geq 0$, $\sigma = O(n^c)$ for any input string w of length n.

▶ **Definition 1** ([12]). The suffix array SA_w of a string w of length n is an array of size n containing a permutation of $[0..n - 1]$ that represents the lexicographic order of the suffixes of w, i.e., $w[\mathsf{SA}_w[0]..] \prec \cdots \prec w[\mathsf{SA}_w[n - 1]..]$. The inverse suffix array ISA_w is an array of size n such that $\mathsf{ISA}_w[\mathsf{SA}_w[i]] = i$ for all $0 \leq i < n$. The LCP array LCP_w of a string w of length n is an array of size n such that $\mathsf{LCP}_w[0] = 0$ and $\mathsf{LCP}_w[i] = \mathsf{lcp}_w(\mathsf{SA}_w[i - 1], \mathsf{SA}_w[i])$ for $0 < i < n$.

▶ **Lemma 2** ([9, 10, 11, 7]). *For any string w of length n, the arrays $\mathsf{SA}_w, \mathsf{ISA}_w, \mathsf{LCP}_w$ can be computed in $O(n)$ time and space.*

For any array A and $0 \leq i \leq j < |A|$, let $\mathsf{rmq}_A(i, j)$ denote a Range Minimum Query (RMQ), i.e., $\mathsf{rmq}_A(i, j) = \arg\min_{k \in [i..j]}\{A[k]\}$. It is well known that A can be preprocessed in linear time and space so that $\mathsf{rmq}_A(i, j)$, for any $0 \leq i \leq j < |A|$, can be answered in constant time [1]. Since $\mathsf{lcp}_w(i, j) = \mathsf{LCP}_w[\mathsf{rmq}_{\mathsf{LCP}_w}(i' + 1, j')]$ where $i' = \min\{\mathsf{ISA}_w(i), \mathsf{ISA}_w(j)\}$ and $j' = \max\{\mathsf{ISA}_w(i), \mathsf{ISA}_w(j)\}$, it follows that a string of length n can be preprocessed in $O(n)$ time and space so that for any $0 \leq i, j < n$, $\mathsf{lcp}_w(i, j)$ can be computed in $O(1)$ time.

Our algorithm relies on sparse suffix arrays. For a string w of length n and any set $P \subseteq [0..n-1]$ of positions, let $\mathsf{SSA}_P[0..|P|-1]$ be an array consisting of entries of SA that are in P, i.e., for any $0 \leq i < |P|$, $\mathsf{SSA}_P[i] \in P$, and $w[\mathsf{SSA}_P[0]..] \prec \cdots \prec w[\mathsf{SSA}_P[|P|-1]..]$. The sparse LCP array $\mathsf{SLCP}_P[0..|P| - 1]$ is defined analogously, $\mathsf{SLCP}_P[i] = \mathsf{lcp}_w(\mathsf{SSA}_P[i - 1], \mathsf{SSA}_P[i])$.

Let $1 \leq \tau \leq n$ be a parameter called the *sampling rate*. When, $P = [0..n - 1]^\tau_p$, for some $0 \leq p < \tau \leq n$, SSA_P is called the evenly space sparse suffix array with sampling rate τ and offset p. Given an evenly spaced sparse suffix array SSA_P, we can compute in $O(\frac{n}{\tau})$ time, a representation of the sparse inverse suffix array ISA_P as an array X of size $O(\frac{n}{\tau})$ where $\mathsf{X}[\lfloor \mathsf{SSA}_P[i]/\tau \rfloor] = i$, i.e., $\mathsf{ISA}_P[i] = X[\lfloor i/\tau \rfloor]$ for all $i \in P$. By directly applying the algorithm of Kasai et al. [9], SLCP_P can be computed from SSA_P and (the representation of) ISA_P in $O(n)$ time and $O(\frac{n}{\tau})$ space.

3 Data structure and query computation

Our algorithms are based on the same observation as used in [2].

▶ **Observation 3** ([2]). *For any positions $i, j, k \in [0..n-1]$ if $\mathsf{lcp}(j,k) \geq \mathsf{lcp}(i,j)$ then, $\mathsf{lcp}(i,j) = \min\{\mathsf{lcp}(i,k), \mathsf{lcp}(j,k)\}$.*

The observation allows us to reduce the computation of lcp values between a pair of positions, to the computation of lcp values between another pair of values, both from a specific subset of positions. For each specific position i, called a sampled position, and for each such subset S, a position $\pi(i,S) = \arg\max_{i' \in S}\{\mathsf{lcp}(i,i')\}$ is precomputed. The idea is that the size of S gets smaller after each reduction, therefore giving a bound on the query time.

▶ **Corollary 4.** *For any pair of positions $i \in S \subseteq [0..n-1]$ and $j \in [0..n-1]$, $\mathsf{lcp}(i,j) = \min\{\mathsf{lcp}(i,\pi(j,S)), \mathsf{lcp}(j,\pi(j,S))\}$.*

3.1 Bille et al.'s data structure

We first introduce a slightly modified version of the deterministic data structure by Bille et al. [2] that uses $O(\frac{n}{\tau})$ space and allows queries in $O(\tau \log^2 \frac{n}{\tau})$ time, where τ is a parameter in the range $1 \leq \tau \leq n$. We note that the modifications do not affect the asymptotic complexities.

Let $t = \tau \left\lceil \log \frac{n}{\tau} \right\rceil$, $p = (n-1) \bmod t$ and let $\mathcal{P} = [0..n-1]_p^t$ be the set of positions called *sampled positions*. The data structure of [2] to compute $\mathsf{lcp}(i,j)$ for any $0 \leq i < j < n$ consists of two main parts, one for when $j - i \geq t$, and the other for when $j - i < t$. Since we will use the latter part as is, we will only describe the former. The query time, space, and preprocessing time of the latter part are respectively, $O(\tau \log \frac{n}{\tau})$, $O(\frac{n}{\tau})$, and $O(n)$ (see Section 2 of [2]).

Consider a full binary tree where the root corresponds to the interval $[0..n-1]$, and for any node, the left and right children split their parent interval almost evenly, but assuring that the right-most position in the left child is a sampled position. Thus, there will be $\lceil n/t \rceil$ leaves corresponding to intervals of size t (except perhaps for the leftmost interval which may be smaller), and the height of the tree is $O(\log \frac{n}{t})$. For any internal node v in the tree, let I_v denote its corresponding interval, and $\ell(v), r(v)$ respectively the left and right children of v. For all sampled positions $i \in I_{r(v)} \cap \mathcal{P}$, a position $\pi(i, I_{\ell(v)}) = \arg\max_{i' \in I_{\ell(v)}}\{\mathsf{lcp}(i,i')\}$ and a value $L(i, I_{\ell(v)}) = \mathsf{lcp}(i, \pi(i, I_{\ell(v)}))$ are computed and stored. The size of the data structure is therefore $O(\frac{n}{t} \log \frac{n}{t}) = O(\frac{n}{\tau})$.

Assume w.l.o.g. that $j > i$. A query for $\mathsf{lcp}(i,j)$ with $j - i \geq t$ is computed as follows. First, compare up to $\delta < t$ characters of $w[i..]$ and $w[j..]$ until we encounter a mismatch, in which case we obtain an answer, or $j + \delta$ is a sampled position. Let I_v be the interval such that $i + \delta \in I_{\ell(v)}$ and $j + \delta \in I_{r(v)}$. From the preprocessing, we obtain a position $\pi(j + \delta, I_{\ell(v)}) \in I_{\ell(v)}$, which, from Corollary 4, satisfies:

$$\begin{aligned}
\mathsf{lcp}(i,j) &= \delta + \mathsf{lcp}(i+\delta, j+\delta) \\
&= \delta + \min\{\mathsf{lcp}(i+\delta, \pi(j+\delta, I_{\ell(v)})), \mathsf{lcp}(j+\delta, \pi(j+\delta, I_{\ell(v)}))\} \\
&= \delta + \min\{\mathsf{lcp}(i+\delta, \pi(j+\delta, I_{\ell(v)})), L(j+\delta, I_{\ell(v)})\}
\end{aligned}$$

Thus, the problem can be reduced to computing $\mathsf{lcp}(i + \delta, \pi(j + \delta, I_{\ell(v)}))$, where both $i + \delta, \pi(j + \delta, I_{\ell(v)}) \in I_{\ell(v)}$, and we apply the algorithm recursively. Note that if $j \in I_{r(v)}$ we have, from the definition of the intervals, that $j + \delta \in I_{r(v)}$, so each recursion takes us

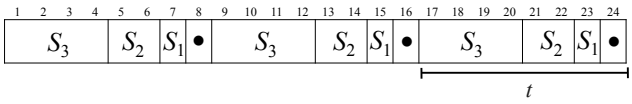


Figure 1 Sets S_k, with $k = 1, 2, 3$, for the sampled positions specified by black dots.

further down the tree. When an interval corresponding to a leaf node is reached, we have that $j - i < t$ and use the other data structure (for a description of which we refer the reader to [2]). Since we compare up to t characters at each level, the total query time is $O(t \log \frac{n}{t}) = O(\tau \log^2 \frac{n}{\tau})$.

3.2 New data structure

Let $t = \tau \lceil \log \tau \rceil$, $p = (n-1) \bmod t$, and let $\mathcal{P} = [0..n-1]^t_p$ be the set of sampled positions. Instead of considering a hierarchy of intervals of positions, we classify the positions according to their distance to the closest sampled position to their right. Define $S_k = \{i \mid (i+d) \bmod t = p, d \in ([2^{k-1}..2^k - 1] \cap [1..t-1])\}$ for $k = 1, \ldots, \lceil \log t \rceil$ (see also Figure 1).

The preprocessing computes and stores for each sampled position $i \in \mathcal{P}$ and each S_k, a position $\pi(i, S_k) = \arg\max_{i' \in S_k} \{\mathsf{lcp}(i, i')\}$, and a value $L(i, S_k) = \mathsf{lcp}(i, \pi(i, S_k))$. Also, $\mathsf{SLCP}_{\mathcal{P}}$ is computed and preprocessed for range minimum queries so that for any $i, j \in \mathcal{P}$, $\mathsf{lcp}(i, j)$ can be computed in constant time. Thus, the space required for the data structure is $O(\frac{n}{t} \log t) = O(\frac{n}{\tau})$.

A value $\mathsf{lcp}(i, j)$ is computed as follows. First, compare up to δ characters of $w[i..]$ and $w[j..]$ until we encounter a mismatch, in which case we obtain an answer, or, either $i+\delta$ or $j+\delta$ is a sampled position. If both $i+\delta$ and $j+\delta$ are sampled positions, $\mathsf{lcp}(i, j) = \delta + \mathsf{lcp}(i+\delta, j+\delta)$ can be answered in constant time. Assume w.l.o.g. that only $j + \delta$ is a sampled position, and let k be such that $i + \delta \in S_k$. Then, from Corollary 4 and the preprocessing, we have

$$\begin{aligned} \mathsf{lcp}(i, j) &= \delta + \mathsf{lcp}(i + \delta, j + \delta) \\ &= \delta + \min\{\mathsf{lcp}(i + \delta, \pi(j + \delta, S_k)), \mathsf{lcp}(j + \delta, \pi(j + \delta, S_k))\} \\ &= \delta + \min\{\mathsf{lcp}(i + \delta, \pi(j + \delta, S_k)), L(j + \delta, S_k)\} \end{aligned}$$

and the problem has been reduced to computing $\mathsf{lcp}(i+\delta, \pi(j+\delta, S_k))$ where both $i+\delta, \pi(j+\delta, S_k) \in S_k$, and the processes are repeated. Notice that in the next step, at least 2^{k-1} characters are compared until one of the two positions becomes a sampled position. This implies that the remaining distance to the closest sampled position of the other position will be at most $2^{k-1} - 1$, and thus the position will be in $S_{k'}$ for some $k' \leq k - 1$. Therefore, the process will only be repeated at most $\lceil \log t \rceil$ times. Because the number of characters compared in each step is bounded by t and is at least halved every step, the total number of character comparisons, and thus the query time, is $O(t) = O(\tau \log \tau)$.

3.3 Combining the structures

We can combine the structures described in Sections 3.1 and 3.2, to achieve $O(\tau \log \frac{n}{\tau})$ query time using $O(\frac{n}{\tau})$ space for $1 \leq \tau \leq n$. Furthermore, we can achieve $O(\tau \min\{\log \tau, \log \frac{n}{\tau}\})$ query time by choosing the better structure depending on τ. More precisely, when $\tau \leq \frac{n}{\tau}$ (i.e., $\tau \leq \sqrt{n}$), we simply use the structure of Section 3.2, and when $\tau \geq \frac{n}{\tau}$ (i.e., $\tau \geq \sqrt{n}$), we use the combined structure. Thus, we assume below that $\tau \geq \frac{n}{\tau}$.

Let $t = \tau \lceil \log \frac{n}{\tau} \rceil$, $p = (n-1) \bmod t$ and let $\mathcal{P} = [0..n-1]^t_p$ be the set of positions called *sampled positions*. We consider both the structures described in Section 3.1 and Section 3.2, with the following modifications. Let $d_t = 2^{\lceil \log t \rceil - \lceil \log \frac{n}{t} \rceil} = O(\frac{t^2}{n})$. For Bille et al.'s data structure, we make two modifications. First, for each node I_v and sampled position $i \in I_{r(v)} \cap \mathcal{P}$, we only consider points that are at most d_t from the closest sampled position to the right, i.e., instead of $\pi(i, I_{\ell(v)})$ and $L(i, I_{\ell(v)})$, we compute and store a position $\pi(i, I_{\ell(v)} \cap D) = \arg\max_{i' \in I_{\ell(v)} \cap D} \{\mathsf{lcp}(i, i')\}$ and a value $L(i, I_{\ell(v)} \cap D)$, where $D = \{i' \mid (i' + d) \bmod t = p, 0 \leq d < d_t\}$. In addition to this, we compute and store for all sampled position $i \in I_{\ell(v)} \cap \mathcal{P}$, a position $\pi(i, I_{r(v)} \cap D) = \arg\max_{i' \in I_{r(v)} \cap D} \{\mathsf{lcp}(i, i')\}$ and $L(i, I_{r(v)} \cap D) = \mathsf{lcp}(i, \pi(i, I_{r(v)} \cap D))$. This will only double the total size of the structure and thus the space usage remains $O(\frac{n}{\tau})$. For the new data structure, we keep the definitions of $\pi(i, S_k)$ and $L(i, S_k)$, but store these values only for $k = \lceil \log t \rceil - \lceil \log \frac{n}{t} \rceil, \ldots, \lceil \log t \rceil$. Thus, although the value of t has changed, the total size of the data structure is still $O(\frac{n}{t} \log \frac{n}{t}) = O(\frac{n}{\tau})$.

Queries $\mathsf{lcp}(i, j)$ are answered as follows: First use the new data structure recursively using the original algorithm until the problem is reduced to a query between a sampled position and another position not in any S_k ($k \in [\lceil \log t \rceil - \lceil \log \frac{n}{t} \rceil .. \lceil \log t \rceil]$). This means that the distance from either of the query positions to the closest sampled position is at most d_t. The total number of character comparisons conducted is $O(t) = O(\tau \log \frac{n}{\tau})$. Then, we switch to Bille et al.'s structure using the original algorithm with the exception that we continue until *either* $i + \delta$ *or* $j + \delta$ (instead of just $j + \delta$) is a sampled position when comparing up to δ characters of $w[i..]$ and $w[j..]$. Since the distance to the closest sampling position is at most $O(\frac{t^2}{n})$ and by definition of $\pi(i, I_{\ell(v)} \cap D)$ and $\pi(i, I_{r(v)} \cap D)$, we have that this condition holds for all following recursive calls. Thus, at most $O(\frac{t^2}{n})$ character comparisons will be conducted at each level, for a total of $O(\frac{t^2}{n} \log \frac{n}{t}) = O(t(\frac{n}{t})^{-1} \log \frac{n}{t}) = O(t) = O(\tau \log \frac{n}{\tau})$.

4 Building the structures

Bille et al. [2] describe a preprocessing that runs in $O(n^2)$ time[1] and $O(\frac{n}{\tau})$ space. Here, we show that this can be reduced to $O(\tau n + n \log \frac{n}{\tau})$ time using the same space. While the algorithm of [2] builds the sparse suffix array containing only the suffixes starting at sampled positions and applies pattern matching, our trick is to build a sparse suffix array and sparse LCP array that includes other suffixes as well, in several (namely τ) rounds, so that the suffixes with maximum LCP with respect to each sampled position can be found by scans of the suffix array.

For integer alphabets, sparse suffix arrays and sparse LCP arrays can be constructed in $O(n)$ time if $O(n)$ space is allowed, simply by first building the (normal) suffix array and LCP array and removing the unwanted elements. For constant size alphabets, the evenly spaced sparse suffix array and sparse LCP array with sampling rate τ can be constructed in $O(n)$ time and $O(\frac{n}{\tau})$ space [8]. However, when the alphabet size σ is not constant, this is $O(n \log \sigma)$ time and $O(\frac{n}{\tau})$ space, since the computation is based on character comparisons. (Notice that simple application of linear time algorithms for computing the suffix array for

[1] However, we believe the analysis in Section 2.5 of [2] is not entirely correct; although the size of $|I|$ is halved at each level, their numbers double, and so the time complexity should be $O(n \cdot n + n \cdot (n/2) \cdot 2 \cdots + n \cdot (n/t) \cdot t) = O(n^2 \log \frac{n}{t})$ time. Also, they assume that the evenly spaced sparse suffix array can be constructed in $O(n)$ time and $O(\frac{n}{\tau})$ space for the integer alphabet. However, the paper they cite assumes a constant size alphabet and to the best of our knowledge, we do not know of an algorithm achieving such space-time trade-off.

the meta string will not achieve $O(n)$ time and $O(\frac{n}{\tau})$ space, since the use of radix sort implies $\Omega(\sigma)$ space for the buckets.) Repeated τ times, this results in $O(n\tau \log \sigma)$ time using $O(\frac{n}{\tau})$ space.

We first describe a technique to compute the sparse suffix array and the corresponding LCP array that contains two sets of evenly spaced suffixes, namely for offsets p and q, and to repeat this τ times, namely for offsets $p = (n-1) \bmod \tau$ and $q = (n-1) \bmod \tau, \ldots, (n-\tau) \bmod \tau$, so that the total time for their construction is $O(n\tau)$ time using $O(\frac{n}{\tau})$ space. Then, we describe the construction of the data structures of Section 3 using this technique.

4.1 Common tools

For any string (or meta-string) w and $0 \leq i < |w|$, let CA_w denote an array containing a permutation of $[0..|w|-1]$ such that $w[\mathsf{CA}_w[i]] \leq w[\mathsf{CA}_w[j]]$ for any $0 \leq i < j < |w|$, i.e., CA_w is an array of positions sorted according to the character at each position. (Note that CA_w is not necessarily unique.)

▶ **Lemma 5.** *For any string w and $0 \leq p < \tau$, $\mathsf{CA}_{\hat{w}_{\tau,p}}$ can be computed in $O(n \log \tau)$ time using $O(\frac{n}{\tau})$ space.*

Proof. Since each character of w can be represented in $O(\log n)$ bits, the length of each meta-character of $\hat{w}_{\tau,p}$ is $O(\tau \log n)$ bits. We simply use LSD radix sort with a bucket size of $\frac{n}{\tau}$, i.e., we bucket sort using $\log(n/\tau)$ bits at a time. Thus, $O(\frac{\tau \log n}{\log(n/\tau)})$ rounds of bucket sort is conducted on $\frac{n}{\tau}$ items, resulting in $O(\frac{n \log n}{\log(n/\tau)}) = O(\frac{n(\log \tau + \log(n/\tau))}{\log(n/\tau)}) = O(n \log \tau)$ time giving the result. ◀

▶ **Lemma 6.** *For any string w and $0 \leq p < \tau$, $\mathsf{CA}_{\hat{w}_{\tau,p}}$ can be computed from $\mathsf{CA}_{\hat{w}_{\tau,p'}}$, where $p' = (p+1) \bmod \tau$, in $O(\frac{n}{\tau} \log \tau)$ time and $O(\frac{n}{\tau})$ space.*

Proof. We simply continue the LSD radix sort, and do an extra $O(\frac{\log n}{\log(n/\tau)})$ rounds of bucket sort for the preceding character of each meta-character, which results in $O(\frac{n}{\tau} \cdot \frac{\log n}{\log(n/\tau)}) = O(\frac{n}{\tau} \cdot \frac{\log \tau + \log(n/\tau)}{\log(n/\tau)}) = O(\frac{n}{\tau} \log \tau)$ time. ◀

▶ **Lemma 7.** *For any string w, $0 \leq p, q < \tau$, let $P = [0..n-1]^{\tau}_{p}$ and $Q = [0..n-1]^{\tau}_{q}$. Given $\mathsf{CA}_{\hat{w}_{\tau,p}}$ and $\mathsf{CA}_{\hat{w}_{\tau,q}}$, $\mathsf{SSA}_{P \cup Q}$ and $\mathsf{SLCP}_{P \cup Q}$ can be computed in $O(n)$ time using $O(\frac{n}{\tau})$ space.*

Proof. We first compute $\mathsf{CA}_{w'}$ for meta-string $w' = \hat{w}_{\tau,p} 0 \hat{w}_{\tau,q}$. This can be done in $O(n)$ time and $O(\frac{n}{\tau})$ space by merging $\mathsf{CA}_{\hat{w}_{\tau,p}}$ and $\mathsf{CA}_{\hat{w}_{\tau,q}}$, (and adding $|\hat{w}_{\tau,p}0|$ to entries in $\mathsf{CA}_{\hat{w}_{\tau,q}}$) since each comparison of meta characters can be done in $O(\tau)$ time. Using $\mathsf{CA}_{w'}$, we then rename the characters of w' and create a string w^* such that $w^*[i] = |\{w'[j] \mid w'[j] < w'[i], 0 \leq j < |w'|\}| + 1$, in $O(n)$ time and $O(\frac{n}{\tau})$ space. Since w^* consists of integers bounded by its length, we can apply any linear-time suffix sorting algorithm and compute SA_{w^*} and LCP_{w^*} in $O(\frac{n}{\tau})$ time and space. As the lexicographic order of suffixes of w^* (except for $\mathsf{SSA}_{w^*}[0] = |\hat{w}_{\tau,p}|$) corresponds to the lexicographic order of suffixes of w that start at positions in $P \cup Q$, we can obtain $\mathsf{SSA}_{P \cup Q}$ from SA_{w^*} by appropriately translating the indices. More precisely, for $1 \leq i < |w'|$, let $\mathsf{SSA}_{w^*}[i] = j$. If $0 \leq j < |\hat{w}_{\tau,p}|$, then $\mathsf{SSA}_{P \cup Q}[i-1] = j\tau + p$, and otherwise (if $|\hat{w}_{\tau,p}0| \leq j < |w'|$), then $\mathsf{SSA}_{P \cup Q}[i-1] = (j - |\hat{w}_{\tau,p}0|)\tau + q$. We can also obtain $\mathsf{SLCP}_{P \cup Q}$ from LCP_{w^*} by multiplying a factor of τ and doing up to τ character comparisons per pair of adjacent suffixes in the suffix array, in a total of $O(n)$ time. ◀

▶ **Corollary 8.** *For any string w, let $p = n \bmod \tau$. The arrays $\mathsf{SSA}_{P \cup Q}$ and $\mathsf{SLCP}_{P \cup Q}$ can be computed successively for each $q = p, (p-1) \bmod \tau, \ldots, (p-\tau+1) \bmod \tau$, where $P = [0..n-1]^{\tau}_{p}$ and $Q = [0..n-1]^{\tau}_{q}$, in $O(n\tau)$ time using $O(\frac{n}{\tau})$ space.*

Proof. For $p = q$, we first compute $\mathsf{CA}_{\hat{w}_{\tau,p}} = \mathsf{CA}_{\hat{w}_{\tau,q}}$ using Lemma 5. By applying Lemma 6, we can successively compute $\mathsf{CA}_{\hat{w}_{\tau,q}}$ for $q = (p-1) \bmod \tau, \ldots, (p-\tau+1) \bmod \tau$. Thus, with Lemma 7, we can successively compute $\mathsf{SSA}_{P\cup Q}$ and $\mathsf{SLCP}_{P\cup Q}$ in $O(n\tau)$ total time and $O(\frac{n}{\tau})$ space. ◀

4.2 Faster construction of Bille et al.'s data structure

We show that Bille et al.'s data structure can be constructed in $O(n\tau + n\log\frac{n}{\tau})$ time using $O(\frac{n}{\tau})$ space. Let $p = (n-1) \bmod \tau$. Using Corollary 8, we successively compute $\mathsf{SSA}_{P\cup Q}$ and $\mathsf{SLCP}_{P\cup Q}$ for each $q = p, (p-1) \bmod \tau, \ldots, (p-\tau+1) \bmod \tau$, where $P = [0..n-1]^{\tau}_{p}$ and $Q = [0..n-1]^{\tau}_{q}$. This can be done in a total of $O(n\tau)$ time, and $O(\frac{n}{\tau})$ space. Recall that $t = \tau\lceil\log\frac{n}{\tau}\rceil$, and $\mathcal{P} = [0..n-1]^{t}_{p'}$, where $p' = (n-1) \bmod t$. Since t is a multiple of τ, we have $\mathcal{P} \subseteq P$.

For each q we do the following: $\mathsf{SLCP}_{P\cup Q}$ is preprocessed in $O(\frac{n}{\tau})$ time and space to answer RMQ in constant time, thus allowing us to compute $\mathsf{lcp}(i,j)$ for any $i, j \in P \cup Q$ in constant time. For any interval $I_v \subseteq [0..n-1]$ corresponding to a node in the binary tree let $I^q_v = I_v \cap (P \cup Q)$. Note that for $I_{\mathsf{root}} = [0..n-1]$, $\mathsf{SSA}_{I^q_{\mathsf{root}}} = \mathsf{SSA}_{P\cup Q}$. Now, for any node I_v, assume that $\mathsf{SSA}_{I^q_v}$ is already computed. By simple linear time scans on $\mathsf{SSA}_{I^q_v}$, we can obtain, for each sampled position $i = \mathsf{SSA}_{I^q_v}[x] \in I^q_{r(v)} \cap \mathcal{P}$, the two suffixes $\mathsf{SSA}_{I^q_v}[j^-], \mathsf{SSA}_{I^q_v}[j^+] \in I^q_{\ell(v)} \cap Q$ which are lexicographically closest to i, i.e., $j^- = \max\{j < x \mid \mathsf{SSA}_{I^q_v}[j] \in I^q_{\ell(v)} \cap Q\}$, $j^+ = \min\{j > x \mid \mathsf{SSA}_{I^q_v}[j] \in I^q_{\ell(v)} \cap Q\}$, if they exist. Then, the larger of $\mathsf{lcp}(i, \mathsf{SSA}_{I^q_v}[j^-])$ and $\mathsf{lcp}(i, \mathsf{SSA}_{I^q_v}[j^+])$ gives $\pi(i, I^q_{\ell(v)} \cap Q) = \arg\max_{i' \in I^q_{\ell(v)} \cap Q}\{\mathsf{lcp}(i,i')\}$ and $L(i, I^q_{\ell(v)} \cap Q) = \mathsf{lcp}(i, \pi(i, I^q_{\ell(v)} \cap Q))$. Since $i, \mathsf{SSA}_{I^q_v}[j^+], \mathsf{SSA}_{I^q_v}[j^-] \in P \cup Q$, these values can be computed in constant time, which is $O(|I^q_v|)$ total time for all sampled positions $i \in I^q_{r(v)} \cap \mathcal{P}$. Next, for the child intervals, $\mathsf{SSA}_{I^q_{\ell(v)}}$ and $\mathsf{SSA}_{I^q_{r(v)}}$ can be computed in $O(|I^q_v|)$ time by a simple scan on $\mathsf{SSA}_{I^q_v}$, and the computation is performed recursively for each child. Since the union of $I^q_v \cap Q$ over all q is I_v, we have $\pi(i, I_{\ell(v)}) = \pi(i, I^{\hat{q}}_{\ell(v)})$ and $L(i, I_{\ell(v)}) = L(i, I^{\hat{q}}_{\ell(v)})$, where $\hat{q} = \arg\max_{0 \le q' < \tau}\{\mathsf{lcp}(i, \pi(i, I^{q'}_{\ell(v)} \cap Q))\}$, so we can obtain $\pi(i, I_{\ell(v)})$ and $L(i, I_{\ell(v)})$ for each sampled position i and interval I_v by repeating the above process for each q.

Since the processing at each node is linear in the size of the arrays whose total size at a given level is $O(\frac{n}{\tau})$, the total time for the recursion is $O(\frac{n}{\tau}\log\frac{n}{\tau})$ for each q. Thus in total, the preprocessing can be done in $O(n\tau + n\log\frac{n}{\tau})$ time.

▶ **Theorem 9.** *For any string of length n and integer $1 \le \tau \le n$, a data structure of size $O(\frac{n}{\tau})$ can be constructed in $O(n\tau + n\log\frac{n}{\tau})$ time using $O(\frac{n}{\tau})$ space, such that for any $0 \le i, j < n$, $\mathsf{lcp}(i,j)$ can be answered in $O(\tau\log^2\frac{n}{\tau})$ time.*

4.3 Fast construction of new data structure

Let $p = (n-1) \bmod \tau$. Using Corollary 8, we successively compute $\mathsf{SSA}_{P\cup Q}$ and $\mathsf{SLCP}_{P\cup Q}$ for each $q = p, (p-1) \bmod \tau, \ldots, (p-\tau+1) \bmod \tau$, where $P = [0..n-1]^{\tau}_{p}$ and $Q = [0..n-1]^{\tau}_{q}$. This can be done in a total of $O(n\tau)$ time, and $O(\frac{n}{\tau})$ space. Recall that $t = \tau\lceil\log\tau\rceil$, and $\mathcal{P} = [0..n-1]^{t}_{p'}$, where $p' = (n-1) \bmod t$. Since t is a multiple of τ, we have $\mathcal{P} \subseteq P$.

For each q we do the following: $\mathsf{SLCP}_{P\cup Q}$ is preprocessed in $O(\frac{n}{\tau})$ time and space to answer RMQ in constant time, thus allowing us to compute $\mathsf{lcp}(i,j)$ for $i, j \in P \cup Q$ in constant time. Let $S^q_k = S_k \cap Q$ for any $1 \le k \le \lceil\log t\rceil$. Next, we conduct for each $k = 1, \ldots, \lceil\log t\rceil$, linear time scans on $\mathsf{SSA}_{P\cup Q}$ so that for each sampled position $i = \mathsf{SSA}_{P\cup Q}[x] \in \mathcal{P}$, the two suffixes $\mathsf{SSA}_{P\cup Q}[j^-], \mathsf{SSA}_{P\cup Q}[j^+] \in S^q_k$ which are lexicographically closest to i, i.e., $j^- = \max\{j < x \mid \mathsf{SSA}_{P\cup Q}[j] \in S^q_k\}$, $j^+ = \min\{j > x \mid \mathsf{SSA}_{P\cup Q}[j] \in S^q_k\}$, if they exist. Then, the

larger of $\mathsf{lcp}(i, \mathsf{SSA}_{P\cup Q}[j^-])$ and $\mathsf{lcp}(i, \mathsf{SSA}_{P\cup Q}[j^+])$ gives $\pi(i, S_k^q) = \arg\max_{i' \in S_k^q}\{\mathsf{lcp}(i, i')\}$. Since $i, \mathsf{SSA}_{P\cup Q}[j^+], \mathsf{SSA}_{P\cup Q}[j^-] \in P \cup Q$, these values can be computed in constant time, resulting in a total of $O(\frac{n}{\tau}\log\tau)$ time for all i and k. Since the union of S_k^q over all q is S_k, we have $\pi(i, S_k) = \pi(i, S_k^{\hat{q}})$ and $L(i, S_k) = L(i, S_k^{\hat{q}})$, where $\hat{q} = \arg\max_{0\le q'<\tau}\{\mathsf{lcp}(i, \pi(i, S_k^{q'}))\}$, so we can obtain $\pi(i, S_k)$ and $L(i, S_k)$ for each sampled position i and S_k by repeating the above process for each q, taking $O(n\log\tau)$ time. Thus, the total time for preprocessing, dominated by Corollary 8, is $O(n\tau)$.

▶ **Theorem 10.** *For any string of length n and integer $1 \le \tau \le \frac{n}{\log n}$, a data structure of size $O(\frac{n}{\tau})$ can be constructed in $O(n\tau)$ time using $O(\frac{n}{\tau})$ space, such that for any $0 \le i, j < n$, $\mathsf{lcp}(i, j)$ can be answered in $O(\tau\log\tau)$ time.*

4.4 Fast construction of combined data structure

The construction of the combined data structure is done using the same algorithms as described in Sections 4.2 and 4.3, with only minor modifications. For Bille et al.'s data structure, we only need to consider in addition to sampled positions, the positions in $D = \{i' \mid (i' + d) \bmod t = p, 0 \le d < d_t\}$ due to the modification introduced for the combination. This reduces the array sizes (and thus the computation time) needed for the computation of $\pi(i, I_{\ell(v)})$ and $\pi(i, I_{r(v)})$ (and $L(i, I_{\ell(v)})$ and $L(i, I_{r(v)})$) to $O(\frac{n}{t} + \frac{n}{t}\cdot\frac{t^2}{n}\cdot\frac{1}{\tau}) = O(\frac{n}{t} + \frac{t}{\tau}) = O(\frac{n}{\tau\log\frac{n}{\tau}} + \log\frac{n}{\tau})$ for a total of $O(\frac{n}{\tau} + \log^2\frac{n}{\tau}) = O(\frac{n}{\tau})$ for all levels, and for all q, we get $O(n)$ time. Thus, the total time for preprocessing is now dominated by Corollary 8, and is $O(n\tau)$.

▶ **Theorem 11.** *For any string of length n and integer $1 \le \tau \le n$, a data structure of size $O(\frac{n}{\tau})$ can be constructed in $O(n\tau)$ time using $O(\frac{n}{\tau})$ space, such that for any $0 \le i, j < n$, $\mathsf{lcp}(i, j)$ can be answered in $O(\tau\log\frac{n}{\tau})$ time.*

As noted previously, since $\tau \le \frac{n}{\tau}$ when $\tau \le \sqrt{n}$, and $\tau \ge \frac{n}{\tau}$ when $\tau \ge \sqrt{n}$, we get the following by simply choosing the data structure of Theorems 10 and 11, depending on the value of τ.

▶ **Corollary 12.** *For any string of length n and integer $1 \le \tau \le n$, a data structure of size $O(\frac{n}{\tau})$ can be constructed in $O(n\tau)$ time using $O(\frac{n}{\tau})$ space, such that for any $0 \le i, j < n$, $\mathsf{lcp}(i, j)$ can be answered in $O(\tau\min\{\log\tau, \log\frac{n}{\tau}\})$ time.*

5 Applications

Using the proposed data structure, the lexicographic order between two arbitrary suffixes can be computed in $O(\tau\min\{\log\tau, \log\frac{n}{\tau}\})$ time using $O(\frac{n}{\tau})$ space. Thus, using any $O(n\log n)$ comparison based sorting algorithm, we can compute the suffix array of a string of length n in $O(\min\{\log\tau, \log\frac{n}{\tau}\}n\tau\log n)$ time using $O(\frac{n}{\tau})$ working space, excluding the input and output. The best known deterministic space/time trade-off is $O(n\tau^2)$ time (for $1 \le \tau \le \sqrt[4]{n}$) using the same space [7], and our algorithm is better when $\tau = \Omega(\log^{1+\epsilon} n)$ for any $\epsilon > 0$.

Acknowledgements. The authors thank the anonymous reviewers for careful reading of the paper and for helpful comments.

References

1. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proc. Latin'00*, pages 88–94, 2000.
2. Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *Proc. CPM 2015*, pages 65–76, 2015.
3. Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. *J. Discrete Algorithms*, 25:42–50, 2014.
4. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
5. Juha Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theor. Comput. Sci.*, 387(3):249–257, 2007.
6. Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. CPM'13*, volume 7922 of *Lecture Notes in Computer Science*, pages 189–200. Springer, 2013.
7. Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.
8. Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Proc. COCOON'96*, pages 219–230, 1996.
9. Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. CPM'01*, pages 181–192, 2001.
10. Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proc. CPM'03*, pages 186–199, 2003.
11. Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proc. CPM'03*, pages 200–210, 2003.
12. Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
13. Simon J. Puglisi and Andrew Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proc. ISAAC'08*, volume 5369 of *Lecture Notes in Computer Science*, pages 124–135. Springer, 2008.

Space-Efficient Dictionaries for Parameterized and Order-Preserving Pattern Matching

Arnab Ganguly[*1], Wing-Kai Hon[†2], Kunihiko Sadakane[3], Rahul Shah[4], Sharma V. Thankachan[5], and Yilin Yang[6]

1 **School of Electrical Engineering and Computer Science, Louisiana State University, USA**
agangu4@lsu.edu
2 **Department of Computer Science, National Tsing Hua University, Taiwan**
wkhon@cs.nthu.edu.tw
3 **Department of Mathematical Informatics, University of Tokyo, Japan**
sada@mist.i.u-tokyo.ac.jp
4 **School of Electrical Engineering and Computer Science, Louisiana State University, USA; and**
National Science Foundation, USA
rahul@csc.lsu.edu, rahul@nsf.gov
5 **School of Computational Science and Engineering, Georgia Institute of Technology, USA**
sharma.thankachan@gatech.edu
2 **Department of Computer Science, National Tsing Hua University, Taiwan**
yilinyang@cs.nthu.edu.tw

Abstract

Let S and S' be two strings, having the same length, over a totally-ordered alphabet. We consider the following two variants of string matching.

- *Parameterized Matching:* The characters of S and S' are partitioned into static characters and parameterized characters. The strings are a parameterized match iff the static characters match exactly, and there exists a one-to-one function which renames the parameterized characters in S to those in S'.
- *Order-Preserving Matching:* The strings are an order-preserving match iff for any two integers $i, j \in [1, |S|]$, $S[i] \prec S[j] \iff S'[i] \prec S'[j]$, where $\prec$ denotes the precedence order of the alphabet.

Let $\mathcal{P}$ be a collection of d patterns $\{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_d\}$ of total length n characters, which are chosen from a totally-ordered alphabet Σ. Given a text T, also over Σ, we consider the dictionary indexing problem under the above definitions of string matching. Specifically, the task is to index $\mathcal{P}$, such that we can report all positions j (called *occurrences*) where at least one of the patterns $\mathsf{P}_i \in \mathcal{P}$ is a parameterized match (resp. an order-preserving match) with the same-length substring of T starting at j. Previous best-known indexes occupy $O(n \log n)$ bits, and can report all *occ* occurrences in $\mathcal{O}(|T| \log |\Sigma| + occ)$ time. We present space-efficient indexes that occupy $\mathcal{O}(n \log |\Sigma| + d \log n)$ bits, and reports all *occ* occurrences in $\mathcal{O}(|T|(\log |\Sigma| + \log_{|\Sigma|} n) + occ)$ time for parameterized matching, and in $\mathcal{O}(|T| \log n + occ)$ time for order-preserving matching.

1998 ACM Subject Classification F.2.2 Pattern Matching

[*] The work of Arnab Ganguly was supported by National Science Foundation Grants CCF–1017623 and CCF–1218904.
[†] The work of Wing-Kai Hon was supported by National Science Council Grants 102-2221-E-007-068-MY3 and 105-2918-I-007-006.

Keywords and phrases Parameterized Matching, Order-preserving Matching, Dictionary Indexing, Aho-Corasick Automaton, Sparsification

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.2

1 Introduction

Designing succinct data-structures for the classical pattern matching problem of finding all occurrences of a pattern P in a fixed text T can be traced back to the seminal work of Grossi and Vitter [14], Ferragina and Manzini [8], and Sadakane [26]. This established an active research area of designing succinct data structures. (See [25] for a comprehensive survey.) The focus was now on either improving these initial breakthroughs [5, 9, 10, 11, 13, 22, 23, 27], or on designing succinct data structures for other variants [4, 6, 12, 17, 24, 30]. Dictionary matching, a typical example of these variants, is a classical problem in string matching and is defined as follows. Let $\mathcal{P}$ be a collection of d patterns $\{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_d\}$ of total length n characters which are chosen from a totally-ordered alphabet Σ of size σ. Given a text T, also over Σ, the task is to report all positions j (called *occurrences*) such that at least one of the patterns $\mathsf{P}_i \in \mathcal{P}$ exactly matches an equal-length substring of T that starts at j. The classical solution for this problem is the Aho-Corasick (AC) automaton [1] which occupies $\Theta(m \log m)$ bits of space, where $m \leq n + 1$ is the number of states in the automaton, and finds all *occ* occurrences in time $\mathcal{O}(|T| \log \sigma + occ)$. The query complexity can be improved to optimal $\mathcal{O}(|T| + occ)$ using perfect-hashing techniques. To the best of our knowledge, the first two succinct indexes for the problem are by Hon et al. [16] and Tam et al. [29]. Later, Belazzougui [4] presented an $m \log \sigma + \mathcal{O}(m) + \mathcal{O}(d \log(n/d))$ bit index with optimal $\mathcal{O}(|T| + occ)$ query time.

The first problem that we consider is popularly known as the *Parameterized Pattern Matching* problem. The problem has significant attention [2, 15, 18, 19, 21] since its inception by Baker in 1993 [3]. The alphabet Σ is partitioned into two disjoint sets: Σ_s containing static-characters (s-characters) and Σ_p containing parameterized characters (p-characters). Two strings S and S', both over Σ, are a parameterized match (p-match) iff $|S| = |S'|$ and there is a one-to-one function f such that $S[i] = f(S'[i])$. For any s-character $c \in \Sigma_s$, we have $f(c) = c$. Thus, for $\Sigma_s = \{A, B, C\}$ and $\Sigma_p = \{w, x, y, z\}$, the strings $AxBxCy$ and $AzBzCx$ are a p-match, but $AxBxCy$ and $AzBwCx$ are not. We consider the *Parameterized Dictionary Matching* problem which was introduced by Idury and Schäffer [18]. This is similar to the standard dictionary problem, just that Σ is partitioned into Σ_s and Σ_p, and we consider the p-matches of a pattern with the text. Idury and Schäffer presented an AC-automaton like solution which occupies $\mathcal{O}(m \log m) = \mathcal{O}(n \log n)$ bits and reports all *occ* occurrences in $\mathcal{O}(|T| \log \sigma + occ)$ time. The following theorem summarizes our contribution.

▶ **Theorem 1.** *By maintaining an index of $\mathcal{P}$ in $\mathcal{O}(n \log \sigma + d \log n)$ bits, all occ occurrences where a pattern in $\mathcal{P}$ and T are a p-match can be reported in $\mathcal{O}(|T|(\log \sigma + \log_\sigma n) + occ)$ time.*

The second problem we consider is a variant of the recently introduced *Order-Preserving Pattern Matching* problem [7, 20]. Two strings S and S' are an order-preserving match (o-match) iff $|S| = |S'|$ and for any two integers $i, j \in [1, |S|]$, we have $S[i] \prec S[j] \iff S'[i] \prec S'[j]$. Thus, for the alphabet $\{A, B, C, D\}$ with the total-order $A \prec B \prec C \prec D$, the string ABC is an o-match with BCD, but not with CDB. Likewise, AAB matches CCD, but does not match ABC. We consider the *Order-Preserving Dictionary Matching* problem introduced by Kim et al. [20]. As with the p-dictionary matching problem, the

match in this case is defined according to order-preserving matching. Kim et al. presented an AC-automaton like approach which occupies $\mathcal{O}(n \log n)$ bits and reports all occurrences in $\mathcal{O}(|T| \log \sigma + occ)$ time. The following theorem summarizes our contribution.

▶ **Theorem 2.** *By maintaining an index of $\mathcal{P}$ in $\mathcal{O}(n \log \sigma + d \log n)$ bits, all occ occurrences where a pattern in $\mathcal{P}$ and T are an o-match can be reported in $\mathcal{O}(|T| \log n + occ)$ time.*

1.1 Map

Our techniques are largely based on the sparsification technique of Hon et al. [16] for the classical dictionary matching problem. For a parameter Δ, this technique condenses every Δ characters of each pattern separately and then creates an AC-automaton for the condensed patterns. Likewise, the text is also condensed starting at a position i. Now the condensed text is matched in the AC-automaton, and all occurrences are reported. The occurrences reported in this run lie in the set $\{i, i + \Delta, i + 2\Delta, \dots\}$. All occurrences are reported by repeating the process for $i = 1, 2, 3, \dots, \Delta$. By properly choosing Δ, different trade-offs for index sizes and query time can be obtained. Broadly speaking, we use this technique to sparsify the AC-automaton based approaches of Idury and Schäffer [18] for p-dictionary matching and of Kim et al. [20] for o-dictionary matching. However, the sparsification technique does not immediately extend to the case of parameterized matching and order-preserving matching. For example, it is not clear whether a condensed alphabet has to be treated as a p-character or an s-character. Also, how do we define the one-to-one mapping? Similarly, how do we impose the total-order on the condensed alphabet in the case of order-preserving matching? A more serious issue is how to handle truncating of characters at the beginning of a currently matched text, which is essential for the AC-automaton based approaches of Idury and Schäffer and of Kim et al.

In Section 2, we first address the p-dictionary problem, and prove Theorem 1. In Section 3, using similar techniques, we arrive at Theorem 2.

2 Parameterized Dictionary Matching

We assume that the p-characters in $\mathsf{P}_i \in \mathcal{P}$ are from the set $\{0, 1, \dots, |\Sigma_p| - 1\}$. Also, the s-characters are disjoint from the set of integers. (The latter assumption can be easily removed by mapping the s-characters onto the set $\{|\Sigma_p|, |\Sigma_p| + 1, \dots, \sigma - 1\}$ such that the kth smallest s-character has value $|\Sigma_p| + k - 1$.) The patterns can be initially processed in $\mathcal{O}(n \log \sigma)$ time to ensure that these conditions hold.

2.1 Encoding Scheme

[3] introduced the following encoding scheme to enable matching of parameterized strings. Given a string S, obtain a string $\mathsf{prev}(S)$ by replacing the first occurrence of every p-character in S by 0 and any other occurrence by the difference in position from its previous occurrence. Thus, $\mathsf{prev}(A1B2A1C0) = A0B0A4C0$. Baker [3] showed that two strings S and S' are a p-match iff $\mathsf{prev}(S) = \mathsf{prev}(S')$. Although this scheme makes p-matching of strings easier to handle, for our purposes, it suffers from a drawback. Specifically, $\mathsf{prev}(S)$ is a string over an alphabet of size $\Theta(n)$ in the worst case, whereas the original alphabet size σ may be much smaller in comparison.

In order to alleviate this, we introduce the following encoding scheme, which is still simple and does not suffer from this drawback. Given a string S over Σ, let $c_0, c_1, \dots, c_k$ be the order in which every $c_i \in \Sigma_p$ appears in S. We obtain a string $\mathsf{pEncode}(S)$ by replacing every

occurrence of c_i by i in S. Thus, $\mathsf{pEncode}(A1B2A1C0) = A0B1A0C2$. By maintaining an integer-array of length $|\Sigma_p|$, we can compute $\mathsf{pEncode}(S)$ in $\mathcal{O}(|S|)$ time[1]. The following observations are immediate.

▶ **Observation 3.** *Two strings S and S' are a p-match iff $\mathsf{pEncode}(S) = \mathsf{pEncode}(S')$. A string S matches another string S' at a position i iff $\mathsf{pEncode}(S) = \mathsf{pEncode}(S'[i, i+|S|-1])$.*

▶ **Observation 4.** *For a string S, assume that the parameterized characters in $\mathsf{pEncode}(S)$ belong to the set $\{0, 1, 2, \ldots, |\Sigma_p| - 1\}$. Then, $\mathsf{pEncode}(S[i, |S|]) = \mathsf{pEncode}(\mathsf{pEncode}(S)[i, |S|])$.*

2.2 Overview

We design our index by classifying the patterns into *long* and *short* based on a parameter $\Delta = \lceil \log_\sigma n \rceil$. The patterns are encoded and maintained explicitly occupying $n \log \sigma$ bits in total. For short patterns (having length less than Δ), we create a trie and use a rather brute-force approach to find all occurrences. On the other hand, reporting the occurrences of long patterns (having length at least Δ) requires sophisticated (and more involved) indexing and querying techniques. Moving forward, when we refer to an occurrence, we imply both the position in the text where a pattern occurs and also the pattern itself. Also, we report all patterns that occur at a particular position. (The query process can be easily adapted to the case when only the position is to be reported.) Then, the set of occurrences of long patterns and short patterns are mutually disjoint and are handled separately. Specifically, we prove the following lemmas of which Theorem 1 is an immediate consequence.

▶ **Lemma 5.** *Let $\mathcal{P}$ be a dictionary consisting of d patterns, each having length at least $\lceil \log_\sigma n \rceil$. By indexing $\mathcal{P}$ in a data-structure occupying $\mathcal{O}(n \log \sigma + d \log n)$ bits, we can report all occ occurrences of the patterns in $\mathcal{O}(|T|(\log \sigma + \log_\sigma n) + occ)$ time.*

▶ **Lemma 6.** *Let $\mathcal{P}$ be a dictionary consisting of d patterns, each having length less than $\lceil \log_\sigma n \rceil$. By indexing $\mathcal{P}$ in a data-structure occupying $n \log \sigma + \mathcal{O}(d \log n)$ bits, we can report all occ occurrences of the patterns in $\mathcal{O}(|T|(\log \sigma + \log_\sigma n) + occ)$ time.*

We assume that no two patterns P_i and P_j exist such that $\mathsf{pEncode}(\mathsf{P}_i) = \mathsf{pEncode}(\mathsf{P}_j)$. For such patterns, we can keep only one pattern in the dictionary, and it is trivial to handle reporting of all patterns for an occurrence in the claimed space-time bounds. We also assume that the p-characters in T are from $\{0, 1, \ldots, |\Sigma_p| - 1\}$ and the s-characters are either disjoint from the set of integers or belong to the set $\{|\Sigma_p|, |\Sigma_p| + 1, \ldots, \sigma - 1\}$. An initial pre-processing of the text in $\mathcal{O}(|T| \log \sigma)$ time ensures that these conditions hold. The $\mathcal{O}(|T| \log \sigma)$ factor in the query complexity of Lemmas 5 and 6 and Theorem 1 is due to this pre-processing.

2.3 Long Patterns (Proof of Lemma 5)

We consider the patterns which are of length at least Δ, where $\Delta = \lceil \log_\sigma n \rceil$. For a string S and Δ, we use $\mathsf{tail}(S)$ to denote the largest suffix of S whose length is a multiple of Δ and $\mathsf{head}(S)$ is the remaining (possibly empty) prefix of S. We begin by obtaining

[1] Initialize a counter $C = 0$ and an integer array A such that $A[c] = -1$ for every $c \in \Sigma_p$. Traverse the string S from left to right. If $S[i] \in \Sigma_p$ (i.e., $S[i] < |\Sigma_p|$) check $A[S[i]]$; otherwise, $\mathsf{pEncode}(S)[i] = S[i]$. If $A[S[i]] = -1$ then assign $\mathsf{pEncode}(S)[i] = A[S[i]] = C$, increment C by one and proceed. Otherwise, assign $\mathsf{pEncode}(S)[i] = A[S[i]]$ and proceed. Note that s-characters remain unchanged.

$\mathsf{pEncode}(\mathsf{tail}(\mathsf{P}_i))$ for every $\mathsf{P}_i \in \mathcal{P}$, and maintain the encoded tails explicitly. Now, we encode $\mathsf{head}(\mathsf{P}_i)$ from right to left using the same encoding that was used for its tail. More specifically, form the string P'_i by concatenating $\mathsf{tail}(\mathsf{P}_i)$ with the reverse of $\mathsf{head}(\mathsf{P}_i)$. Then, the desired encoding of the jth character in the reversed head is given by $\mathsf{pEncode}(\mathsf{P}'_i)[|\mathsf{tail}(\mathsf{P}_i)| + j]$. The following observation is due to the definition of p-match and Observation 3.

▶ **Observation 7.** *Let S and S' be two strings having equal length. Then S and S' are a p-match iff both the conditions are satisfied: (i) the p-encoded tails of both S and S' are equal, and (ii) the p-encoded heads (as described above) of both S and S' are equal.*

The space needed for maintaining the encoded heads and tails of all patterns combined is $n \log \sigma$ bits.

2.3.1 Creating the Index

We create a tree $\mathcal{T}_{out}$ with d nodes where node v_i corresponds to the pattern P_i. A node v_j is the parent of a node v_i iff P_j is the longest pattern such that it is a p-match with a proper-suffix of P_i. In other words, v_j is the parent of a node v_i iff P_j is the longest pattern such that $|\mathsf{P}_j| < |\mathsf{P}_i|$ and $\mathsf{pEncode}(\mathsf{P}_j) = \mathsf{pEncode}(\mathsf{P}_i[|\mathsf{P}_i| - |\mathsf{P}_j| + 1, |\mathsf{P}_i|])$. This *output tree* will be useful for reporting occurrences of a pattern and is analogous to the report links in the AC-automaton [1]. Specifically, let k be a position in the text T such that P_i is the longest pattern which has an occurrence ending at k. Then all patterns whose occurrence ends at k can be found out by following the parent pointers starting at node v_i. Clearly, the start position of all such occurrences can be easily found. Space occupied by the tree is $\mathcal{O}(d \log n)$ bits.

Let Σ' be an alphabet such that each character in Σ' corresponds to a Δ-length string over the alphabet Σ. Thus, Σ' contains at most σ^{Δ} characters, and each character can be represented in $\Delta \log \sigma$ bits. Starting from left, we group every Δ characters of $\mathsf{pEncode}(\mathsf{tail}(\mathsf{P}_i))$, and replace it by the corresponding character from Σ'. In order to efficiently map this Δ-length string over Σ to its corresponding character in Σ', we maintain a perfect hash-table $\mathcal{H}$. Note that the number of Δ-length strings to be stored is at most $\lceil n/\Delta \rceil$. The space occupied by $\mathcal{H}$ is $\mathcal{O}(n/\Delta \times \Delta \log \sigma) = \mathcal{O}(n \log \sigma)$ bits. Create a trie $\mathcal{T}_{tail}$ for all the condensed encoded pattern tails of $\mathcal{P}$. Specifically, if the pattern length is not a multiple of Δ, then we ignore its head while creating the trie. Note that $\mathcal{T}_{tail}$ has at most $\lceil n/\Delta \rceil$ nodes. Each edge in $\mathcal{T}_{tail}$ corresponds to a Δ-length substring of some $\mathsf{pEncode}(\mathsf{P}_i)$. We maintain a pointer to the start location of this substring in $\mathsf{pEncode}(\mathsf{P}_i)$. (Given an edge, this allows us to find any jth character of the corresponding Δ-length substring of $\mathsf{pEncode}(\mathsf{P}_i)$ in $\mathcal{O}(1)$ time. The purpose of this will become clear when we discuss how to query the trie.) The space needed to store this information is $\mathcal{O}((n/\Delta) \log n) = \mathcal{O}(n \log \sigma)$ bits.

For any node u in $\mathcal{T}_{tail}$, we use $\mathsf{path}(u)$ to denote the string obtained by concatenating the edge labels (which are characters from Σ') one the path from root to the node u, and $\mathsf{path}_e(u)$ to denote the expanded string for $\mathsf{path}(u)$ i.e., the string obtained by mapping each character of $\mathsf{path}(u)$ to its corresponding Δ-length string over Σ. For each node u, we maintain the following information.

- a *goto link* as in the case of the AC-automaton for navigating the trie: given a node u and a character $c \in \Sigma'$, we can find its child v where the edge (u, v) is (conceptually) labeled by c, or report that no such child exists. (This is facilitated by the hash-table $\mathcal{H}$, whereby we read Δ characters from T, encode it, and use it to find the corresponding character from Σ'.)

- a *failure link* as in the case of the AC-automaton: Let S be the largest proper suffix of $\mathsf{path}(u)$ for which there exists a node v, such that $\mathsf{path}_e(v)$ is same as the string obtained by expanding S, re-encoding it according to Observation 4, and then compressing it back. Then, the failure link of u points to v.
- an *output link* from u to the node v_i in $\mathcal{T}_{out}$ such that P_i is the longest pattern satisfying $\mathsf{pEncode}(\mathsf{P}_i) = \mathsf{pEncode}(\mathsf{path}_e(u)[|\mathsf{path}_e(u)| - |\mathsf{P}_i| + 1, |\mathsf{path}_e(u)|])$, where the re-encoding is according to Observation 4.
- $\mathsf{alphaDepth}(u)$ i.e., the number of distinct integers less than $|\Sigma_p|$ in $\mathsf{path}_e(u)$. (Conceptually, this is the number of distinct p-characters.)

The space required to maintain goto links, failure links, output links, and alphabet depth over all nodes is $\mathcal{O}(\lceil n/\Delta \rceil(\Delta \log \sigma + \log n + \log \sigma)) = \mathcal{O}(n \log \sigma)$ bits.

Lastly, we maintain a succinct representation of $\mathcal{T}_{tail}$ using the techniques of Sadakane and Navarro [28]. Using this, in $\mathcal{O}(1)$ time, we can find (i) node-depth of a node, and (ii) $\mathsf{levelAncestor}(u, D)$ = the node (if any) on the path from root to u that has node-depth D. (The root has depth zero.) The space needed is $2\lceil n/\Delta \rceil + o(n/\Delta) = \mathcal{O}(n/\Delta)$ bits.

In summary, $\mathcal{T}_{tail}$ occupies $\mathcal{O}(n \log \sigma + n/\Delta + d \log n) = \mathcal{O}(n \log \sigma + d \log n)$ bits.

Now, we focus on the head of each pattern. Consider a pattern P_i. First, we reverse $\mathsf{head}(\mathsf{P}_i)$, then encode it (as described in the beginning of this section). Create two copies of the resultant head, each of which is obtained by appending two special s-characters $\$_i$ and $\#_i$, neither of which belongs to Σ. Locate the (distinct) node u such that $\mathsf{path}_e(u)$ is same as $\mathsf{pEncode}(\mathsf{tail}(\mathsf{P}_i))$. Note that u is defined and we call it the *locus* of P_i. Consider all patterns which have the same locus u. Create a compacted trie for the modified heads of all those patterns, and let u be the root of that trie. We call this the *head-trie* of u and is denoted by $\mathcal{T}_{head}(u)$. The parent of each leaf in $\mathcal{T}_{head}(u)$ corresponds to a pattern, say P_j, in the dictionary. We mark all such nodes in $\mathcal{T}_{head}(u)$ and label them with the corresponding pattern index j. Furthermore, for each node in $\mathcal{T}_{head}(u)$, we maintain a pointer to its nearest marked ancestor. The space occupied by each node for marking and labeling is $\mathcal{O}(\log n)$ bits. Each edge in $\mathcal{T}_{head}(u)$ is labeled by a substring (of length less than Δ) of the encoded head of some pattern P_j. We maintain a pointer to the start point of the corresponding substring of $\mathsf{pEncode}(\mathsf{P}_j)$, and also its length. This occupies $\mathcal{O}(\log n)$ bits for each edge. We also equip $\mathcal{T}_{head}(u)$ to allow constant time navigation operation from a node to the edge where the next character of an encoded head matches. This can be facilitated using perfect hashing based on the (unique) first character of the edge to its children, and occupies $\mathcal{O}(\log \sigma)$ bits for each transition (edge). Since there are d patterns, the number of nodes and edges in all such tries combined is $\mathcal{O}(d)$. Thus, the total space occupied for maintaining all head-tries is $\mathcal{O}(d \log n + d \log \sigma) = \mathcal{O}(d \log n)$ bits.

In summary, the total space occupied by the resultant trie (denoted as $\mathcal{T}_{long}$), all encoded patterns, and the hash-table $\mathcal{H}$ is $\mathcal{O}(n \log \sigma + d \log n)$ bits.

2.3.2 Finding Occurrences

Starting from position $j = 1$, we obtain $\mathsf{pEncode}(T[j, \Delta])$ and use its corresponding character from Σ' to traverse the trie $\mathcal{T}_{long}$ from the root. We repeat this process for the next Δ characters from T, and so on. More specifically, suppose we have reached a node u in $\mathcal{T}_{head}$ such that $\mathsf{path}_e(u) = \mathsf{pEncode}(T[j, j + |\mathsf{path}_e(u)| - 1])$. At this point, we have the following cases to consider.

- There is an output link associated with u, implying the existence of a pattern which is a p-match with a suffix of T ending at $j + |\mathsf{path}_e(u)| - 1$. All such patterns and starting locations can be found out in $\mathcal{O}(1)$ time per output by using the output link and $\mathcal{T}_{out}$.

- $\mathcal{T}_{head}(u)$ is non-empty implying that there is a pattern P_i such that the encoded tail of P_i is same as $\mathsf{path}_e(u)$. To report all possible occurrences of such patterns ending at $(j + |\mathsf{path}_e(v)| - 1)$, we use the encoded characters corresponding to $T[j-1], T[j-2], \ldots, T[j-\Delta+1]$ to traverse $\mathcal{T}_{head}(u)$ until no more traversal is possible. Suppose the last encountered node in this trie is v. We report all patterns with an occurrence ending at j by following the marked ancestor linkage from v.
- There is a child v of u such that the edge label of (u, v) is same as the character from Σ' corresponding to the last Δ characters of $\mathsf{pEncode}(T[j, j + |\mathsf{path}_e(v)| - 1])$. In this case, we traverse to v, and continue the process. Otherwise, follow the failure link of u.

Note that following the output link results in occurrence of at least one pattern. Each occurrence (i.e., the index and the corresponding pattern) can be reported in $\mathcal{O}(1)$ time. Moving forward we show how to deal with the head-trie, failure links, and goto links.

For our purposes, we maintain an array A of length $|\Sigma_p|$ such that for any $c \in \Sigma_p$, $A[c]$ equals the last position at which c appeared in T that has been read so far. (Initially each entry in the array A is empty.) We also maintain an array B of length $|\Sigma_p|$ such that for any $c \in \Sigma_p$, $B[c]$ gives us the desired encoding.

First, we show how to appropriately encode the incoming characters $T[j-1], T[j-2], \ldots, T[j-\Delta+1]$ when we traverse $\mathcal{T}_{head}(u)$. Initialize the array B to be empty. Note that it suffices to find the encoding for the first occurrence of every p-character starting from $j-1$ as the encoding for all future occurrences remains the same and can be obtained using B. Let c be a p-character. If $B[c]$ is not empty then use it to obtain the desired encoding. Otherwise, find the last occurrence of c using $A[c]$. We use the state of the array A at node u, and do not modify it while traversing the head-trie. We have the following two cases.

- **c appears in $T[j, j + |\mathsf{path}_e(u)| - 1]$:** Assume that the last occurrence is the λth character starting from j and (u', v') be the edge on which this occurrence lies i.e., $|\mathsf{path}(u')| < \lambda/\Delta \leq |\mathsf{path}(v')|$. We locate $v' = \mathsf{levelAncestor}(u, D)$ and $u' = \mathsf{levelAncestor}(u, D-1)$, where $D = |\mathsf{path}(v')| = \lceil \lambda/\Delta \rceil$ is the node-depth of v'. The encoding corresponding to c is exactly the $(\lambda - \Delta \cdot |\mathsf{path}(u')|)$th character of the label on this edge, and can be found using the pointer from the edge to the start of the corresponding substring of some encoded pattern tail. Set $B[c]$ to the encoded value. The time needed is $\mathcal{O}(1)$ per character.
- **c does not appear in $T[j, j + |\mathsf{path}_e(u)| - 1]$:** We maintain a counter C initialized to $\mathsf{alphaDepth}(u)$. Whenever we encounter such a c, the encoding is given by the value of C. Set $B[c]$ to the value of the counter. Following this, we increment C by one. The time needed is $\mathcal{O}(1)$ per character.

Thus, the time required to traverse each head-trie is $\mathcal{O}(\Delta)$ and each occurrence in the head-trie is reported in $\mathcal{O}(1)$ time by following the marked ancestor linkage.

Now, we concentrate on the failure link from u to v and show how to re-encode the text when we truncate characters from position j. Assume that k is the number of edges on the path from root to u (i.e., k is the node-depth of u) and that the failure link truncates $k'\Delta$ characters starting from j. Clearly, $1 \leq k' \leq k$. Therefore, we are now trying to find a match for the positions starting from $j' = j + k'\Delta$ and we need to re-encode the text T starting from j'. Since it is ensured that $\mathsf{pEncode}(T[j', j' + (k-k')\Delta - 1])$ is same as $\mathsf{path}_e(v)$, we are required to find the encoding of every p-character starting from $j'' = j' + (k-k')\Delta$.

Initialize the array B to be empty. Note that it suffices to find the encoding for the first occurrence of every p-character starting from j'', as the encoding for all future occurrences remains the same and can be obtained using the array B. Let c be a p-character. If $B[c]$ is

non-empty, then use it to obtain the desired encoding. Otherwise, find the last occurrence of c using $A[c]$. Note that we need the state of the array A at node u, which can be easily obtained by maintaining a copy of it whenever a new edge is traversed. (We delete the old copy when a new edge is traversed as it will not be required any more.) We have the following two cases.

- **c appears in $T[j', j'' - 1]$:** As described previously using $\mathsf{levelAncestor}(\cdot, \cdot)$ queries, we locate the position on the edge of the last occurrence. Then using the pointer from the edge we find the desired encoding and set $B[c]$. The time needed is $\mathcal{O}(1)$ per character.
- **c does not appear in $T[j', j''-1]$:** We maintain a counter C initialized to $\mathsf{alphaDepth}(v)$. Whenever we encounter such a c, the encoding is given by the value of C. Following this, we increment C by one. The time needed is $\mathcal{O}(1)$ per character.

The goto transition is achieved easily as follows. We read the next Δ characters from the text, encode them, and use the hash table $\mathcal{H}$ to traverse to the desired node. Since encoding each character can be performed in $\mathcal{O}(1)$ time (using the arrays A and B as described previously), each goto operation takes $\mathcal{O}(\Delta)$ time.

Now, we bound the query complexity. Initially, encoding the string T starting from $j = 1$ can be performed in $\mathcal{O}(|T|)$ time. Recall that on following a failure link, we truncate at least Δ characters starting from j. We read Δ characters on the failed edge (i.e., the one which was read unsuccessfully immediately before following the failure link). Thus, we can charge the characters on the failed edge to the first truncated Δ characters. This gives us an amortized complexity of $\mathcal{O}(1)$ per character. The number of failure link operations is at most $\lceil |T|/\Delta \rceil$. Thus, the number of nodes and edges traversed in the tail-trie is $\mathcal{O}(|T|/\Delta)$. For each edge, we read Δ characters and encoding the p-characters can be performed in $\mathcal{O}(1)$ time per character. For each node in the tail-trie, we will examine less than Δ characters in the head-trie; each of these characters can be appropriately encoded in $\mathcal{O}(1)$ time. Thus, the time required to traverse $\mathcal{T}_{long}$ (without reporting occurrences) is $\mathcal{O}((|T|/\Delta) \cdot \Delta) = \mathcal{O}(|T|)$. Each occurrence in the head-trie or the output tree is reported in $\mathcal{O}(1)$ time.

At the end of this process, for $j = 1$, we have reported occurrences of all patterns which end at a position of the form $j, j + \Delta, j + 2\Delta, \ldots$. The time required is $\mathcal{O}(|T| + occ_j)$. By repeating the process for $j = 2, 3, \ldots, \Delta$, all occ_ℓ occurrences of long patterns are reported in $\mathcal{O}(|T|\Delta + occ_\ell) = \mathcal{O}(|T| \log_\sigma n + occ_\ell)$ time.

Summarizing the discussions in this section, we obtain Lemma 5.

2.4 Short Patterns (Proof of Lemma 6)

Processing short patterns (having length less than Δ) is similar to that for head-tries. For all short patterns P_i, we create a compacted trie $\mathcal{T}_{short}$ for the strings $\mathsf{pEncode}(\mathsf{P}_i) \circ \$_i$ and $\mathsf{pEncode}(\mathsf{P}_i) \circ \#_i$, where $\circ$ denotes concatenation. The number of nodes in the trie is $\mathcal{O}(d)$. As in case of tail tries, we maintain a pointer from each edge to the start of the corresponding substring labeling the edge, and the length of the substring. We also equip each edge of $\mathcal{T}_{short}$ to support constant time navigation. Mark all nodes u if there is an encoded pattern which is the same as that obtained by concatenating the edge labels from root to u. The total space is bounded by $\mathcal{O}(d \log n)$ bits.

To find occurrences of short patterns, we use a rather brute force approach. Starting from $j = 1$, simply encode the next Δ characters of T, and use it to traverse the trie $\mathcal{T}_{short}$ until no more traversal is possible. Report j if at least one marked node is encountered in this traversal, and in that case, also report the patterns corresponding to these marked nodes. We repeat the process for $j = 2, 3, \ldots, |T|$. Since for each j at most 2Δ characters are

checked, the time required to report all occ_s occurrences of short patterns is $\mathcal{O}(|T|\Delta + occ_s) = \mathcal{O}(|T| \log_\sigma n + occ_s)$.

Summarizing the discussions in this section, we obtain Lemma 6.

3 Order-Preserving Dictionary Matching

As in the case of parameterized matching, we assume that the patterns are over an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, where the total-order on Σ is the natural order of integers. By initially pre-processing the patterns in $\mathcal{O}(n \log \sigma)$ time this condition is ensured. We use the following encoding scheme to convert a string S over Σ to a string $\mathsf{oEncode}(S)$. For every character $S[i]$, $\mathsf{oEncode}(S)[i]$ is the number of distinct characters in $S[1, i]$ having value at most $S[i]$. For example, consider the string $S = 5452316$, where each character is a single-digit integer. Then, $\mathsf{oEncode}(S) = 1121216$. The following is due to Kim et al. [20].

▶ **Observation 8.** *Two strings S and S' are an* o-match *iff* $\mathsf{oEncode}(S) = \mathsf{oEncode}(S')$. *A string S matches another string S' at a position i iff* $\mathsf{oEncode}(S) = \mathsf{oEncode}(S'[i, i + |S| - 1])$.

3.1 Creating the Index

As in case of p-patterns, we categorize o-patterns into long and small w.r.t the same parameter $\Delta = \lceil \log_\sigma n \rceil$. We also define the head and tail of the pattern similar to that in case of p-patterns. The tails are encoded using $\mathsf{oEncode}(\cdot)$ and are maintained explicitly. The encoding for the head of a pattern P_i is obtained as follows. Create a string P'_i first by reversing $\mathsf{head}(\mathsf{P}_i)$, then appending it at the end of $\mathsf{tail}(\mathsf{P}_i)$. Then, the encoding of the jth character in the reversed head is given by $\mathsf{oEncode}(\mathsf{P}'_i)[|\mathsf{tail}(\mathsf{P}_i)| + j]$. The following observation is due to the definition of o-match and Fact 8.

▶ **Observation 9.** *Let S and S' two be strings having equal length. Then S and S' are an o-match iff both the conditions are satisfied: (i) the o-encoded tails of both S and S' are equal and (ii) the o-encoded heads (as described above) of both S and S' are equal.*

For long patterns, the index is similar to that for p-patterns, except that we use the above encoding scheme. Also, we do not pre-process the resultant trie $\mathcal{T}_{tail}$ for answering $\mathsf{levelAncestor}(\cdot, \cdot)$-queries. For short patterns, the index is same except that for the encoding scheme. Thus, space is bounded by $\mathcal{O}(n \log \sigma + d \log n)$ bits.

3.2 Finding Occurrences

We assume that the characters in $|T|$ are from $\{1, 2, \ldots, \sigma\}$ with the total order being same as in that of the patterns. An initial pre-processing of the text in $\mathcal{O}(|T| \log \sigma)$ time ensures that this condition holds. Note that this does not affect the final query complexity of Theorem 2.

The querying process remains exactly similar to that for p-matching. Obviously, we use $\mathsf{oEncode}(\cdot)$ for encoding T, computing which requires a different technique. We maintain an array A of length σ such that $A[c]$ equals the position of last occurrence (if any) of $c \in \Sigma$ in the text read so far. Initially, for each $c \in \Sigma$, we assign $A[c] = -1$. Also, we maintain a balanced binary search tree (BST) $\mathcal{T}_{bin}$, which is initially empty. Suppose we are at position k in the text. If $A[T[k]] = -1$, we add $T[k]$ to $\mathcal{T}_{bin}$. We find the number of characters in $\mathcal{T}_{bin}$ that are at most $T[k]$, which gives us the desired encoding. Then, we update $A[T[k]] = k$ and proceed. Note that the size of $\mathcal{T}_{bin}$ is $\mathcal{O}(\sigma)$, which implies every deletion, insertion, and search operation requires $\mathcal{O}(\log \sigma)$ time.

The difficulty comes when we follow a failure link or when we traverse a head-trie. We first discuss the case of a failure link in which we may have to remove several characters from $\mathcal{T}_{bin}$. Suppose, after following a failure link, we are trying to find an occurrence for position j' and we are processing characters of the text starting from j''. (See Section 2.3.2 for detailed definitions of j' and j''.) Clearly, we have to remove those characters c from $\mathcal{T}_{bin}$ for which $A[c] < j'$. The total number of such deletions is at most $|T|$, each requiring $\mathcal{O}(\log \sigma)$ time yielding an amortized time complexity of $\mathcal{O}(\log \sigma)$ per character. To find these characters efficiently, we maintain the characters $c' \in \Sigma$ keyed by $A[c']$ in another BST $\mathcal{T}'_{bin}$. Note that the size of $\mathcal{T}'_{bin}$ is $\mathcal{O}(\sigma)$, which implies every insertion, update, and search operation requires $\mathcal{O}(\log \sigma)$ time. Using $\mathcal{T}'_{bin}$, we can find the desired characters to be removed in $\mathcal{O}(\log \sigma + output_k)$ time, where $output_k$ is the number of characters to be deleted from $\mathcal{T}_{bin}$ when we follow the kth failure link. Note that $\sum_k output_k \leq |T|$. Therefore, maintaining $\mathcal{T}'_{bin}$ and finding the desired characters to be removed on following a failure link have an amortized time complexity of $\mathcal{O}(\log \sigma)$ per character.

Traversing a head-trie is achieved similarly. Suppose, we are considering the string $T[j, j' - 1]$. Then, we have to encode characters $T[j-1], T[j-2], \ldots, T[j-\Delta+1]$ based on $T[j, j' - 1]$. With the aid of $\mathcal{T}'_{bin}$, we maintain another BST that contains only the characters in the interval $[j, j' - 1]$. The desired encoding of each character can be obtained in $\mathcal{O}(\log \sigma)$ amortized time.

Thus, the jth running of the algorithm requires $\mathcal{O}(|T| \log \sigma + occ_j)$ time for long patterns. Reporting all occ_ℓ occurrences of long patterns requires $\mathcal{O}(|T|\Delta \log \sigma + occ_\ell) = \mathcal{O}(|T| \log n + occ_\ell)$ time. For short patterns, since we follow the same brute-force strategy, it is easy to see that time required to report all occ_s occurrences is $\mathcal{O}(|T| \log n + occ_s)$.

This completes the proof of Theorem 2.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

2 Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994. doi:10.1016/0020-0190(94)90086-8.

3 Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 71–80, 1993. doi:10.1145/167088.167115.

4 Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, pages 88–100, 2010. doi:10.1007/978-3-642-13509-5_9.

5 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014. doi:10.1145/2635816.

6 Sudip Biswas, Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Forbidden extension queries. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*, pages 320–335, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.320.

7 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving incomplete suffix trees and order-preserving indexes. In *String Processing and Information Retrieval – 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, pages 84–95, 2013. doi:10.1007/978-3-319-02432-5_13.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

9 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

10 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly fm-index. In *String Processing and Information Retrieval, 11th International Conference, SPIRE 2004, Padova, Italy, October 5-8, 2004, Proceedings*, pages 150–160, 2004. doi:10.1007/978-3-540-30213-1_23.

11 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007. doi:10.1145/1240233.1240243.

12 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Succinct non-overlapping indexing. In *Combinatorial Pattern Matching – 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 – July 1, 2015, Proceedings*, pages 185–195, 2015. doi:10.1007/978-3-319-19929-0_16.

13 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, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

14 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406, 2000. doi:10.1145/335305.335351.

15 Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. In *Algorithms – ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*, pages 414–425, 2004. doi:10.1007/978-3-540-30140-0_38.

16 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Compressed index for dictionary matching. In *2008 Data Compression Conference (DCC 2008), 25-27 March 2008, Snowbird, UT, USA*, pages 23–32, 2008. doi:10.1109/DCC.2008.62.

17 Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *J. ACM*, 61(2):9:1–9:36, 2014. doi:10.1145/2590774.

18 Ramana M. Idury and Alejandro A. Schäffer. Multiple matching of parameterized patterns. In *Combinatorial Pattern Matching, 5th Annual Symposium, CPM 94, Asilomar, California, USA, June 5-8, 1994, Proceedings*, pages 226–239, 1994. doi:10.1007/3-540-58094-8_20.

19 Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 – March 2, 2013, Kiel, Germany*, pages 400–411, 2013. doi:10.4230/LIPIcs.STACS.2013.400.

20 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014. doi:10.1016/j.tcs.2013.10.006.

21 S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 631–637, 1995. doi:10.1109/SFCS.1995.492664.

22 Veli Mäkinen and Gonzalo Navarro. Compressed compact suffix arrays. In *Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings*, pages 420–433, 2004. doi:10.1007/978-3-540-27801-6_32.

23 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*, pages 45–56, 2005. doi:10.1007/11496656_5.

24 J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top- k term-proximity in succinct space. In *Algorithms and Computation – 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, pages 169–180, 2014. doi:10.1007/978-3-319-13075-0_14.

25 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007. doi:10.1145/1216370.1216372.

26 Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Algorithms and Computation, 11th International Conference, ISAAC 2000, Taipei, Taiwan, December 18-20, 2000, Proceedings*, pages 410–421, 2000. doi:10.1007/3-540-40996-3_35.

27 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003. doi:10.1016/S0196-6774(03)00087-7.

28 Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 134–149, 2010. doi:10.1137/1.9781611973075.13.

29 Alan Tam, Edward Wu, Tak Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009, Proceedings*, pages 39–50, 2009. doi:10.1007/978-3-642-03784-9_5.

30 Dekel Tsur. Top-k document retrieval in optimal space. *Inf. Process. Lett.*, 113(12):440–443, 2013. doi:10.1016/j.ipl.2013.03.012.

Encoding Two-Dimensional Range Top-k Queries

Seungbum Jo[1], Rahul Lingala[2], and Srinivasa Rao Satti[3]

1 **Seoul National University, Korea**
`sbcho@tcs.snu.ac.kr`
2 **IIT Bombay, India**
`lingalarahul7@gmail.com`
3 **Seoul National University, Korea**
`ssrao@cse.snu.ac.kr`

Abstract

We consider various encodings that support range Top-k queries on a two-dimensional array containing elements from a total order. For an $m \times n$ array, with $m \leq n$, we first propose an almost optimal encoding for answering one-sided Top-k queries, whose query range is restricted to $[1 \dots m][1 \dots a]$, for $1 \leq a \leq n$. Next, we propose an encoding for the general Top-k queries that takes $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$ bits. This generalizes the one-dimensional Top-k encoding of Gawrychowski and Nicholson [ICALP, 2015]. Finally, for a $2 \times n$ array, we obtain a $2 \lg \binom{3n}{n} + 3n + o(n)$-bit encoding for answering Top-2 queries.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Encoding model, top-k query, range minimum query

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.3

1 Introduction

Given a one-dimensional (1D) array $A[1 \dots n]$ from a total order and $1 \leq k \leq n$, the *Range Top-k query on A* (Top-$k(i, j, A), 1 \leq i, j \leq n$) returns the positions of k largest values in $A[i \dots j]$. We can extend this query to the two-dimensional (2D) array case. Given a 2D array $A[1 \dots m][1 \dots n]$, from a total order and $1 \leq k \leq mn$, the *Top-k query on A* (Top-$k(i, j, a, b, A), 1 \leq i, j \leq m, 1 \leq a, b \leq n$) returns the positions of k largest values in $A[i \dots j][a \dots b]$. Without loss of generality, we assume that all elements in A are distinct by ordering equal elements in the lexicographic order of their positions, and also assume that $m \leq n$. If the k positions of a Top-k query are reported in sorted order of the corresponding values, we refer to the query as *sorted* Top-k query; and refer to it as *unsorted* Top-k query, otherwise. For $1 \leq i, j \leq m$ and $1 \leq a, b \leq n$, we can also classify Top-k queries on 2D array by its range as follows.

1-sided query: The query range is $[1 \dots m][1 \dots b]$.
4-sided query: The query range is $[i \dots j][a \dots b]$.

We can also consider 2-sided and 3-sided queries which correspond to the ranges $[1 \dots j][1 \dots a]$ and $[1 \dots j][a \dots b]$ respectively. We consider how to support the Top-k queries in the *encoding model* in which we do not have access to the original input array A at query time. The minimum size of an encoding is also referred to as the *effective entropy* of the input data (with respect to the queries) [7].

In the rest of the paper, we assume that for Top-k encodings, k is at most the size of the array (either 1D or 2D). Also, unless otherwise mentioned, we assume that all Top-k queries are sorted Top-k queries.

Table 1 The summary of our results for Top-k queries on $m \times n$ 2D array. The value T is given by the formula $T = \sum_{i=0}^{\min(m,k)} i!\binom{m}{i}\binom{k}{i}$.

Array size	Query range	Space	Query time
$m \times n$	one-sided	$n\lceil \lg T \rceil$ bits	-
$2 \times n$	four-sided, $k \leq 2$	$2\lg\binom{3n}{n} + 3n + o(n)$ bits	-
$m \times n$	four-sided	$O(mn \lg n)$ bits	$O(k)$
$m \times n$	four-sided	$m^2 \lg\binom{(k+1)n}{n} + m \lg m + o(n)$ bits	-

1.1 Previous Work

Encoding Top-k queries on 1D array has been widely studied in the recent years. For a 1D array $A[1 \ldots n]$, Chan and Wilkinson [4] proposed a data structure that uses $\Theta(n)$ words and answers selection queries (i.e., selecting the k-th largest element) in $O(\lg k / \lg \lg n)$ time[1]. Grossi et al. [8] considered the Top-k encoding problem, and obtained an $O(n \lg \kappa)$-bit encoding which can answer the Top-k queries for any $k \leq \kappa$ in $O(\kappa)$ time or alternately, using $O(n \lg^2 \kappa)$ bits with $O(k)$ query time. (They also considered one-sided Top-k query, they proposed $n \lg k + O(n)$-bit encoding with $O(k)$ query time.) The space usage of this encoding was improved to $O(n \lg \kappa)$ bits, maintaining the $O(k)$ query time, by Navarro et al. [10]. Recently, Gawrychowski and Nicholson [6] proposed an $(k+1)nH(1/(k+1)) + o(n)$-bit[2] encoding for Top-k queries and showed that at least $(k+1)nH(1/(k+1))(1 - o(1))$ bits are required to encode Top-k queries.

To the best of our knowledge, there are no results on range Top-k queries for 2D array with general k. For $k = 1$, the Top-k query is same as the *Range Maximum Query (RMQ)*, which has been well-studied for 1D as well as for 2D arrays. For a 2D $m \times n$ array, Brodal et al. [1] proposed an $O(nm \min(m, \lg n))$-bit encoding which answers RMQ queries in $O(1)$ time. Brodal et al. [2] improved the space bound to the optimal $O(nm \lg m)$ bits, although this encoding does not support the queries efficiently.

1.2 Our Results

For an $m \times n$ 2D array A, we first obtain an $n\lceil \lg T \rceil$-bit encoding for answering one-sided Top-k queries, where $T = \sum_{i=0}^{\min(m,k)} i!\binom{m}{i}\binom{k}{i}$. We then show that any encoding that supports Top-k queries on A must use at least $n \lg T$ bits.

Next, we observe that one can obtain an $O(mn \lg n)$-bit data structure which answers 4-sided Top-k queries on A in $O(k)$ time, by combining the results of [3] and [1]. We then propose an $m^2 \lg\binom{(k+1)n}{n} + m \lg m + o(n)$-bit encoding for 4-sided Top-k queries on A, by extending the Top-k encoding of Gawrychowski and Nicholson for 1D arrays [6].

When $k = 2$ and $m = 2$, the above encoding takes $4\lg\binom{3n}{n} + o(n) \approx 11.02n$ bits. For this case, we propose an alternative encoding which uses $2\lg\binom{3n}{n} + 3n + o(n) \approx 8.51n$ bits (and can answer the 4-sided Top-2 queries on A). All these results are summarized in Table 1.

We assume the standard word-RAM model [9] with word size $\Theta(\lg n)$.

2 Encoding one-sided range Top-k queries on two dimensional array

In this section, we consider the encoding of one-sided Top-k queries on a 2D array $A[1 \ldots m][1 \ldots n]$. We first introduce the encoding by simply extending the encoding of one-sided Top-k

[1] We use $\lg n$ to denote $\log_2 n$

[2] $H(x) = x \lg(1/x) + (1-x)\lg(1/(1-x))$

queries for 1D array proposed by Grossi et al. [8]. Next we propose an optimal encoding for one-sided Top-k queries on A.

For a 1D array $A'[1 \dots n]$, one can define another 1D array $X[1 \dots n]$ such that $X[i] = i$ for $1 \leq i \leq k$ and for $k < i \leq n$, $X[i] = X[i']$ if there exist a position $i' < i$ such that $A'[i]$ is larger than $A'[i']$ which is the k-th largest value in $A'[1 \dots i-1]$, and $X[i] = k+1$ otherwise. One can answer the Top-$k(1, i, A')$ by finding the rightmost occurrence of every element $1 \dots k$ in $X[1 \dots i]$. By representing X (along with some additional auxiliary structures) using $n \lg k + O(n)$ bits, Grossi et al. [8] obtained an encoding which supports 1-sided Top-k queries on A' in $O(k)$ time.

For a 2D array A, one can encode A to support one-sided Top-k queries by writing down the values of A in column-major order into a 1D array, and using the encoding described above – resulting in the following encoding.

▶ **Proposition 1.** *A 2D array $A[1 \dots m][1 \dots n]$ can be encoded using $mn \lg k + O(n)$ bits to support one-sided Top-k queries in $O(k)$ time.*

Now we describe an optimal encoding of A which supports one-sided Top-k queries. For 1D array $A'[1 \dots n]$, we can define another 1D array $B'[1 \dots n]$ such that for $1 \leq i \leq n$, $B'[i] = l$ if $A'[i]$ is the l-th largest element in $A'[1 \dots i]$ with $l \leq k$, and $B'[i] = k+1$ otherwise. Then we answer the Top-$k(1, i, A')$ query as follows. We first find the rightmost position $p_1 \leq i$ such that $B'[p_1] \leq k$. Then we find the positions $p_2 > p_3 \cdots > p_k$ such that for $2 \leq j \leq k$, p_j is the rightmost position in $A'[1 \dots p_{j-1} - 1]$ with $B'[p_j] \leq k - j + 1$. Finally, we return the positions $p_1, p_2, \dots, p_k$. Therefore by storing B' using $n\lceil \lg (k+1) \rceil$ bits, we can answer the one-sided Top-k queries on A'. Also we can sort $A'[p_1], \dots, A'[p_k]$ using the property that for $1 \leq b < a \leq k$, $A'[p_a] < A'[p_b]$ if and only if one of the following two conditions hold: (i) $B'[p_a] \geq B'[p_b]$, or (ii) $B'[p_a] < B'[p_b]$ and there exist $q = B'[p_b] - B'[p_a]$ positions $j_1, j_2, \dots, j_q$ such that $p_a < j_1 < \cdots < j_q < p_b$ and $B'[j_r] \leq B'[p_a]$ for $1 \leq r \leq q$.

We can extend this encoding for the one-sided Top-k queries on a 2D array A. For $1 \leq j \leq n$, we first define the elements of j-th column in A as $a_{1j} \dots a_{mj}$. Then we define the sequence $S_j = s_{1j} \dots s_{mj}$ such that for $1 \leq i \leq m$, $s_{ij} = l$ if a_{ij} is the l-th largest element in $A[1 \dots m][1 \dots j]$ with $l \leq k$ and $s_{ij} = k+1$ otherwise. Since there exist $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}\binom{k}{i} i!$ possible S_i sequences (T is the total number of ways in which we can choose i out of the m rows for new entries into the Top-k positions, summed over all possible values of i), we can store $S^A = S_1 \dots S_n$ using $n\lceil \lg T \rceil$ bits and we can answer the one-sided Top-$k(1, m, 1, j)$ queries on A by the following procedure.

1. Find the rightmost column q, for some $q \leq j$, such that S_q has $\ell > 0$ elements $s_{p_1 q}, \dots, s_{p_\ell q}$ where $s_{p_1 q} < \cdots < s_{p_\ell q} < k+1$. If $\ell = k$, we return the positions of $A[p_1][q] \dots A[p_k][q]$ as the answers of the query, and stop. Otherwise (if $\ell < k$), we return the positions of $A[p_1][q] \dots A[p_\ell][q]$, and
2. Repeat Step 1 by setting k to $k - \ell$, and j to $q - 1$.

We can return the positions in the sorted order of their corresponding values similar to the 1D array case as described above. This encoding takes less space than the encoding in the Proposition 1 since $mn \lg k = n \lg (1 + (k-1))^m = n \lg \sum_{i=0}^{m} \binom{m}{i} (k-1)^i \geq n \lg T$. The following theorem shows that the space usage of this encoding is essentially optimal for answering one-sided Top-k queries on A.

▶ **Theorem 2.** *Any encoding of a 2D array $A[1 \dots m][1 \dots n]$ that supports one-sided Top-k queries requires $n \lg T$ bits, where $T = \sum_{i=0}^{\min(m,k)} i! \binom{m}{i}\binom{k}{i}$.*

Proof. Suppose there are two distinct sequences $S^A = S_1 \dots S_i$ and $S^{A'} = S'_1 \dots S'_i$ which give one-sided Top-k encodings of 2D arrays A and A', respectively. For $1 \leq b \leq n$, if $S_b \neq S'_b$

then Top-$k(1, m, 1, b, A) \neq$ Top-$k(1, m, 1, b, A')$ by the definition of S^A and $S^{A'}$. Since for an $m \times n$ array, there are T^n distinct sequences $S^{A_1} \dots S^{A_{T^n}}$, it is enough to prove that for $1 \leq q \leq T^n$, each $S^{A_q} = S_1^q \dots S_n^q$ has an array A such that $S^A = S^{A_q}$.

Without loss of generality, suppose that all elements in A come from the set $L = \{1, \dots, mn\}$. Then we can reconstruct A from the rightmost column using S^{A_q} as follows. If $s_{jn}^q \leq k$, for $1 \leq j \leq m$, we assign the s_{jn}^q-th largest element in L to $A[j][n]$. After we assign all values in the rightmost column with $s_{jn}^q \leq k$, we discard all assigned values from L, move to $(n-1)$-th column and repeat the procedure. After we assign all values in A whose corresponding values in S^{A_q} are smaller than $k+1$, we assign the remaining values in L to remaining positions in A_q which are not assigned yet. Thus for any $1 \leq b \leq n$, if S_b^q has $\ell > 0$ elements $s_{p_1 b}, \dots, s_{p_\ell b}$ where $s_{p_1 b} < \dots < s_{p_\ell b} < k+1$, then the b-th column in A contains ℓ-largest elements in $A[1 \dots m][1 \dots b]$ by the above procedure. This shows that $S^A = S^{A_q}$. ◀

3 Encoding range Top-k queries on two dimensional array

In this section, we give an encoding which supports general Top-k queries on 2D array. For an $m \times n$ 2D array, we first introduce an $O(mn \lg n)$-bit encoding which supports Top-k query in $O(k)$ time by using the RMQ encoding of Brodal et al. [2].

▶ **Proposition 3.** *A 2D array $A[1 \dots m][1 \dots n]$ can be encoded using $O(mn \lg n)$ bits to support unsorted Top-$k(i, j, a, b, A)$ in $O(k)$ time for $1 \leq a, b \leq m$ and $1 \leq i, j \leq n$.*

Proof. We use a data structure similar to the one outlined in [3] (based on Frederikson's heap selection algorithm [5]) for answering unsorted Top-k queries in 1D array[3]. First encode A using $O(mn \lg n)$ bits to support RMQ (range maximum) queries in constant time for any rectangular range in A. This encoding also supports finding the rank (i.e., the position in sorted order) of any element in A in $O(1)$ time [1]. Next, let $x = A[x_1][x_2]$ be the maximum value in $A[i \dots j][a \dots b]$, which can be found using an RMQ query on A. Then consider the 4-ary heap obtained by the following procedure. The root of the heap is x, and its four subtrees are formed by recursively constructing the 4-ary heap on the sub-arrays $A[i \dots x_1 - 1][a \dots b]$, $A[x_1 + 1 \dots j][a \dots b]$, $A[x_1][a \dots x_2 - 1]$ and $A[x_1][x_2 + 1 \dots b]$, respectively. Now, we can find the k largest elements in the above 4-ary heap in $O(k)$ time using the algorithm proposed by Frederickson [5] (note that this algorithm only builds a heap with $O(k)$ nodes which is a connected subgraph of the above 4-ary heap). ◀

We now introduce another encoding to support Top-k queries on an $m \times n$ 2D array A. This encoding extends the optimal Top-k encoding of Gawrychowski and Nicholson [6] for a 1D array. This encoding does not support the queries efficiently. Compared to the encoding of Proposition 3, this encoding uses less space when $n = \Omega(k^m)$. We first review the Gawrychowski and Nicholson [6]'s optimal Top-k encoding for 1D array, and show how to extend this encoding to the 2D array case.

For a given 1D array $A'[1 \dots n]$, we define the sequence of arrays $S^{A'} = S_1^{A'} \dots S_n^{A'}$, where for $1 \leq j \leq n$ and $1 \leq i \leq j$, $S_j^{A'}$ is an array of size j defined as follows.

$$S_j^{A'}[i] = \begin{cases} p & \text{if there are } p\ (< k) \text{ elements larger than } A'[i] \text{ in } A'[i+1 \dots j] \\ k & \text{otherwise} \end{cases}$$

[3] Brodal et al. [3] also give another structure to answer sorted Top-k queries, with the same time and space bounds.

A_1	3	7	8	2	6	4
A_2	6	4	10	3	5	2

$S_1^{A_1}$	0					
$S_2^{A_1}$	1	0				
$S_3^{A_1}$	2	1	0			
$S_4^{A_1}$	2	1	0	0		
$S_5^{A_1}$	2	2	0	1	0	
$S_6^{A_1}$	2	2	0	2	0	0

$S_1^{A_2}$	0					
$S_2^{A_2}$	0	0				
$S_3^{A_2}$	1	1	0			
$S_4^{A_2}$	1	1	0	0		
$S_5^{A_2}$	1	2	0	1	0	
$S_6^{A_2}$	1	2	0	1	0	0

$I_1^{(1,2)}$	1					
$I_2^{(1,2)}$	2	0				
$I_3^{(1,2)}$	2	1	1			
$I_4^{(1,2)}$	2	1	1	1		
$I_5^{(1,2)}$	2	1	1	2	0	
$I_6^{(1,2)}$	2	1	1	2	0	0

$I_1^{(2,1)}$	0					
$I_2^{(2,1)}$	1	0				
$I_3^{(2,1)}$	2	1	0			
$I_4^{(2,1)}$	2	1	0	0		
$I_5^{(2,1)}$	2	2	0	1	0	
$I_6^{(2,1)}$	2	2	0	2	0	0

Figure 1 Top-k encoding of the 2D array A when $k = 2$.

See Figure 1 for an example.

If $S_j^{A'}[i] < k$, we call $A[i]$ in $A[1 \dots j]$ as *active*, otherwise $A[i]$ is *inactive* in $A[1 \dots j]$.

Gawrychowski and Nicholson [6] show that for $1 \leq i, j \leq n$, Top-$k(i, j, A')$ can be answered using $S_j^{A'}[i \dots j]$. They obtained a $\lg \binom{(k+1)n}{n} + o(n)$-bit encoding of $S^{A'}$ by representing $\delta_1^{A'} \dots \delta_{n-1}^{A'}$ (where $\delta_i^{A'} = \sum_{l=1}^{i+1} S_{i+1}^{A'}[l] - \sum_{l=1}^{i} S_i^{A'}[l]$) in unary, and compressing the sequence using the following lemma.

▶ **Lemma 4** ([11]). *Let S be a string of length n over the alphabet $\Sigma = \{1, 0\}$ containing m 1s. One can encode S using $\lg \binom{n}{m} + o(n)$ bits to access any position in S in constant time.*

Since $\sum_{i-1}^{n-1} \delta_i^{A'} \leq kn$, the unary sequence has kn zeros and n ones. The following lemma states their result for 1D arrays.

▶ **Lemma 5** ([6]). *Given a 1D array $A[1 \dots n]$, there is an encoding of A using $\lg \binom{(k+1)n}{n} + o(n)$ bits which supports Top-k queries.*

We now describe how to extend this encoding to a 2D $m \times n$ array A. For $1 \leq i \leq m$, let $A_i[1 \dots n]$ be the array of the i-th row in A. We construct Top-k encodings for the rows $A_1 \dots A_m$ using Lemma 5, and this takes $m \lg \binom{(k+1)n}{n} + o(n)$ bits. In addition, for every $1 \leq i \neq j \leq m$, we define the sequence of arrays, $I^{(i,j)} = I_1^{(i,j)} \dots I_n^{(i,j)}$ to represent S^i with respect to the elements in A_j. For $1 \leq r \leq n$, $I_r^{(i,j)}$ is an array of size r defined as follows.

$$I_r^{(i,j)}[s] = \begin{cases} p & \text{if } i > j \text{ and there are } p\ (< k) \text{ elements which are larger than } A_i[s] \text{ in } A_j[s+1 \dots r] \\ q & \text{if } i < j \text{ and there are } q\ (< k) \text{ elements which are larger than } A_i[s] \text{ in } A_j[s \dots r] \\ k & \text{otherwise (if there are } \geq k \text{ elements, in the above two cases)} \end{cases}$$

See Figure 1 for an example.

We can answer the Top-$k(i, j, a, b, A)$ queries as follows. We first define the 1D array $B[1 \dots b(j-i+1)]$ by writing down the values of $A[i \dots j][1 \dots b]$ in column-major order. Then we observe that Top-$k(i, j, a, b, A)$ can be answered using $S^B_{b(j-i+1)}[a(j-i+1)+1 \dots b(j-i+1)]$.

The following lemma shows that we can compute the values in $S^B_{b(j-i+1)}$ using $S^{A_1} \dots S^{A_m}$ and all the arrays $I_b^{(c,d)}$, for $1 \le c \ne d \le m$.

▶ **Lemma 6.** *Given a 2D array $A[1 \dots m][1 \dots n]$, for $1 \le i \le j \le m$ and $1 \le b \le n$, let $B[1 \dots q]$ be the 1D array of size $q = (j-i+1)b$ obtained by writing the elements of $A[i \dots j][1 \dots b]$ in column-major order. Also, for any $1 \le s \le q$, let (s_{row}, s_{col}) be the position corresponding $B[s]$ in A (which can be computed using $s_{col} = \lceil s/(j-i+1) \rceil$ and $s_{row} = s - (s_{col}-1) \cdot (j-i+1) + (i-1)$). Then*

$$S^B_q[s] = \min\left(k, \left(S_b^{A_{s_{row}}}[s_{col}] + \sum_{i \le \ell \le j, \ell \ne s_{row}} I_b^{(s_{row},\ell)}[s_{col}]\right)\right).$$

Proof. It is enough to count the number of elements in B (i.e., in $A[i \dots j][a \dots b]$) which are larger than $B[s]$ (i.e., $A[s_{row}][s_{col}]$) in $B[s+1 \dots q]$ (i.e., the corresponding elements in A). Let L be the set of these elements. If $|L| \ge k$, then $S^B_q[s] = k$. In the following, we describe how to compute $S^B_q[s]$ when $|L| < k$.

From the definition of $S_b^{A_{s_{row}}}$, it follows that the number of elements in L which are in row s_{row} is $S_b^{A_{s_{row}}}[s_{col}]$.. Also, for any row $\ell \ne s_{row}$, $I_b^{(s_{row},\ell)}[s_{col}]$ is the number of elements in L that belong to row ℓ. From all these values, we can compute $|L|$. ◀

By Lemma 6, we can answer the Top-k queries on A using the Top-k encodings of all the rows $A_1, \dots, A_m$, together with all the arrays $I^{(i,j)}$, for all $1 \le i \ne j \le m$. Since we can recover the order of all active elements in the prefix of i-th row using S^{A_i} [6], we can decode $I_p^{(i,j)}$ using $I_{p-1}^{(i,j)}$ and $\gamma_p^{ij} = \sum_{l=1}^{p} I_p^{(i,j)}[l] - \sum_{l=1}^{p-1} I_{p-1}^{(i,j)}[l]$ by the following procedure, for $p > 1$.

1. Append 0 to $I_{p-1}^{(i,j)}$. Let this array be $J_{p-1}^{(i,j)}$.
2. Find the positions of $\gamma_{p-1}^{(i,j)}$ smallest active values in $A_i[1 \dots p]$ using S^{A_i}, and increase the values of $J_{p-1}^{(i,j)}$ in these positions by 1.

Therefore, using $I_1^{(i,j)}$, and $\gamma_2^{(i,j)}, \dots, \gamma_n^{(i,j)}$, we can encode $I^{(i,j)}$. Since the sum $\sum_{\ell=2}^{\ell=n} \gamma_\ell^{(i,j)}$ is at most kn, we can encode all the arrays $I^{(i,j)}$ (for all possible $i \ne j$) using $m(m-1)\lg\binom{(k+1)n}{n} + o(n)$ bits (by converting $\gamma_\ell^{(i,j)}$'s into unary, as in the encoding of Lemma 5). Also, to encode $I_1^{(i,j)}$ for $i < j$ (note that if $i > j$, $I_1^{(i,j)}$ is always 0), we need to store the ordering of all elements in the first column, which takes $m \lg m$ bits. This gives a proof of the following theorem.

▶ **Theorem 7.** *Given a 2D array $A[1 \dots m][1 \dots n]$, there is an encoding of A using $m^2 \lg\binom{(k+1)n}{n} + m \lg m + o(n)$ bits which can answer the Top-k queries.*

4 Encoding range Top-2 queries on $2 \times n$ array

In this section, we consider a special case of Top-k encodings for 2D arrays when the array has only two rows, and $k = 2$. Note that for these parameter values, Theorem 7 gives an encoding of size $4\lg\binom{3n}{n} + o(n) \approx 11.02n$ bits. We describe an alternative approach which results in an encoding of size $2\lg\binom{3n}{n} + 3n + o(n) \approx 8.51n$ bits.

For $i \in \{1, 2\}$, let A_i be the array $[a_{i1}, \dots, a_{in}]$ of size n constituting the i-th row of A. We maintain Top-k encodings for A_1 and A_2, which enable us to support the Top-k queries on the individual rows. To support queries that span both the rows, we store an auxiliary structure of size at most $3n$ bits.

We construct a weighted DAG, D_A, such that each node in D_A is labeled with a *range* $[a, b]$, where $1 \le a \le b \le n$, and has a *weight* $w([a, b]) \in \{1, 2\}$. In the rest of this section, we

A_1	1	21	17	12	20	3	15	11	10
A_2	6	5	16	14	19	2	18	4	7

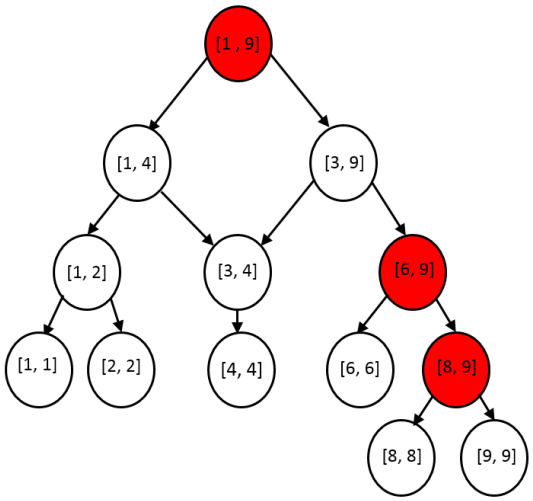


Figure 2 $2 \times n$ array A and the DAG D_A. Nodes with weight 2 are colored red, while those with weight 1 are not colored.

use the notation Top-2($[a, b]$) to refer to the query Top-2$(1, 2, a, b, A)$. We also use (i, a) to denote the position in the i-th row and a-th column in A. Now we define D_A as follows.

1. The root of D_A is labeled with the range $[1, n]$, and $w([1, n]) = 2$.
2. If $a = b$, then $[a, b]$ is a leaf node in D_A, with weight $w([a, b]) = 1$.
3. Suppose there exists a non-leaf node $[a, b]$ in D_A, such that the answers to the query Top-2($[a, b]$) are (i, a') and (j, b'), for some $1 \leq i, j \leq 2$ and $a \leq a' \leq b' \leq b$. Then the at most two children of the node $[a, b]$ are $[a, b' - 1]$and $[a' + 1, b]$.
   - **Case 1.** If $a' = b'$ and $a < b' - 1$, $w([a, b' - 1]) = 2$.
   - **Case 2.** If $a' = b'$ and $a' + 1 < b$, $w([a' + 1, b]) = 2$.
   - **Case 3.** In all other cases, $w([a, b' - 1]) = w([a' + 1, b]) = 1$.

See Figure 2 for an example. Note that a node can have at most two parents since each end point of the interval corresponding to a node can be shared by exactly one of its children. If the two parents of a node belong to two different cases, then the weight of the child node is set to be the smaller of the weights set in the two cases. For example, in Figure 2, the node $[3, 4]$ belongs to Case 3 through the parent node $[1, 4]$, and belongs to Case 1 through the parent $[3, 9]$. Hence, its weight is set to 1. Also, not all intervals of the form $[a, a]$ need to appear as leaves in D_A (eg., $[3, 3]$ in Figure 2).

From the construction of D_A, one can observe that if there is a node $[a, b]$ in D_A, with $1 < a \leq b < n$, then the columns $a - 1$ and $b + 1$ both contain at least one element that is larger than the second largest elements in the sub-array $A[1 \ldots 2][a \ldots b]$. From this observation, it follows that given any two distinct nodes x and y in D_A, the answers to the queries Top-2(x) and Top-2(y) are distinct (if there are two distinct nodes $[a, b]$ and $[a', b']$ with $b < b'$ such that Top-2($[a, b]$) = Top-2($[a', b']$), then Top-2($[a, b + 1]$) = Top-2($[a', b']$), contradicting the fact that Top-2($[a, b]$) $\neq$ Top-2($[a, b + 1]$). The case when $b > b'$, $a > a'$ or $a < a'$ is analogous). In addition, we use the following property of D_A in proving lemmas in this section.

▶ **Proposition 8.** *Let A be a $2 \times n$ array and D_A be its corresponding weighted DAG. For any distinct two nodes p and q in D_A, $p \subset q$ if and only if p is descendant of q.*

Proof. From the construction of D_A, it is the case that if p is a descendant of q, then $p \subset q$. Now, suppose $p \subset q$ and $p = [a_p, b_p]$ is not descendant of q. Then there exists a node q' which is a descendant of q such that $p \subset q'$, but no child of q' contains p. Since neither of the children of q' contain p, both column positions of Top-2(q') must belong to p (otherwise, at least one of the children of q' would contain p). But this would imply that Top-2(q') = Top-2(p), which leads to a contradiction since every node in D_A has distinct Top-2 answers. ◀

Furthermore, the following lemma shows that D_A contains all distinct answers for Top-2$([a, b])$, for $1 \leq a \leq b \leq n$ (in other words, the answers to any Top-2$([a, b])$ query on A are same as the answers to the Top-2 query on some node in the DAG).

▶ **Lemma 9.** *Let A be a $2 \times n$ array. For $1 \leq a \leq b \leq n$, for any interval $[a, b]$, there exist a node p in D_A such that Top-2$([a, b])$ = Top-2(p).*

Proof. We first show that there exists a unique p such that p contains the interval $[a, b]$ and none of the children of p (fully) contain $[a, b]$. We then show that the Top-2$([a, b])$ = Top-2(p).

Since the root in D_A contains all columns in A, it is easy to see that there exists at least one node $p = [a_p, b_p]$ in D_A such that $[a, b] \subset p$ but no child of p contains $[a, b]$. Suppose that there exists another node $p' = [a'_p, b'_p]$ such that $[a, b] \subset p'$ but no child of p' contains $[a, b]$. From Proposition 8, it follows that $p \not\subset p'$ and $p' \not\subset p$ (otherwise, one of them would be a descendant of the other, contradicting the conditions on p and p'). Now, suppose that $a_p < a'_p < b_p < b'_p$ (the case when $a'_p < a_p < b'_p < b_p$ is analogous). Then there exists a column $c < a'_p$ such that p has a child node $[c, b_p]$ which contains $[a, b]$ by the property of D_A (note that $a'_p \leq a \leq b \leq b_p$), contradicting the fact that p does not have such a child. This shows that there is a unique such p in D_A.

Now we claim that Top-2$([a, b])$ = Top-2(p). Suppose that there exist a $c \notin [a, b]$ in p such that column c contains at least one of the answers to Top-2(p). Also without loss of generality, we assume that $c < a$ (the case when $c > b$ can be handled in a similar way). Then by the property of D_A, p has a child $[c + 1, b_p]$ which still contains $[a, b]$, contradicting the fact that p does not have such a child. ◀

The following lemma shows that for any node $p = [a, b]$ in D_A, we can answer the query Top-2(p) using $w(p)$ additional bits if we know the answers to the Top-2 query on the parent node of p, and also the answers to the queries Top-2(a, b, A_1) and Top-2(a, b, A_2).

▶ **Lemma 10.** *Let A be a $2 \times n$ array. Given a non-root node $p = [a_p, b_p]$ in D_A, and its parent node $q = [a_q, b_q]$, if we know the answers to the query Top-2(q), then using the Top-2 encodings of A_1 and A_2 along with $w(p)$ additional bits, we can answer the query Top-2(p).*

Proof. If p is a leaf node (i.e., if $a_p = b_p$), we need $w(p) = 1$ extra bit to compare $A_1[a_p]$ and $A_2[a_p]$. If not, let (i_1, j_1) and (i_2, j_2), with $j_1 \leq j_2$, be the answers to the query Top-2(q). Also, for $i \in \{1, 2\}$, let f_i and s_i be the positions of the first and the second maxima, respectively, in the i-th row, $A_i[a_p \ldots b_p]$. Then we can answer the query Top-2(p) as follows. Without loss of generality, assume that $a_q < a_p$.

Case 1. $j_1 < j_2$: In this case, the interval p contains $f_{i_2} = j_2$, and this is the position of the maximum value in p. If $i_2 = 1$ ($i_2 = 2$), we can find the second maximum in p by comparing the values $A_1[s_1]$ and $A_2[f_2]$ ($A_2[s_2]$ and $A_1[f_1]$); the result of this comparison can be stored with $w(p) = 1$ extra bit.

Case 2. $j_1 = j_2$**:** In this case, the interval p does not contain j_1 $(= j_2)$. Therefore, to find the maximum element in p, we store the comparison between the values $A_1[f_1]$ and $A_2[f_2]$ using 1 bit. To find the second maximum element, if $A_1[f_1] > A_2[f_2]$ $(A_1[f_1] < A_2[f_2])$, then we store the comparison the values $A_2[f_2]$ and $A_1[s_1]$ $(A_1[f_1]$ and $A_2[s_2])$ using 1 extra bit. Thus the number of required extra bits is $w(p) = 2$. ◀

The following lemma bounds the total weight of all the nodes in D_A, which in turn bounds the extra space used by the Top-2 encoding of A in addition to the Top-2 encodings of the individual rows.

▶ **Lemma 11.** *For a $2 \times n$ array A, the sum of the weights of all nodes in D_A is at most $3n$.*

Proof. Let $f(p) = (r_p^f, c_p^f)$ and $s(p) = (r_p^s, c_p^s)$ be the positions of the first and the second largest elements in Top-2(p), respectively. Also, for each column $1 \leq j \leq n$, let f_j and s_j be the positions of the first and the second maxima in A, respectively, in the j-th column. We traverse D_A in level order. Whenever we visit a node $p = [a, b]$ in D_A, if $w(p) = 2$, then we pick the two positions $f(p)$ and $s(p)$, and otherwise (if $w(p) = 1$), we pick the position $s(p)$. We now claim that for all $1 \leq j \leq n$, f_j is picked at most twice, and s_j is picked at most once, during the level-order traversal of all the nodes in D_A. It is easy to show that statement of the lemma follows from this claim.

Case 1. Visiting a node p with $w(p) = 1$: We first show that any s_j is picked at most once. For $1 \leq j \leq n$, suppose that node p is the first node (in level order) which picks s_j. Since the only case in which this happens is when Top-2$(p) = \{f_j, s_j\}$, it follows that p is the unique node in D_A that picks s_j (as mentioned earlier, distinct nodes have distinct Top-2 answers, and s_j cannot be a position in the answers for a Top-2 query unless f_j is also an answer to the same query).

We now show that any f_j is picked at most twice. Suppose we pick f_j when we visit a node $p = [a_1, b_1]$. We need to prove that there can be at most one other node that can pick f_j. Assume, on the contrary, that there are two more distinct nodes $p_2 = [a_2, b_2]$, $p_3 = [a_3, b_3]$ such that we pick f_j when we visit these nodes. Since $w(p_2) = w(p_3) = 1$ (note that if the weight of a node is 2, then f_j can be picked at most once – see Case 2 in this proof), only f_j is picked as the second largest element when we visit p_2 and p_3. Also, by the construction of D_A, f_j is not picked if we pick f_j in any ancestor or descendant of p. Therefore, p_2 and p_3 are neither ancestor nor descendant of p, and by Proposition 8, for any two distinct $q, r \in \{p, p_2, p_3\}$, $q \not\subset r$ and $q \cap r \neq \emptyset$.

Now without loss of generality, suppose that $1 \leq a_1 < a_2 < a_3 < j < b_1 < b_2 < b_3 \leq n$. Note that if $f(p)$ exists between a_3-th and b_1-th column, Top-2(p) = Top-2(p_2) = Top-2(p_3) = $\{f(p), f_j\}$ since $s(p) = s(p_2) = s(p_3) = f_j$. This leads to a contradiction since distinct nodes should have distinct Top-2 answers (for the same reason, $f(p_2)$, and $f(p_3)$ cannot exist between a_3-th and b_1-th column). Therefore, $a_1 \leq c_p^f < a_3$ and $b_1 < c_{p_3}^f \leq b_3$. Now suppose that $b_1 < c_{p_2}^f \leq b_2$ (the case when $a_2 \leq c_{p_2}^f < a_3$ is analogous). Then f_j cannot be picked when we visit the node p_3 since the value in f_j is smaller than the values in both $f(p_2)$ and $f(p_3)$. This leads to a contradiction, proving that there can be at most two nodes whose weight is 1 which pick f_j during the traversal.

Case 2. Visiting a node p with $w(p) = 2$: In this case, we prove that neither $f(p)$ nor $s(p)$ are picked by any node other than p. (Thus, in this case, both $f(p)$ and $s(p)$ are picked only once.) By the construction of D_A, neither $f(p)$ nor $s(p)$ can be picked in any ancestor of p. Also, since neither $f(p)$ nor $s(p)$ can be the second largest elements in any of the descendants of p, we can't pick either of them after visiting the node p. We now claim that there is no node q such that $p \cap q \neq \emptyset$, $p \not\subset q$ and $q \not\subset p$. By Proposition 8, if

the claim is true, p has an intersection only with its ancestors or descendants, which do not pick both $f(p)$ and $s(p)$ during the traversal.

We assume, contrary to the above claim, that for the node $p = [a, b]$, there exists a node $q = [a_q, b_q]$ such that $p \cap q \neq \emptyset$, $p \not\subset q$ and $q \not\subset p$. Also without loss of generality, suppose that $1 \leq a_q < a < b_q < b \leq n$. Now consider the node r, which is an element in the lowest common ancestor (LCA)[4] of the nodes p and q. If any answer of the Top-2(r) query does not exist in $[a_q, b]$, one of r's child is a common ancestor of the nodes p and q, contradicting the fact that r is the LCA of p and q. Therefore, both answers of Top-2(r) exist in c-th and d-th column where $a_q \leq c \leq d \leq b$. Also, both nodes p and q can exist only if $a_q \leq c < a$ and $b_q < d \leq b$, in which case, $f([c + 1, b])$ exists in d-th column. Furthermore, by the construction of D_A, $c_s^f = d$ for any node s in the path from node $[c + 1, b]$ to node p. Therefore for any parent node of p, both answers of Top-2 exist in the d-th column since $w(p) = 2$, contradicting the fact that $b_q < d \leq b$. This leads to a contradiction that such q exists. ◀

▶ **Theorem 12.** *A $2 \times n$ array A can be encoded using $2 \lg \binom{3n}{n} + 3n + o(n)$ bits, to answer Top-2 queries.*

Proof. We first encode the first and the second rows in A using $2 \lg \binom{3n}{n} + o(n)$ bits, to answer Top-2 queries on each row, using the encoding in Lemma 5. For each node p in D_A in level order, we write down a $w(p)$-length bit-string which contains the additional bits needed to answer the query Top-2(p) (as mentioned in Lemma 10). The resulting bit-string, d_{D_A}, has length at most $3n$, by Lemma 11. A Top-2$(1, 2, a, b, A)$ query can be answered as follows. We find the last node $q = [a_q, b_q]$ in level order such that $a_q \leq a$ and $b \leq b_q$ using the Top-2 encodings for each row and the bit string d_{D_A}. Since Top-2(q) is same as the Top-2$(1, 2, a, b, A)$ by the Lemma 9, we can answer Top-2$(1, 2, a, b, A)$ by finding Top-2(a_q, b_q, A_1) and Top-2(a_q, b_q, A_2), and reading the appropriate $w(q)$ bits in d_{D_A} to pick the first and the second largest elements among these four candidates. ◀

5 Conclusion

In this paper, we obtained space-efficient encodings which answer Top-k queries on 2D arrays. In particular, for an $m \times n$ 2D array, we proposed an optimal encoding when the query is one-sided. We also proposed two different encodings that answer the general (four-sided) queries. Also when $k = 2$ and $m = 2$, we obtain an encoding which uses less space than the general encoding. We end with following open problems:

- Can we support the queries efficiently on our proposed encodings of Theorem 2, Theorem 7, and Theorem 12?
- For 2 and 3-sided queries, can we obtain encodings which use less space than the encoding for the 4-sided Top-k queries on 2D array?
- Is the effective entropy of unsorted Top-k queries smaller than the effective entropy of sorted Top-k queries on 2D arrays?

[4] For nodes p and q in DAG D, we define a LCA of p and q as the set of nodes whose out-degree is zero in the subgraph of D induced by the common ancestors of p and q.

References

1. Gerth S. Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012. doi:10.1007/s00453-011-9499-0.
2. Gerth Stølting Brodal, Andrej Brodnik, and Pooya Davoodi. The encoding complexity of two dimensional range minimum data structures. In *ESA 2013, 2013. Proceedings*, pages 229–240, 2013.
3. Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve, and Alejandro López-Ortiz. Online sorted range reporting. In *ISAAC 2009, Proceedings*, pages 173–182, 2009. doi:10.1007/978-3-642-10631-6_19.
4. Timothy M. Chan and Bryan T. Wilkinson. Adaptive and approximate orthogonal range counting. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013,2013*, pages 241–251, 2013.
5. Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.
6. Pawel Gawrychowski and Patrick K. Nicholson. Optimal encodings for range top- k k , selection, and min-max. In *ICALP 2015, Proceedings, Part I*, pages 593–604, 2015. doi:10.1007/978-3-662-47672-7_48.
7. Mordecai J. Golin, John Iacono, Danny Krizanc, Rajeev Raman, and S. Srinivasa Rao. Encoding 2d range maximum queries. In *ISAAC*, pages 180–189, 2011. doi:10.1007/978-3-642-25591-5_20.
8. Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti. Encodings for range selection and top-k queries. In *ESA 2013*, pages 553–564, 2013.
9. P. B. Miltersen. Cell probe complexity – a survey. *FSTTCS*, 1999.
10. Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti. Asymptotically optimal encodings for range selection. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014,2014*, pages 291–301, 2014.
11. Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 2007.

Efficient Index for Weighted Sequences

Carl Barton[1], Tomasz Kociumaka[*2], Solon P. Pissis[3], and Jakub Radoszewski[†4]

1 **The Blizard Institute, Barts and The London School of Medicine and Dentistry, Queen Mary University of London, UK**
c.barton@qmul.ac.uk
2 **Institute of Informatics, University of Warsaw, Warsaw, Poland**
kociumaka@mimuw.edu.pl
3 **Department of Informatics, King's College London, London, UK**
solon.pissis@kcl.ac.uk
4 **Institute of Informatics, University of Warsaw, Warsaw, Poland; and Department of Informatics, King's College London, London, UK**
jrad@mimuw.edu.pl

Abstract

The problem of finding factors of a text string which are identical or similar to a given pattern string is a central problem in computer science. A generalised version of this problem consists in implementing an index over the text to support efficient on-line pattern queries. We study this problem in the case where the text is *weighted*: for every position of the text and every letter of the alphabet a probability of occurrence of this letter at this position is given. Sequences of this type, also called position weight matrices, are commonly used to represent imprecise or uncertain data. A weighted sequence may represent many different strings, each with probability of occurrence equal to the product of probabilities of its letters at subsequent positions. Given a probability threshold $\frac{1}{z}$, we say that a pattern string P *matches* a weighted text at starting position i if the product of probabilities of the letters of P at positions $i, \ldots, i + |P| - 1$ in the text is at least $\frac{1}{z}$. In this article, we present an $O(nz)$-time construction of an $O(nz)$-sized index that can answer pattern matching queries in a weighted text over a constant-sized alphabet in optimal time. This improves upon the state of the art by a factor of $z \log z$ in construction time and space. Other applications of this data structure include an $O(nz)$-time construction of the weighted prefix table and an $O(nz)$-time computation of all covers of a weighted sequence, which improve upon the time complexity of the state of the art by the same factor.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases weighted sequence, position weight matrix, indexing, weighted suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.4

1 Introduction

Finding factors of a *text* resembling a *pattern* constitutes a classical problem in computer science. Apart from its theoretical interest, it is the core computation of many applications [14] such as search engines, bioinformatics, natural language processing and database search.

* Supported by the Polish Ministry of Science and Higher Education under the "Iuventus Plus" program in 2015-2016 grant no 0392/IP3/2015/73.
† The author is a Newton International Fellow. Supported by the Polish Ministry of Science and Higher Education under the "Iuventus Plus" program in 2015-2016 grant no 0392/IP3/2015/73.

In many situations the text can be considered as fixed and the patterns may arrive later. The algorithmic challenge is then to provide fast and direct access to all the factors of the text via the implementation of an *index*. The most widely used data structures for this purpose are the *suffix tree* and the *suffix array* [7]. These data structures can be constructed in $\mathcal{O}(n)$ time for a text of length n. Then all locations of a pattern of length m can be found in the optimal time $\mathcal{O}(m + Occ)$, where Occ is the number of occurrences.

The pattern matching problem for *uncertain* sequences has been less explored [12]. In this work we consider a type of uncertain sequences called *weighted sequences* (also known as position weight matrices, PWM). In a weighted sequence every position contains a subset of the alphabet and every letter is assigned a probability of occurrence such that at each position the probabilities sum up to 1. Such sequences are common in various applications: (i) data measurements such as imprecise sensor measurements; (ii) flexible modelling of DNA sequences such as DNA binding profiles; (iii) when observations are private and thus sequences of observations may have artificial uncertainty introduced deliberately.

In the *weighted pattern matching* (WPM) problem we are given a string of length m called a pattern, a weighted sequence of length n called a text, both over an alphabet Σ of size σ, and a *threshold probability* $\frac{1}{z}$. The task is to find all positions in the text where the fragment of length m represents the pattern with probability at least $\frac{1}{z}$. Each such position is called an *occurrence* of the pattern; we also say that the fragment and the pattern *match*. An $\mathcal{O}(\sigma n \log m)$-time solution for the WPM problem based on Fast Fourier Transform was proposed in [6]. This problem was also considered in [1] where a reduction to property matching in a text of size $\mathcal{O}(nz^2 \log z)$ was proposed.

In this article, we are interested in the indexing version of the WPM problem, that is, constructing an index to provide efficient procedures for answering queries related to the content of a fixed weighted sequence. In [11], the authors presented the *weighted suffix tree* allowing $\mathcal{O}(m + Occ)$-time WPM queries; the construction time and size of that data structure is $\mathcal{O}(n\sigma^{z \log z})$. A direct application of the results in [1] reduces the construction time and the size of that index to $\mathcal{O}(nz^2 \log z)$. The index structure built in [11] consists of a compacted trie of all of the factors with probability greater than or equal to $\frac{1}{z}$. A similar – though more general – indexing data structure, which assumes $z = \mathcal{O}(1)$, was presented in [4] with query time $\mathcal{O}(m + m \times Occ)$. Here we propose a tree-like data structure that is similar to the aforementioned ones which is, however, constructed and stored much more efficiently. Note that the proposed index is constructed and works for a predetermined parameter z, as opposed to the one of [4] which can additionally answer queries for $z' < z$.

Our model of computations. We assume word-RAM model with word size $w = \Omega(\log(nz))$. We consider the log-probability model of representations of weighted sequences in which probabilities can be multiplied exactly in $\mathcal{O}(1)$ time.

A common assumption in practice is that $\sigma = \mathcal{O}(1)$ since the most commonly studied alphabet is $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. In this case a weighted sequence of length n has a representation of $\mathcal{O}(n)$ size. We describe the indexing data structure under this assumption. In the Conclusions Section we briefly discuss the construction of the index for larger alphabets.

Our contribution. We present an $\mathcal{O}(nz)$-time construction of an $\mathcal{O}(nz)$-sized index that answers weighted pattern matching queries in optimal $\mathcal{O}(m + Occ)$ time improving upon [1] by a factor of $z \log z$. Applications of our data structure include an $\mathcal{O}(nz)$-time construction of the *weighted prefix table* and an $\mathcal{O}(nz)$-time computation of all *covers* of a weighted sequence, which improve upon [2] and [11], respectively, by the same factor in the complexity.

Structure of the article. In Section 2 basic notation related to weighted sequences, tries and compacted tries is presented. In particular, we introduce an important notion of *extensions of solid prefixes*, which is then used to construct an intermediate data structure that is crucial to our index, called *solid factor trie*, in Section 3. The weighted index is described in Section 4. First, in Section 4.1, we show how the main component of the index, *compacted trie of maximal solid factors*, is obtained from the solid factor trie, and then, in Section 4.2, a black-box description of the weighted index together with all the auxiliary data structures is given. Section 5 contains two examples of applications of the weighted index. We end with a Conclusions Section where we sketch changes to be made to the index in the case of a superconstant-sized integer alphabet.

2 Preliminaries

Let $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$ be an alphabet. A *string* S over Σ is a finite sequence of letters from Σ. By $S[i]$, for $1 \leq i \leq |S|$, we denote the i-th letter of S. The *empty string* is denoted by ε. By $S[i..j]$ we denote the string $S[i] \ldots S[j]$ called a *factor* of S (if $i > j$, then the factor is an empty string). A factor is called a *prefix* if $i = 1$ and a *suffix* if $j = |S|$. A factor U of a string S is called *proper* if $U \neq S$. By S^R we denote the reversal (the mirror image) of S.

▶ **Definition 1** (Weighted sequence). A weighted sequence $X = x_1 x_2 \ldots x_n$ of length $|X| = n$ over an alphabet $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$ is a sequence of sets of pairs of the form:

$$x_i = \{(s_j,\ \pi_i^{(X)}(s_j))\ :\ j \in \{1, 2, \ldots, \sigma\}\}.$$

If the considered weighted sequence is unambiguous, we write π_i instead of $\pi_i^{(X)}$. Here, $\pi_i(s_j)$ is the occurrence probability of the letter s_j at the position $i \in \{1, \ldots, n\}$. These values are non-negative and sum up to 1 for a given i.

The *probability of matching* of a string P with a weighted sequence X, both having the same length, equals

$$\mathcal{P}(P, X) = \prod_{i=1}^{|P|} \pi_i^{(X)}(P[i]).$$

We say that a string P *matches* a weighted sequence X with probability at least $\frac{1}{z}$, denoted by $P \approx_{\frac{1}{z}} X$, if $\mathcal{P}(P, X) \geq \frac{1}{z}$. By $X[i..j]$ we denote a weighted sequence called a *factor* of X and equal to $x_i \ldots x_j$ (if $i > j$, then the factor is an empty weighted sequence). We then say that a string P *occurs* in X at position i if P matches the factor $X[i..i + |P| - 1]$. We also say that P is a *solid factor* of X (starting, occurring) at position i. By $Occ_{\frac{1}{z}}(P, X)$ we denote the set of all positions where P occurs in X. The main problem considered in the article can be formulated as follows.

▶ **Problem** (Weighted Indexing).
Input: A weighted sequence X of length n over an alphabet Σ of size σ and a threshold probability $\frac{1}{z}$.
Queries: For a given pattern string P of length m, check if $Occ_{\frac{1}{z}}(P, X) \neq \emptyset$, compute $|Occ_{\frac{1}{z}}(P, X)|$, or report all elements of $Occ_{\frac{1}{z}}(P, X)$.

We say that P is a *(right-)maximal solid factor* of X at position i if P is a solid factor of X at position i and no string $P' = Ps$, for $s \in \Sigma$, is a solid factor of X at this position.

▶ **Fact 2** (Amir et al. [1]). *A weighted sequence has at most z different maximal solid factors starting at a given position.*

For each position of a weighted sequence X we define the *heaviest letter* as the letter with the maximum probability (breaking ties arbitrarily). By $\mathbf{X}$ we denote a string obtained from X by choosing at each position the heaviest letter. We call $\mathbf{X}$ the *heavy string* of X.

2.1 Extensions of solid factors

Let us fix a weighted sequence X of length n. If F is a solid factor of X starting at position i and ending at position j, $j \geq i-1$, then the string $F\mathbf{X}[j+1..n]$ is called *the extension of the solid factor* F. By $\mathcal{E}$ we denote the set of extensions of all solid factors of X.

▶ **Observation 3.** *$\mathcal{E}$ is exactly the set of extensions of all* maximal *solid factors of X.*

Proof. Let $F\mathbf{X}[j+1..n] \in \mathcal{E}$ be an extension of a solid factor F starting at position i and let $k \in \{j, \ldots, n\}$ be the maximum index such that $F\mathbf{X}[j+1..k]$ is a solid factor of X starting at position i. Then $M = F\mathbf{X}[j+1..k]$ is a maximal solid factor, as it cannot be extended by the most probable letter $\mathbf{X}[k+1]$, and $F\mathbf{X}[j+1..n] = M\mathbf{X}[k+1..n]$ is its extension. ◀

The following observation shows that $\mathcal{E}$ is closed under suffixes.

▶ **Observation 4.** *If $S \in \mathcal{E}$, $S \neq \varepsilon$, then the longest proper suffix S' of S also belongs to $\mathcal{E}$.*

Proof. Assume that S is an extension of a solid factor F. If $|F| \geq 1$, then S' is an extension of the longest proper suffix of F. Otherwise, S' is an extension of an empty factor. ◀

2.2 Tries

We consider rooted labeled trees with labels on edges, called *tries*. The labels are letters from Σ; edges going down from a single node have distinct labels. The root is denoted by *root*.

If T is a trie and u, v are its two nodes such that v is an ancestor of u, then by $str(u, v)$ we denote the string spelled by the edge labels on the path from u to v. We say that $\{str(u, root) : u \in T\}$ are *the suffixes of the trie* T. As usual by $lca(x, y)$ we denote the lowest common ancestor of the nodes x and y. By L_i for $i \geq 0$ we denote the i-th *level* of T that consists of nodes at depth i in the trie.

A *compacted trie* is a trie in which maximal paths whose inner nodes have degree 2 are represented as single edges with string labels. Usually such labels are not stored explicitly, but as pointers to a base string (or base strings); only the first letters are stored. The remaining nodes are called *explicit*, whereas the nodes that are removed due to compactification are called *implicit*. A well-known example of a compacted trie is a suffix tree of a string [7].

A *suffix tree of a trie* T, denoted by $\mathcal{S}(T)$, is a compacted trie of the strings $str(u, root)$ for $u \in T$; see [5, 15, 16]. The explicit nodes of $\mathcal{S}(T)$ that correspond to $str(u, root)$ for $u \in T$ are called *terminal* nodes. The string labels of the edges of $\mathcal{S}(T)$ are not stored explicitly, but correspond to upward paths in the trie T. For a node v of $\mathcal{S}(T)$, by $str(v)$ we denote the concatenation of labels of the edges from the root of $\mathcal{S}(T)$ to v.

▶ **Fact 5** (Shibuya [16]). *The suffix tree of a trie with N nodes has size $\mathcal{O}(N)$ and can be constructed in $\mathcal{O}(N)$ time.*

3 Solid factor trie

For a weighted sequence X of length n, a *solid factor trie* of X, denoted by $\mathcal{T}$, is a trie having as suffixes the *reversals of the strings from* $\mathcal{E}$. By this definition:

▶ **Observation 6.** *If S is a solid factor of X, then there exist nodes u, v in $\mathcal{T}$ such that $str(u, v) = S$.*

It turns out that the solid factor trie represents all maximal solid factors of X much more efficiently than if each of them was stored separately.

▶ **Lemma 7.** *The solid factor trie $\mathcal{T}$ has at most z nodes at each level.*

Proof. By Observation 4, each node at the level i in $\mathcal{T}$ comes from a string of length i in $\mathcal{E}$. By Observation 3 and Fact 2, there are at most z strings of length i in $\mathcal{E}$. ◀

We proceed with a construction of the solid factor trie in time linear in the size of the trie. For this, we need to equip the data structure with additional values; these enhancements will also turn out useful in the construction of the weighted index.

For each edge of the trie we store, in addition to its letter label, its probability defined as the probability of this letter at the respective position in X. If v is an ancestor of u, then by $\pi(u, v)$ we denote the product of probabilities of edges on the path from u to v. Let H be the heavy path in $\mathcal{T}$ that corresponds to $\mathbf{X}$ and let h be the leaf on this path. For each node v of $\mathcal{T}$ we retain the node $\mathit{back}(v)$ defined as $\mathit{lca}(v, h)$ and the probability $\pi\text{-}\mathit{back}(v) = \pi(v, \mathit{back}(v))$. We also denote $\mathit{str}\text{-}\mathit{back}(v) = \mathit{str}(v, \mathit{back}(v))$ (those values are not stored).

▶ **Theorem 8.** *The solid factor trie $\mathcal{T}$ of a weighted sequence X of length n can be constructed in $\mathcal{O}(nz)$ time.*

Proof. The trie is constructed by the algorithm Construct-$\mathcal{T}(X, n)$. We add new nodes to $\mathcal{T}$ level by level. First we extend the heavy path. A node v at level $i-1$ receives a child with an edge labeled by a letter s if and only if $s\,\mathit{str}\text{-}\mathit{back}(v)$ is a solid factor at position $n-i+1$; this condition is checked using the $\pi\text{-}\mathit{back}(v)$ values. Then we assign the child its values of back and $\pi\text{-}\mathit{back}$. The correctness of the algorithm follows from the claim below.

▶ **Claim.** After the i-th step of the outmost loop of the algorithm Construct-$\mathcal{T}(X, n)$, the trie's suffixes are the reversals of the strings from $\mathcal{E}$ of length at most i.

Proof. The proof goes by induction on i. The case of $i = 0$ is trivial. Let us assume that the claim holds for $i-1$ and prove that it then also holds for i. We need to show that if a node u is created by the algorithm at the i-th level, then $\mathit{str}(u, \mathit{root}) \in \mathcal{E}$ and, conversely, if $S \in \mathcal{E}$ is a string of length i, then a node u such that $\mathit{str}(u, \mathit{root}) = S$ is created by the algorithm at the i-th level. We prove the two implications separately.

(⇒) If the node u is created for some letter s, then, by the inductive hypothesis and the condition checked in the algorithm, $s\,\mathit{str}\text{-}\mathit{back}(v)$ is a solid factor of X starting at position $n-i+1$. Let j be the level of the node $\mathit{back}(v)$. Then:

$$\mathit{str}(u, \mathit{root}) = s\,\mathit{str}\text{-}\mathit{back}(v)\,\mathbf{X}[n-j+1..n] \in \mathcal{E}.$$

Algorithm Construct-$\mathcal{T}(X,\, n)$

$h_0 := root$; $L_0 := \{h_0\}$;
for $i := 1$ **to** n **do**
  Create a new node h_i being a child of h_{i-1} with the letter $\mathbf{X}[n-i+1]$;
  $back(h_i) := h_i$;
  $\pi\text{-}back(h_i) := 1$;
  $L_i := \{h_i\}$;
  foreach $v \in L_{i-1}$ **do**
    foreach $s \in \Sigma$ *in order of non-increasing* $\pi^{(X)}_{n-i+1}(s)$ **do**
      if $v = h_{i-1}$ **and** $s = \mathbf{X}[n-i+1]$ **then continue**;
      if $\pi^{(X)}_{n-i+1}(s) \cdot \pi\text{-}back(v) \geq \frac{1}{z}$ **then**
        Create a new node u being a child of v with the letter s;
        $back(u) := back(v)$;
        $\pi\text{-}back(u) := \pi^{(X)}_{n-i+1}(s) \cdot \pi\text{-}back(v)$;
        $L_i := L_i \cup \{u\}$;
      else break;

(⇐) Let S' be the longest proper suffix of S. Then $S' \in \mathcal{E}$ due to Observation 4. By the inductive hypothesis, there exists a node v in L_{i-1} such that $str(v, root) = S'$. Then S is an extension of the solid factor $s\, str\text{-}back(v)$, so indeed $\pi^{(X)}_{n-i+1}(s) \cdot \pi\text{-}back(v) \geq \frac{1}{z}$ and the node u corresponding to S will be created. ◀

Let us proceed with the complexity analysis. In each step of the innermost foreach-loop (apart from the step involving a node of the heavy path), either a new node is created or the execution of the loop is interrupted. For a given i, the former takes place $|L_i|$ times in total and the latter takes place at most $|L_{i-1}|$ times in total. The whole algorithm works in $\mathcal{O}(\sum_{i=0}^{n} |L_i|) = \mathcal{O}(nz)$ time due to Lemma 7. ◀

Let us introduce additional values to $\mathcal{T}$ that enable recovering the maximal solid factors of X. For a node $u \in L_i$, by $end(u)$ we denote its ancestor v such that $str(u, v)$ is a maximal solid factor at position $n-i+1$ in X. Moreover, by $len(u)$ we denote $|str(u, v)|$.

▶ **Lemma 9.** *The values $end(u)$ and $len(u)$ for all nodes u of $\mathcal{T}$ can be computed in $\mathcal{O}(nz)$ time.*

Proof. Clearly, it suffices to focus on the *end*-pointers, as the *len*-values can be computed from these pointers in linear time if only we store for each node its level in the trie.

For each node u, $end(u)$ is an ancestor of $back(u)$ (possibly equal to $back(u)$), therefore it is located on the heavy path H. For each node $v \in H$ from the leaf h up to the root we will set the *end*-pointers for all nodes u such that $end(u) = v$. In the computation we use the following property of the pointers:

▶ **Observation 10.** *If x is an ancestor of y, then $end(x)$ is an ancestor of $end(y)$.*

A node will be called *active* if it is a descendant of v such that its *end*-pointer has not been computed yet but its children's *end*-pointers have all been computed. After a node $v \in H$ has been considered, a set A containing all the active nodes u together with the values $\pi(u, v)$ is stored. Initially the set is empty.

For the next node $v \in H$ we first update the set A. If $v = h$, then we simply insert v to A with the probability 1. Otherwise, we iterate through all the nodes u in the set A and multiply their probabilities by the probability of the edge $\pi(v', v)$ where v' is the child of v on the heavy path. Then we insert to A all the leaves in the subtrees of $\mathcal{T}$ corresponding to children of v other than v'; their probabilities in A are the values of π-*back*.

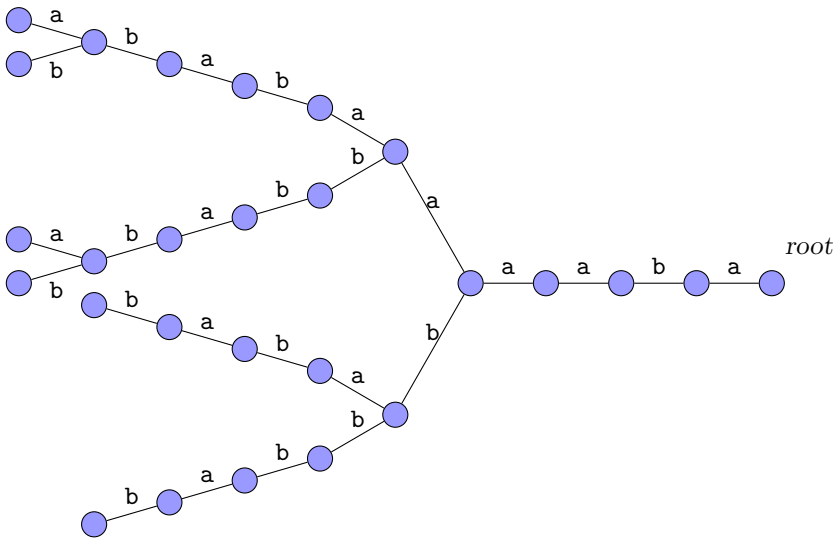
Next, we try to set the *end*-pointers for the elements of A and their ancestors. If v is the root, we simply set the pointers to the root to all the elements of A and their ancestors. Otherwise, let $w \in H$ be the parent of v. We iterate through all the elements $u \in A$ and for each of them check if $\pi(u, w) = \pi(u, v)\pi(v, w)$ is at least $\frac{1}{z}$. If so, we simply leave u in A for the next iterations. Otherwise, we set $\mathit{end}(u) = v$. If u was the last child of its parent for which we computed the *end*-pointer, we add the parent of u to A. In order to efficiently check this condition, each node counts its children whose *end*-pointer is yet to be determined.

The correctness of the algorithm follows from Observation 10. The running time is proportional to the total number of times a node from A is visited. When a node $v \in H$ is considered, for each node $u \in A$ either its *end*-pointer is set, which obviously happens at most $|\mathcal{T}| = \mathcal{O}(nz)$ times in total, or $\mathit{str}(u, v)$ corresponds to a left-maximal solid factor ending at position corresponding to the level of v in $\mathcal{T}$, which can happen at most z times by Fact 2. This implies $\mathcal{O}(nz)$ time complexity of the whole algorithm. ◀

▶ **Example 11.** The figure below shows an example of $\mathcal{T}$ for $z = 4$ and

$$X = [(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]\mathtt{bab}[(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)][(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]\mathtt{aaba}.$$

Among a few heavy strings of X, we can select $\mathbf{X} = \mathtt{ababaaaaba}$.



4 Construction of the weighted index

Our index for a weighted sequence X is based on a compacted trie of all maximal solid factors of X. We first show how this compacted trie can be constructed from the suffix tree $\mathcal{S}(\mathcal{T})$ of the solid factor trie $\mathcal{T}$. Next, we describe in detail all the components of the resulting weighted index.

4.1 Compacted trie of maximal solid factors

First of all, from Fact 5 and Lemma 7 we obtain an efficient construction of $\mathcal{S}(\mathcal{T})$:

▶ **Lemma 12.** *The suffix tree of the solid factor trie can be constructed in $\mathcal{O}(nz)$ time.*

The trie $\mathcal{T}$ represents more than the (maximal) solid factors of X, and so does $\mathcal{S}(\mathcal{T})$. However, the *len*-values that we computed in $\mathcal{T}$ let us delimit the maximal solid factors. Using them we can transform $\mathcal{S}(\mathcal{T})$ into a compacted trie $\mathcal{T}'$ of all maximal solid factors of X. Assume that in $\mathcal{S}(\mathcal{T})$ each terminal node stores, as its label, the starting position in X of the string from $\mathcal{E}$ that it represents (i.e., its depth). Then in $\mathcal{T}'$ a terminal's label is a list of starting positions in X of occurrences of the corresponding maximal solid factor.

▶ **Theorem 13.** *A compacted trie $\mathcal{T}'$ of all maximal solid factors of a weighted sequence X of length n can be constructed in $\mathcal{O}(nz)$ time.*

Proof. We start by constructing the solid factor trie $\mathcal{T}$ of X, together with the *len*-values, and its suffix tree $\mathcal{S}(\mathcal{T})$. By Theorem 8 and Lemmas 9 and 12, these steps take $\mathcal{O}(nz)$ time. Now it suffices to properly trim $\mathcal{S}(\mathcal{T})$. For a terminal node v in $\mathcal{S}(\mathcal{T})$ corresponding to $str(u, root)$ in $\mathcal{T}$, as $len(v)$ we store $len(u)$. Then we need to "lift" such a terminal node to depth $len(v)$ in $\mathcal{S}(\mathcal{T})$. In practice we proceed as follows.

For an (explicit or implicit) node u of $\mathcal{S}(\mathcal{T})$, by $maxlen(u)$ we denote the maximum value of $len(v)$ for a descendant terminal node v. As a result of trimming we leave only those (explicit or implicit) nodes u for which $maxlen(u)$ is at least as big as their depth in the trie; we call such nodes *relevant* nodes and the remaining nodes *irrelevant* nodes.

This procedure can be implemented in linear time. Indeed, the *maxlen*-values for all explicit nodes can be computed with a single bottom-up traversal. In another bottom-up traversal, we consider all irrelevant explicit nodes. Let w be such a node and let v be its explicit parent. Assume that v is located at depth d. If $maxlen(w) \leq d$, w is removed from $\mathcal{S}(\mathcal{T})$ and its label is appended to its parent's label. Otherwise, we cut the edge connecting v and w at depth $maxlen(w)$ and move the irrelevant node w there, making it relevant. ◀

4.2 The weighted index

As already mentioned, our weighted index is based on the compacted trie $\mathcal{T}'$ of all maximal solid factors of X. We also need to store the solid factor trie $\mathcal{T}$ which lets us access the string labels of the edges of the compacted trie. For convenience we extend each maximal solid factor in $\mathcal{T}'$ by a symbol $\$ \notin \Sigma$. As a result, each maximal solid factor corresponds to a leaf in $\mathcal{T}'$ which is labeled with a list of starting positions of its occurrences in X.

We assume left-to-right orientation of the children of each node (e.g., lexicographic). A global occurrence list OL is stored being a concatenation of the lists of occurrences in all the leaves of the trie $\mathcal{T}'$ in pre-order. Each node v stores, as $OL(v)$, the occurrence list of leaves in its subtree represented as a pair of pointers to elements of the global list OL. We enhance the occurrence list OL by a data structure for the following colored range listing problem.

▶ **Problem** (Colored range listing)**.** Preprocess a sequence $A[1..N]$ of elements from $[1..S]$ so that, given a range $A[i..j]$, one can list all the distinct elements in that range.

▶ **Fact 14** (Muthukrishnan [13])**.** *A data structure for the colored range listing problem of $\mathcal{O}(N)$ size can be constructed in $\mathcal{O}(N + S)$ time and answers queries in $\mathcal{O}(k + 1)$ time where k is the number of distinct elements reported.*

For all nodes of $\mathcal{T}'$ we also compute the following values (for the purpose of this computation we replace each leaf v with $|OL(v)|$ bogus leaves with single occurrences).

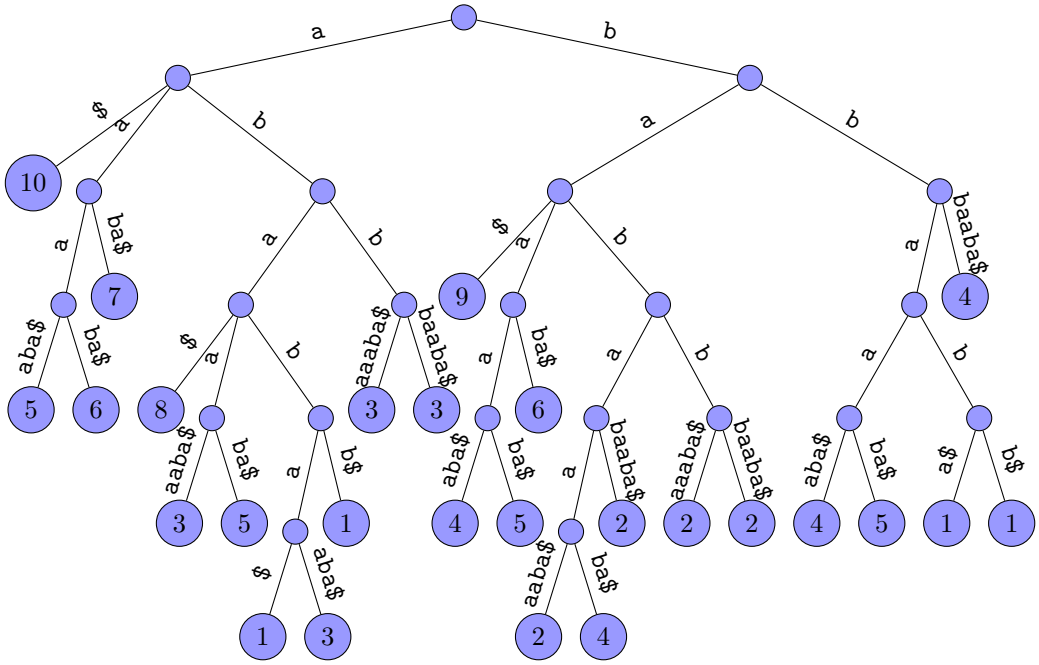
▶ **Fact 15** (Color set size, Hui [10]). *Given a rooted tree of size N with L leaves colored from $[1..S]$, in $\mathcal{O}(N+S)$ time one can find for each node u the number of distinct leaf colors in the subtree of u.*

We denote the resulting data structure as $\mathcal{I}$.

▶ **Theorem 16.** *The index $\mathcal{I}$ for a weighted sequence X can be constructed in $\mathcal{O}(nz)$ time. It answers decision and counting variants of weighted pattern matching queries in $\mathcal{O}(m)$ time, and, if required, reports all occurrences of the pattern in $\mathcal{O}(m+|Occ_{\frac{1}{z}}(P,X)|)$ time.*

Proof. The compacted trie $\mathcal{T}'$ can answer queries if $Occ_{\frac{1}{z}}(P,X) \neq \emptyset$ in $\mathcal{O}(m)$ time. We can use Fact 15 to equip each explicit node with the number of positions where the string represented by the node occurs. This way, $|Occ_{\frac{1}{z}}(P,X)|$ can also be determined in $\mathcal{O}(m)$ time. With the aid of the data structure for colored range listing, we can also report $Occ_{\frac{1}{z}}(P,X)$ in time proportional to the number of reported elements. ◀

▶ **Example 17.** The figure below shows the trie $\mathcal{T}'$ constituting the weighted index for the solid factor trie $\mathcal{T}$ shown in Example 11.



5 Applications of the weighted index

In this section we present two non-trivial applications of the weighted index. In both cases we improve the time complexity of the previously known results by a factor of $z \log z$.

5.1 Weighted longest common prefixes and weighted prefix table

For a weighted sequence X of length n and a pair of indices i, j, $1 \leq i, j \leq n$, by $wlcp(i,j)$ we denote the length of the longest solid factor that occurs in X at both positions i and j. After some preprocessing our weighted index allows to answer such queries in $\mathcal{O}(z)$ time.

▶ **Theorem 18.** *Given a weighted sequence X of length n, after $\mathcal{O}(nz)$-time preprocessing we can answer $wlcp(i,j)$ queries for any $1 \leq i,j \leq n$ in $\mathcal{O}(z)$ time.*

Proof. For each position i in X we precompute the list of leaves $L(i)$ of the weighted index $\mathcal{I}$ that contain i in their occurrence lists. Prior to that, all leaves are numbered in pre-order, and the elements of $L(i)$ are stored in this order. By Fact 2, $|L(i)| \leq z$ for each i.

Observe that $wlcp(i,j)$ is the maximum depth of a lowest common ancestor (lca) of a leaf in $L(i)$ and a leaf in $L(j)$. To determine this value, we merge the lists $L(i)$ and $L(j)$ according to the pre-order. The claim below (Lemma 4.6 in [7]) implies that, computing $wlcp(i,j)$, it suffices to consider pairs of leaves that are adjacent in the resulting list.

▶ **Claim.** If l_1, l_2 and l_3 are three leaves of a (compacted) trie such that l_2 follows l_1 and l_3 follows l_2 in pre-order, then $depth(lca(l_1,l_3)) = \min(depth(lca(l_1,l_2)), depth(lca(l_2,l_3)))$.

Merging two sorted lists, each of length at most z, takes $\mathcal{O}(z)$ time. Finally let us recall that lca-queries in a tree can be answered in $\mathcal{O}(1)$ time after linear-time preprocessing [3, 9]. ◀

The weighted prefix table $WPT[1..n]$ of X is defined as $WPT[i] = wlcp(1,i)$; see [2]. As a consequence of Theorem 18 we obtain an $\mathcal{O}(nz)$-time algorithm for computing this table. It outperforms the algorithm of [2], which works in $\mathcal{O}(nz^2 \log z)$ time.

▶ **Theorem 19.** *The weighted prefix table WPT of a given weighted sequence of length n can be computed in $\mathcal{O}(nz)$ time.*

5.2 Efficient computation of covers

A *cover* of a weighted sequence X is a string P whose occurrences as solid factors of X cover all positions in X; see [11]. More formally, if we define *maxgap* of an ordered set $A = \{a_1, \ldots, a_k\}$ (with $a_1 < \ldots < a_k$) as

$$maxgap(A) = \max\{a_i - a_{i-1} \,:\, i = 2, \ldots, k\},$$

then P is a cover of X if and only if

$$1 \in Occ_{\frac{1}{z}}(P,X) \quad \text{and} \quad maxgap(Occ_{\frac{1}{z}}(P,X) \cup \{n+1\}) \leq |P|.$$

Note that the former condition means exactly that P is a solid prefix of X. An $\mathcal{O}(n)$-time algorithm computing a representation of all the covers of a weighted sequence under the assumption that $z = \mathcal{O}(1)$ was presented in [11]. Here we show an algorithm that works in $\mathcal{O}(nz)$ time.

The algorithm of [11] uses a data structure (which we denote here by $\mathcal{D}$) to store a multiset of elements A from the set $\{2, \ldots, n\}$ allowing three operations:

1. initialisation with a given multiset of elements A;
2. computing $maxgap(\mathcal{D}) = maxgap(A \cup \{1, n+1\})$ for the currently stored multiset A;
3. removing a specified element from the currently stored multiset A.

The data structure has $\mathcal{O}(n)$ size, executes operation 1. in $\mathcal{O}(|A| + n)$ time and supports operations 2. and 3. in constant time. It consists of: (1) an array $C[1..n+1]$ that counts the multiplicity of each element; (2) a list L that stores all distinct elements of $A \cup \{1, n+1\}$ in ascending order and retains its maxgap; and (3) an array $P[1..n+1]$ that stores, for each distinct element of $A \cup \{1, n+1\}$, a pointer to its occurrence in L.

The algorithm of [11], formulated in terms of our index $\mathcal{I}$, works as follows. For a node v let $\mathcal{D}(v)$ be the $\mathcal{D}$-data structure storing the multiset $OL(v) \setminus \{1\}$. The path from the root

to each terminal node that represents a *maximal solid prefix* of X is traversed, and at each explicit node v the data structure $\mathcal{D}(v)$ is computed. To this end, when we move from a node v to its child w on the path, from $\mathcal{D}(v)$ we remove all elements from $OL(w')$ for w' being children of v other than w. Afterwards for the node w we perform the following check, which we call *cover-check*(w): if $maxgap(\mathcal{D}(w)) \leq depth(w)$, report the covers being prefixes of $str(w)$ of length $[\max(maxgap(\mathcal{D}(w)), depth(v)+1)..depth(w)]$. The whole procedure works in $\mathcal{O}(nz^2)$ time, as a single traversal works in linear time w.r.t. the size of the index and there are at most z maximal solid prefixes of X (Fact 2).

Let us show how this algorithm can be implemented to run in $\mathcal{O}(nz)$ time. We will call an explicit node of $\mathcal{I}$ a *prefix node* if it corresponds to a solid prefix of X. To implement the solution, it suffices for each prefix node to compute the $\mathcal{D}$-data structure and apply the *cover-check* routine. A prefix node will be called *branching* if it has more than one child being a prefix node, and *starting* if it is the root or its parent is branching. A maximal path going down the trie from a starting prefix node and passing only through non-starting prefix nodes will be called a *covering* path. Considering the prefix node subtree of $\mathcal{I}$, which contains at most z leaves and, consequently, at most $z-1$ branching nodes, we make the following easy but important observation.

▶ **Observation 20.** *There are $\mathcal{O}(z)$ covering paths and each prefix node belongs to exactly one of them.*

In the algorithm we compute the $\mathcal{D}$-data structures for all starting prefix nodes (by first computing the C-arrays) and then update the data structure efficiently along each covering path. The proofs of the following two lemmas are deferred to the full version of the article.

▶ **Lemma 21.** *$\mathcal{D}(v)$ for all starting prefix nodes v can be computed in $\mathcal{O}(nz)$ time.*

▶ **Lemma 22.** *The values $maxgap(\mathcal{D}(v))$ for all prefix nodes can be computed in $\mathcal{O}(nz)$ time.*

▶ **Theorem 23.** *A representation of size $\mathcal{O}(nz)$ of all covers of a weighted sequence X of length n can be computed in $\mathcal{O}(nz)$ time. In particular, all shortest covers of X can be determined in $\mathcal{O}(nz)$ time.*

Proof. To annotate all the covers on the edges of the index, we compute the maxgaps for all the prefix nodes using Lemma 22 and then apply the constant-time *cover-check* routine for each of the nodes. As for the shortest covers, there are at most z of them (as there are at most z different solid prefixes of X of a specified length, each with probability of occurrence at least $\frac{1}{z}$), so they can all be listed explicitly in $\mathcal{O}(nz)$ time and space. ◀

6 Conclusions

We have presented an index for weighted pattern matching queries which for a constant-sized alphabet has $\mathcal{O}(nz)$ size and admits $\mathcal{O}(nz)$ construction time. It answers queries in optimal $\mathcal{O}(m + Occ)$ time. We have also mentioned two applications of the weighted index. Our index outperforms the previously existing solutions by a factor of $z \log z$ in the complexity.

Generalization to integer alphabets. Let us briefly discuss how to adapt our index to a general integer alphabet. The size of the input is then the total length R of the lists in the representation of the weighted sequence. In the construction of the solid factor trie we need the list at each position to be ordered according to the probabilities of letters. As the size of each list to be sorted is $\min(z, \sigma)$ (at most z letters can have probability at least $\frac{1}{z}$), the

sorting requires $\mathcal{O}(R \log \min(\sigma, z))$ time. The construction of a suffix tree of a tree of [16] works for any integer alphabet. Finally, our weighted index is a compacted trie with children of a node being indexed by the letter of the alphabet. Hence, to avoid an increase of the complexity of a query for a particular child of a node, for a general alphabet one requires to store a hash table of children. With perfect hashing [8] the complexity does not increase but becomes randomized (Las Vegas, running time w.h.p.).

An open question is whether our weighted index, constructed for a predetermined z, can be adapted to answer weighted pattern matching queries for $z' < z$, as it is in the case of [4].

References

1 Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. *Theor. Comput. Sci.*, 395(2-3):298–310, April 2008. doi:10.1016/j.tcs.2008.01.006.

2 Carl Barton and Solon P. Pissis. Linear-time computation of prefix table for weighted strings. In Florin Manea and Dirk Nowotka, editors, *Combinatorics on Words, WORDS 2015*, volume 9304 of *LNCS*, pages 73–84. Springer, 2015. doi:10.1007/978-3-319-23660-5.

3 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *Latin American Symposium on Theoretical Informatics, LATIN 2000*, volume 1776 of *LNCS*, pages 88–94. Springer Berlin Heidelberg, 2000. doi:10.1007/10719839_9.

4 Sudip Biswas, Manish Patil, Sharma V. Thankachan, and Rahul Shah. Probabilistic threshold indexing for uncertain strings. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *19th International Conference on Extending Database Technology, EDBT 2016*, pages 401–412. OpenProceedings.org, 2016. doi:10.5441/002/edbt.2016.37.

5 Dany Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theor. Comput. Sci.*, 191(1-2):131–144, 1998. doi:10.1016/S0304-3975(96)00319-2.

6 Manolis Christodoulakis, Costas S. Iliopoulos, Laurent Mouchard, and Kostas Tsichlas. Pattern matching on weighted sequences. In *Algorithms and Computational Methods for Biochemical and Evolutionary Networks, CompBioNets 2004*, KCL publications, 2004.

7 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

8 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

9 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.

10 Lucas Chi Kwong Hui. Color set size problem with application to string matching. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, CPM 1992*, volume 644 of *LNCS*, pages 230–243. Springer, 1992. doi:10.1007/3-540-56024-6_19.

11 Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios K. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Inform.*, 71(2-3):259–277, 2006. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-07.

12 Yuxuan Li, James Bailey, Lars Kulik, and Jian Pei. Efficient matching of substrings in uncertain sequences. In Mohammed Javeed Zaki, Zoran Obradovic, Pang-Ning Tan,

Arindam Banerjee, Chandrika Kamath, and Srinivasan Parthasarathy, editors, *SIAM International Conference on Data Mining, SDM 2014*, pages 767–775. SIAM, 2014. doi:10.1137/1.9781611973440.88.

13 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In David Eppstein, editor, *13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002*, pages 657–666. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.

14 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. doi:10.1145/375360.375365.

15 Tetsuo Shibuya. Constructing the suffix tree of a tree with a large alphabet. In Alok Aggarwal and C. Pandu Rangan, editors, *Algorithms and Computation, ISAAC 1999*, volume 1741 of *LNCS*, pages 225–236. Springer, 1999. doi:10.1007/3-540-46632-0_24.

16 Tetsuo Shibuya. Constructing the suffix tree of a tree with a large alphabet. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(5):1061–1066, 2003.

Faster Longest Common Extension Queries in Strings over General Alphabets

Paweł Gawrychowski[*1], Tomasz Kociumaka[†2], Wojciech Rytter[‡3], and Tomasz Waleń[§4]

1 **Institute of Informatics, University of Warsaw, ul. Stefana Banacha 2, 02-097 Warsaw, Poland**
gawry@mimuw.edu.pl

2 **Institute of Informatics, University of Warsaw, ul. Stefana Banacha 2, 02-097 Warsaw, Poland**
kociumaka@mimuw.edu.pl

3 **Institute of Informatics, University of Warsaw, ul. Stefana Banacha 2, 02-097 Warsaw, Poland**
rytter@mimuw.edu.pl

4 **Institute of Informatics, University of Warsaw, ul. Stefana Banacha 2, 02-097 Warsaw, Poland**
walen@mimuw.edu.pl

Abstract

Longest common extension queries (often called longest common prefix queries) constitute a fundamental building block in multiple string algorithms, for example computing runs and approximate pattern matching. We show that a sequence of q LCE queries for a string of size n over a general ordered alphabet can be realized in $\mathcal{O}(q \log\log n + n \log^* n)$ time making only $\mathcal{O}(q+n)$ symbol comparisons. Consequently, all runs in a string over a general ordered alphabet can be computed in $\mathcal{O}(n \log\log n)$ time making $\mathcal{O}(n)$ symbol comparisons. Our results improve upon a solution by Kosolobov (Information Processing Letters, 2016), who gave an algorithm with $\mathcal{O}(n \log^{2/3} n)$ running time and conjectured that $\mathcal{O}(n)$ time is possible. We make a significant progress towards resolving this conjecture. Our techniques extend to the case of general unordered alphabets, when the time increases to $\mathcal{O}(q \log n + n \log^* n)$. The main tools are difference covers and the disjoint-sets data structure.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases longest common extension, longest common prefix, maximal repetitions, difference cover

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.5

1 Introduction

While many text algorithms are designed under the assumption of integer alphabet sortable in linear time, in some cases it is enough to assume general alphabet. A general alphabet

* Work done while the author held a post-doctoral position at Warsaw Center of Mathematics and Computer Science.
† Supported by Polish budget funds for science in 2013-2017 as a research project under the "Diamond Grant" program.
‡ Supported by the grant NCN2014/13/B/ST6/00770 of the Polish Science Center.
§ Supported by the grant NCN2014/13/B/ST6/00770 of the Polish Science Center.

can be either ordered, meaning that one can check if one symbol is less than another, or unordered, meaning that only equality of two symbols can be checked. Many classical linear-time string-matching algorithms (e.g. Knuth-Morris-Pratt, Boyer-Moore) work for any unordered general alphabet. Recently, a linear-time algorithm for computing the leftmost critical factorization in such model was given [11]. On the other hand, algorithms related to detecting repetitions usually need $\Omega(n \log n)$ equality tests [18], and an on-line algorithm matching this bound is known [13].

In this paper we consider the longest common extension problem (LCE, in short) in case of general ordered and unordered alphabets. The goal is to preprocess a given word w of length n for queries $\mathrm{LCE}(i, j)$ returning the length of the longest common factor starting at position i and j in w. Such queries are often a basic building block in more complicated algorithms, for example in computing runs [1, 2] as well as in approximate string matching [15].

For integer alphabets of polynomial size, one can preprocess a given string in linear time and space to answer any LCE query in constant time. Preprocessing space can be traded for query time [4, 5] and generalizations to trees [3] and grammar-compressed strings [9, 10, 16, 19] are known. The situation is more complicated for general alphabets. If the alphabet is ordered, then of course we can reduce it to $[1..n]$ by sorting the characters in $\mathcal{O}(n \log n)$ time and preprocess the obtained string in linear time and space to answer any LCE query in constant time. However this increases the total preprocessing time to $\mathcal{O}(n \log n)$. For unordered alphabet the situation is even worse, because the reduction would take $\mathcal{O}(n^2)$ time. A natural question is hence how efficiently we can answer a collection of such queries given one by one (on-line), where we measure the preprocessing time plus the total time taken by all the queries.

It is known that if we can perform on-line $\mathcal{O}(n)$ LCE queries for a given word of length n in total time $T(n)$ making $\mathcal{O}(n)$ symbol comparisons, then we can compute all runs in $\mathcal{O}(n+T(n))$ time making only $\mathcal{O}(n)$ symbol comparisons. An algorithm with $T(n) = \mathcal{O}(n \log^{2/3} n)$ time was recently presented by Kosolobov [14], who posed the existence of a linear-time algorithm as an open question. Much earlier, Breslauer [6] asked in his PhD thesis whether an easier task of square detection (equivalently, checking if a word has at least one run) is possible in linear time in the comparison model. In this paper we make a significant progress towards answering both questions by giving a faster algorithm with $T(n) = \mathcal{O}(n \log \log n)$.

Our result. For a given string of length n over a general ordered alphabet, we can answer on-line a sequence of q LCE queries in $\mathcal{O}(q \log \log n + n \log^* n)$ time making $\mathcal{O}(q + n)$ symbol comparisons. In particular, a sequence of $\mathcal{O}(n)$ queries can be answered in $\mathcal{O}(n \log \log n)$ time. Consequently, all runs in a string over a general ordered alphabet can be computed in $\mathcal{O}(n \log \log n)$ time making $\mathcal{O}(n)$ symbol comparisons. For a general unordered alphabet we answer q LCE queries in $\mathcal{O}(q \log n + n \log^* n)$ time, still making $\mathcal{O}(q + n)$ symbol comparisons.

Overview of the methods. At a very high level, our approach is similar to the one used by Kosolobov. We first show how to calculate $\min(\mathrm{LCE}(i, j), t)$ efficiently, where $t = \mathrm{polylog}\, n$. Then we use a difference cover to sample some positions in the text. Using "short" queries, we can efficiently construct a sparse suffix array for these sampled positions, which in turn allows us to calculate an arbitrary $\mathrm{LCE}(i, j)$ efficiently. The key difference is that instead of calculating $\min(\mathrm{LCE}(i, j), t)$ naively, we use a recursive approach. The main tool there is an efficient Union-Find structure. This is enough to answer $\mathcal{O}(n)$ short queries in $\mathcal{O}(n \log \log n \cdot \alpha(n \log \log n, n \log \log n))$ total time. We can remove the $\alpha(n \log \log n, n \log \log n)$ factor introducing another difference cover and carefully analyzing the running time of the Union-

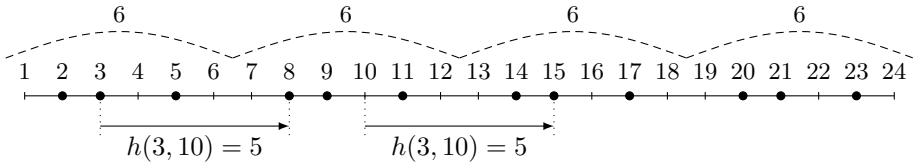


Figure 1 An example of a 6-cover $\mathbf{S}(6) = \{2, 3, 5, 8, 9, 11, 14, 15, 19, 20, 21, 23\}$ (for $\mathbf{D} = \{2, 3, 5\}$), with the elements marked as black circles. For example, we have $h(3, 10) = 5$, since $3+5,\ 10+5 \in \mathbf{S}(6)$.

Find data structure. Finally, we modify the algorithm to work faster when the number of queries q is smaller than n. The main insight allowing us to obtain $\mathcal{O}(q \log\log n + n \log^* n)$ total time is introducing multiple levels of difference covers with some additional properties. Such family of difference covers was implicitly provided in [8].

2 Preliminaries

A difference cover is a number-theoretic tool used throughout the paper. A set $\mathbf{D} \subseteq [0..t-1]$ is said to be a *t-difference-cover* if $[0..t-1] = \{ (x - y) \bmod t : x, y \in \mathbf{D} \}$.

▶ **Lemma 1** (Maekawa [17]). *For every integer t there is t-difference-cover of size $\mathcal{O}(\sqrt{t})$, which can be constructed in $\mathcal{O}(\sqrt{t})$ time.*

A subset X of $[1..n]$ is *t-periodic* if for each $i \in [1..n-t]$ we have: $i \in X \Leftrightarrow i + t \in X$.

A set $\mathbf{S} \subseteq [1..n]$ is called a *t-cover* of $[1..n]$ if $\mathbf{S}$ is t-periodic and there is a constant-time computable function h such that for $1 \le i, j \le n - t$ we have $0 \le h(i, j) \le t$ and $i + h(i, j), j + h(i, j) \in \mathbf{S}(t)$ (see Figure 1).

A t-cover can be obtained by taking a t-difference-cover $\mathbf{D}$ and setting $\mathbf{S}(t) = \{i \in [1..n] : i \bmod t \in \mathbf{D}\}$. This is a well-known construction implicitly used in [7], for example.

▶ **Lemma 2.** *For each $t \le n$ there is a t-cover $\mathbf{S}(t)$ of size $\mathcal{O}(\frac{n}{\sqrt{t}})$ which can be constructed in $\mathcal{O}(\frac{n}{\sqrt{t}})$ time.*

Our another tool is a disjoint-sets data structure. In this problem we maintain a family of disjoint subsets of $[1..n]$, initially consisting of singleton sets. We perform Find queries asking for a subset containing a given element, and Union operations which merge two subsets.

Note that the extremely fast-growing Ackermann function [21] is defined for $i, j \in \mathbb{Z}_{>0}$ as

$$A(i,j) = \begin{cases} 2^j & \text{if } i = 1, \\ A(i-1, 2) & \text{if } i > 1 \text{ and } j = 1, \\ A(i-1, A(i, j-1)) & \text{if } i > 1 \text{ and } j > 1. \end{cases}$$

Moreover, for $n, m \in \mathbb{Z}_{>0}$ $(m \ge n)$ one defines $\alpha(m, n) = \min\{i \ge 1 : A(i, \lfloor \frac{m}{n} \rfloor) > \log n\}$.

▶ **Lemma 3** (Tarjan [20]). *A sequence of up to n* Union *and m* Find *operations on an n-element set can be executed on-line in $\mathcal{O}(n + m \cdot \alpha(m + n, n))$ total time.*

The proof of the following lemma is deferred to the full version of the paper:

▶ **Lemma 4.** *For every $n, m \in \mathbb{Z}_{>0}$, we have $n + m \cdot \alpha(m + n, n) = \mathcal{O}(m + n \log^* n)$.*

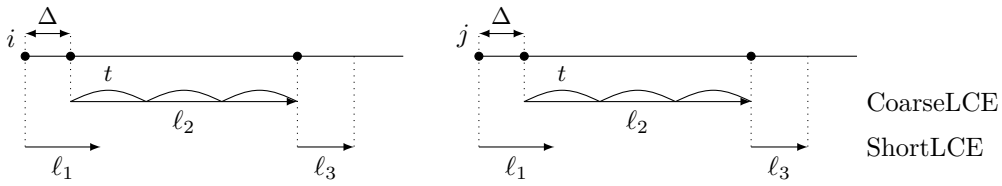


Figure 2 Illustration of Algorithm 1 for the case $\ell_1 \geq \Delta$.

3 Generic LCE algorithm for general ordered alphabets

We define t-short LCE queries by restricting the answer to at most t:

$$\text{ShortLCE}_t(i,j) \;=\; \min(\text{LCE}(i,j),\, t).$$

We define a t-block as a fragment of the input text w which starts in $\mathbf{S}(t)$ and has length t. If a position in $\mathbf{S}(t)$ lies near the end of w, we form a t-block from a suffix of w and enough dummy symbols to reach length t. We also introduce t-coarse LCE queries, which are LCE queries restricted to positions from $\mathbf{S}(t)$ returning the number of matching t-blocks:

$$\text{CoarseLCE}_t(i,j) \;=\; \begin{cases} \lfloor \text{LCE}(i,j)/t \rfloor & \text{if } i,j \in \mathbf{S}(t), \\ \bot & \text{otherwise.} \end{cases}$$

We now describe how to use ShortLCE and CoarseLCE queries for general LCE queries.

▶ **Lemma 5.** *If every sequence of q ShortLCE_t queries and CoarseLCE_t queries can be executed on-line in total time $T(n,q)$, then every sequence of q LCE queries can be executed on-line in total time $T(n,\mathcal{O}(q)) + \mathcal{O}(n+q)$.*

Proof. To calculate $\text{LCE}(i,j)$ we first check if $\text{LCE}(i,j) < t$ by calling $\text{ShortLCE}_t(i,j)$. If so, we are done. Otherwise, we can reduce computing $\text{LCE}(i,j)$ to computing $\text{LCE}(i+\Delta, j+\Delta)$ for any $\Delta \leq t$. In particular, we can choose $\Delta = h_t(i,j)$ so that $i+\Delta, j+\Delta \in \mathbf{S}(t)$. Then we call $\text{CoarseLCE}_t(i+\Delta, j+\Delta)$ which gives us the value $\lfloor \frac{1}{t}(\text{LCE}(i,j) - \Delta) \rfloor$. Computing the exact value of $\text{LCE}(i,j)$ requires another ShortLCE_t query; see Algorithm 1. The whole process is illustrated in Figure 2. ◀

Algorithm 1: GenericLCE(i,j)

$\ell_1 = \text{ShortLCE}_t(i,j)$
if $\ell_1 < t$ **then return** ℓ_1
$\Delta = h_t(i,j)$ ▷ $i+\Delta, j+\Delta \in \mathbf{S}(t)$
$\ell_2 = t \cdot \text{CoarseLCE}_t(i+\Delta, j+\Delta)$
$\ell_3 = \text{ShortLCE}_t(i+\Delta+\ell_2, j+\Delta+\ell_2)$
return $\Delta + \ell_2 + \ell_3$

4 ShortLCE_t queries in $\mathcal{O}(\log t)$ amortized time

In this section we show how to implement fast on-line ShortLCE_t queries. We assume that $t = 2^k$ and set $t' = \Theta(\log t)$ to be a smaller power of two. The amortized running time is $\mathcal{O}(\log t + \sqrt{\log t}\log^* n)$, which in particular is $\mathcal{O}(\log t)$ for $t = \log^{\Omega(1)} n$. The key components are Union-Find structures and t'-covers. We start with a simpler (and slightly slower) algorithm without t'-covers.

4.1 ShortLCE$_t$ queries in $\mathcal{O}(\log t \cdot \alpha((n+q)\log t, n\log t))$ amortized time

▶ **Lemma 6.** *A sequence of q $\mathrm{ShortLCE}_{2^k}(i,j)$ queries can be executed on-line in total time $\mathcal{O}((q+n)k \cdot \alpha((q+n)k, nk))$.*

Proof. We compute $\mathrm{ShortLCE}_{2^k}(i,j)$ using a recursive procedure; see Algorithm 2. The procedure first checks if $w[i..i+2^k-1]$ is already known to be equal to $w[j..j+2^k-1]$ using a Union-Find structure. If so, we are done. Otherwise, if $k=0$, we simply compare $w[i]$ and $w[j]$. If $k>0$, we recursively calculate $\mathrm{ShortLCE}_{2^{k-1}}(i,j)$ and, if the call returns 2^{k-1}, also $\mathrm{ShortLCE}_{2^{k-1}}(i,j)$. Finally, if both calls return 2^{k-1}, we update the Union-Find structure to store that $w[i..i+2^k-1] = w[j..j+2^k-1]$.

Algorithm 2: $\mathrm{ShortLCE}_{2^k}(i,j)$: compute $\mathrm{LCE}(i,j)$ up to length 2^k

```
if Find_k(i) = Find_k(j) then return 2^k
if k = 0 then
    if w[i] = w[j] then ℓ = 1 else ℓ = 0
else
    ℓ = ShortLCE_{2^{k-1}}(i, j)
    if ℓ = 2^{k-1} then
        ℓ = 2^{k-1} + ShortLCE_{2^{k-1}}(i + 2^{k-1}, j + 2^{k-1})
if ℓ = 2^k then Union_k(i, j)
return ℓ
```

To analyze the complexity of the procedure, we first observe that the total number of calls to Union is $\mathcal{O}(nk)$, because each such call discovers that $w[i..i+2^k-1] = w[j..j+2^k-1]$ (which was not known before). Moreover, these calls contribute $\mathcal{O}(nk)$ to the total running time. We argue that the number of executed Find queries and the running time of the remaining operations performed by $\mathrm{ShortLCE}_{2^k}(i,j)$ is proportional to $\mathcal{O}(k+1)$ plus the number of Union calls, which implies the lemma. For the sake of conciseness, #union denotes the number of calls to Union triggered by the considered call to ShortLCE (including itself).

We inductively bound the number of recursive calls triggered by $\mathrm{ShortLCE}_{2^k}(i,j)$:

$$\begin{cases} 2k+1+2\#\mathsf{union} & \text{if } w[i..i+2^k-1] \neq w[j..j+2^k-1], \\ 1+2\#\mathsf{union} & \text{if } w[i..i+2^k-1] = w[j..j+2^k-1]. \end{cases}$$

$\mathrm{ShortLCE}_1$ terminates immediately, so this holds for $k=0$. For $k>0$ we have four cases.

1. $w[i..i+2^k-1]$ is already known to be equal to $w[j..j+2^k-1]$. Then we terminate immediately.
2. $w[i..i+2^{k-1}-1] \neq w[j..j+2^{k-1}-1]$. Then the number of recursive calls triggered by $\mathrm{ShortLCE}_{2^{k-1}}(i,j)$ is $2k-1+2\#\mathsf{union}$ so the number of recursive calls triggered by $\mathrm{ShortLCE}_{2^k}(i,j)$ is $2k+2\#\mathsf{union}$.
3. $w[i..i+2^{k-1}-1] = w[j..j+2^{k-1}-1]$ but $w[i+2^{k-1}..i+2^k-1] \neq w[j+2^{k-1}..j+2^k-1]$. The number of recursive calls triggered by $\mathrm{ShortLCE}_{2^{k-1}}(i,j)$ and $\mathrm{ShortLCE}_{2^{k-1}}(i+2^{k-1}, j+2^{k-1})$ is $1+2\#\mathsf{union}$ and $2k-1+2\#\mathsf{union}$, respectively. The total number of triggered recursive calls is hence $2k+1+2\#\mathsf{union}$.
4. $w[i..i+2^{k-1}-1] = w[j..j+2^{k-1}-1]$ and $w[i+2^{k-1}..i+2^k-1] = w[j+2^{k-1}..j+2^k-1]$. The number of recursive calls triggered by both $\mathrm{ShortLCE}_{2^{k-1}}(i,j)$ and $\mathrm{ShortLCE}_{2^{k-1}}(i+$

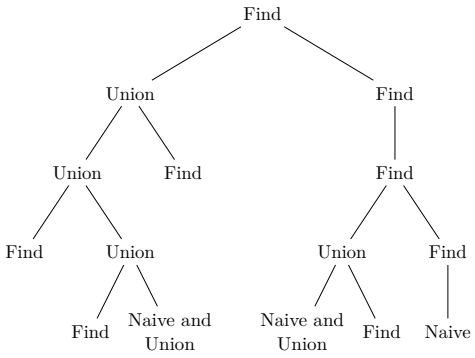


Figure 3 A recursion tree of $\mathrm{SparseShortLCE}_{t,t'}(i,j)$ for some example parameters such that $t = 2^4 t'$. The calls terminating with Union, Find and naive tests (in a segment of size t') are shown as nodes in the figure. The naive tests are only at the bottom of the tree and they are accompanied by Unions (except the last one).

$2^{k-1}, j + 2^{k-1})$ is $1 + 2\#\mathsf{union}$. However, $w[i..i + 2^k - 1]$ was not known to be equal to $w[j..j + 2^k - 1]$, so we then execute $\mathrm{Union}_k(i,j)$. Hence the total number of recursive calls is $1 + 2\#\mathsf{union}$ (rather than of $3 + 2\#\mathsf{union}$).

Consequently, the total running time follows from Lemma 3. ◀

4.2 Faster $\mathrm{ShortLCE}_t$ queries

Assume $t = 2^k = \Omega(\log n)$. We show how to reduce the factor $\alpha(qk + nk, nk)$ introducing a t'-cover, for $t' = 2^{k'}$. We define a sparse version of ShortLCE queries, which are ShortLCE queries restricted to positions from $\mathbf{S}(t')$:

$$\mathrm{SparseShortLCE}_{t,t'}(i,j) = \begin{cases} \mathrm{ShortLCE}_t(i,j) & \text{if } i,j \in \mathbf{S}(t') \\ \bot & \text{otherwise} \end{cases}$$

We slightly modify Algorithm 2 to obtain Algorithm 3, which computes $\min(\mathrm{LCE}(i,j), 2^k)$ for positions $i, j \in \mathbf{S}(t')$.

▶ **Lemma 7.** *A sequence of q $\mathrm{SparseShortLCE}_{2^k,2^{k'}}$ queries can be executed on-line in total time $\mathcal{O}(q(k + 2^{k'}) + n\sqrt{2^{k'}} + \frac{nk}{\sqrt{2^{k'}}} \log^* n)$.*

Algorithm 3: $\mathrm{SparseShortLCE}_{2^k,2^{k'}}(i,j)$: compute $\min(\mathrm{LCE}(i,j), 2^k)$ for $i, j \in \mathbf{S}(2^{k'})$

if $\mathrm{Find}_k(i) = \mathrm{Find}_k(j)$ **then return** 2^k
if $k = k'$ **then**
  Compute naively $\ell = \mathrm{ShortLCE}_{2^{k'}}(i,j)$
else
  $\ell = \mathrm{SparseShortLCE}_{2^{k-1},2^{k'}}(i,j)$
  if $\ell = 2^{k-1}$ **then**
    $\ell = 2^{k-1} + \mathrm{SparseShortLCE}_{2^{k-1},2^{k'}}(i + 2^{k-1}, j + 2^{k-1})$
if $\ell = 2^k$ **then** $\mathrm{Union}_k(i,j)$
return ℓ

Proof. The analysis is similar to the proof of Lemma 6. The total number of calls to Union is now only $\mathcal{O}(\frac{nk}{2^{k'/2}})$ because we always have that $i, j \in \mathbf{S}(2^{k'})$. Hence, excluding the cost of computing $\ell = \text{ShortLCE}_{2^{k'}}(i, j)$, the total time complexity is $\mathcal{O}(qk + \frac{nk}{2^{k'/2}} \log^* n)$ by the same reasoning as in Lemma 6, except that we additionally apply Lemma 4 to bound the running time of the Union-Find data structure (stated in Lemma 3).

Now we analyze the cost of computing $\ell = \text{ShortLCE}_{2^{k'}}(i, j)$. First, observe that for every original call to $\text{SparseShortLCE}_{2^k, 2^{k'}}(i, j)$ we have at most one such computation with $\ell < 2^{k'}$ (because it means that we have found a mismatch and no further recursive calls are necessary). On the other hand, if $\ell = 2^{k'}$, then we call $\text{Union}_{k'}(i, j)$, which may happen at most $\frac{n}{2^{k'/2}}$ times. Therefore, the total complexity of all these naive computations is $\mathcal{O}(n2^{k'/2} + q \cdot 2^{k'})$. ◀

Algorithm 4: $\text{FasterShortLCE}_{2^k, 2^{k'}}(i, j)$

Compute naively $\ell = \text{ShortLCE}_{2^{k'}}(i, j)$

if $\ell < 2^{k'}$ **then return** l

$\Delta = h_{2^{k'}}(i, j)$
$\ell = \Delta + \text{SparseShortLCE}_{2^k, 2^{k'}}(i + \Delta, j + \Delta)$
return $\min(\ell, 2^k)$

The next lemma is a direct consequence of Lemma 7 and Algorithm 4 with $2^{k'} = \Theta(k)$.

▶ **Lemma 8.** *A sequence of q ShortLCE_{2^k} queries can be executed on-line in total time $\mathcal{O}(qk + n\sqrt{k} \log^* n)$.*

5 CoarseLCE$_t$ queries

Let $t = \Omega(\log^2 n)$. Recall that we defined a t-block of w as a factor of size t starting in $\mathbf{S}(t)$. We want to show how to preprocess w in $\mathcal{O}(n \log \log n)$ time, so that any CoarseLCE_t query can be answered in constant time. To this end we proceed as follows:

1. sort all t-blocks in lexicographic order and remove duplicates,
2. encode every t-block with its rank on the sorted list,
3. construct a new string $\text{code}(w)$ of length $\mathcal{O}(n)$ over alphabet $[1..n]$, such that any CoarseLCE_t query can be reduced to an LCE query on $\text{code}(w)$,
4. preprocess $\text{code}(w)$ for LCE queries.

▶ **Lemma 9.** *For $t = \Omega(\log^2 n)$ we can lexicographically sort all t-blocks of w in $\mathcal{O}(n \log t)$ time.*

Proof. Two t-blocks can be lexicographically compared with a ShortLCE_t query. We have $\mathcal{O}(\frac{n}{\sqrt{t}})$ such blocks, hence one of the classical sorting algorithms they can be all sorted using $\mathcal{O}(\frac{n}{\sqrt{t}} \log n) = \mathcal{O}(n)$ queries. By Lemma 8, the total time to execute these queries and sort all t-blocks is therefore $\mathcal{O}(n \log t)$. ◀

We can use the lexicographic order of t-blocks to assign ranks to all t-blocks. Then we reduce CoarseLCE queries to LCE queries in a word $\text{code}(w)$ over an integer alphabet; see Figure 4.

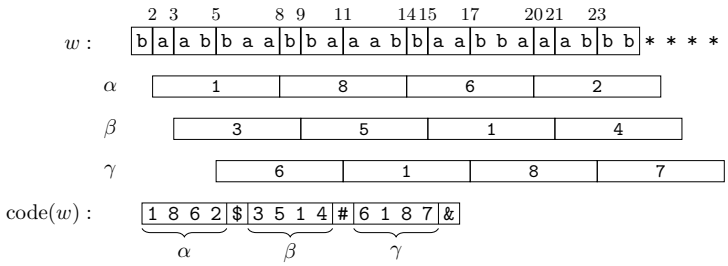


Figure 4 6-blocks of w are lexicographically sorted (using ShortLCE$_t$) and ranked. Then CoarseLCE$_6(2, 11)$ in w is reduced to LCE$(1, 12)$ in code(w).

▶ **Lemma 10.** *For $t = \Omega(\log^2 n)$ we can preprocess w in $\mathcal{O}(n \log t)$ time so that any* CoarseLCE$_t$ *query can be answered in constant time.*

Proof. Using Lemma 9, we assign a number to each t-block, so that two t-blocks are identical if and only if their numbers are equal. The number assigned to the block starting at position $p \in \mathbf{S}(t)$ is denoted rank(p). These numbers are ranks on a sorted list of length $|\mathbf{S}(t)|$, so rank$(p) \in [1..|\mathbf{S}(t)|]$. Then we construct a new string code(w) as follows. Let $\{ i_1, i_2, \ldots i_k \} = [1, t] \cap \mathbf{S}(t)$ and z_s be the word obtained from w by concatenating the numbers assigned to all t-blocks starting at positions $i_s, i_s + t, i_s + 2t, i_s + 3t, \ldots$:

$$z_s = \text{rank}(i_s)\text{rank}(i_s + t)\text{rank}(i_s + 2t)\text{rank}(i_s + 3t) \ldots .$$

Finally, we introduce k new distinct letters $\#_1, \#_2, \ldots, \#_s$ and construct code(w):

$$code(w) = z_1 \cdot \#_1 \cdot z_2 \cdot \#_2 \cdot z_3 \cdot \#_3 \cdots z_k \cdot \#_k.$$

Next, code(w) is preprocessed to answer LCE queries in constant time. A CoarseLCE$_t(p, q)$ query for positions $p, q \in \mathbf{S}(t)$ is answered by first computing positions p', q' corresponding to p, q in code(w). Formally, if $p = i_s \bmod t$, then $p' = |z_1 \#_1 z_2 \#_2 \ldots z_{s-1} \#_{s-1}| + \frac{p - i_s}{t} + 1$; q' is computed similarly. Then an LCE(p', q') query on code(w) returns CoarseLCE$_t(p, q)$. The positions p' and q' can be computed in constant time, so the total query time is constant. Preprocessing code(w) requires constructing its suffix array, which takes linear time for integer alphabets of polynomial size, and preprocessing it for range minimum queries, which also takes linear time. Hence the total preprocessing time is $\mathcal{O}(n \log t)$. ◀

▶ **Theorem 11.** *A sequence of $\mathcal{O}(n)$ LCE queries for a string over a general ordered alphabet can be executed on-line in total time $\mathcal{O}(n \log \log n)$ making only $\mathcal{O}(n)$ symbol comparisons.*

Proof. We set $t = \Theta(\log^2 n)$ and reduce each LCE query to constant number of CoarseLCE$_t$ queries and ShortLCE$_t$ queries as described in Lemma 5. Thus together with Lemma 8 and Lemma 10 we obtain that any sequence of q LCE queries for a string over a general ordered alphabet can be realized in $\mathcal{O}(n \log \log n)$ time. However, the total number of symbol comparisons used by the algorithm might be $\Omega(n \log \log n)$. This can be decreased to $\mathcal{O}(n)$ with yet another Union-Find data structure, where we maintain sets of positions already known to store the same letter. This is essentially the idea used in Lemma 7 of [12]. ◀

6 Faster solution for sublinear number of queries

The algorithm presented in the previous section is not efficient when the number of queries q is significantly smaller than the length of the string n. In this section we show that this can be avoided, and we present an $\mathcal{O}(q \log\log n + n \log^* n)$-time algorithm. This requires some nontrivial changes in our approach. In particular, we need a stronger notion of t-covers, which form a *monotone family*.

$\mathbf{S}(4^0), \mathbf{S}(4^1), \mathbf{S}(4^2), \ldots \subseteq [1, n]$ is a monotone family of covers if the following conditions hold for every k:

1. $\mathbf{S}(4^k)$ is a 4^k-cover (except that h_{4^k} is computable in $\mathcal{O}(k)$ instead of constant time).
2. $\mathbf{S}(4^{k+1}) \subseteq \mathbf{S}(4^k)$.
3. For any $i, j \in \mathbf{S}(4^k)$ we have that $h_{4^{k+1}}(i, j) \in \{0, 4^k, 2 \cdot 4^k\}$, and furthermore for such arguments $h_{4^{k+1}}$ can be evaluated in constant time.
4. $|\mathbf{S}(4^k)| \leq (\frac{3}{4})^k n$.

The existence of such a family is not completely trivial, in particular plugging in the standard construction of $\mathbf{S}(4^k)$ from Lemma 1 does not guarantee that $\mathbf{S}(4^{k+1}) \subseteq \mathbf{S}(4^k)$. The following lemma, implicitly shown in [8], provides an efficient construction.

▶ **Lemma 12** (Gawrychowski et al. [8], Section 4.1). *Let $\mathbf{S}(4^k)$ be the set of non-negative integers $i \in [1, n]$ such that none of the k least significant digits of the base-4 representation of i is zero. Then $\mathbf{S}(4^0), \mathbf{S}(4^1), \mathbf{S}(4^2), \ldots$ is a monotone family of covers, which can be constructed in $\mathcal{O}(n)$ total time.*

6.1 ShortLCE$_t$ queries with monotone family of covers

Similarly as in the proof of Lemma 8, we reduce ShortLCE queries to SparseShortLCE queries. However, now we slightly change the definition of SparseShortLCE queries so that there is only one parameter as follows:

$$\text{SparseShortLCE}_t(i, j) = \begin{cases} \text{ShortLCE}_t(i, j) & \text{if } i, j \in \mathbf{S}(t) \\ \bot & \text{otherwise} \end{cases}$$

▶ **Lemma 13.** *Consider a sequence of q SparseShortLCE$_{4^{k_i}}$ queries for $i \in \{1, \ldots, q\}$. The queries can be answered online in $\mathcal{O}((n+s) \cdot \alpha(n+s, n))$ time where $s = \sum_{i=1}^{q} T_i$ with $T_i = 1$ if the i-th query returns 4^{k_i} and $T_i = k_i + 1$ otherwise.*

Proof. We maintain a separate Union-Find structure for $\mathbf{S}(4^k)$ at every level $k \in \{0, \ldots, K\}$ where $K = \max_{i=1}^{q} k_i$. To answer a query for SparseShortLCE$_{4^k}$, we check if $\text{Find}_k(i) = \text{Find}_k(j)$ and if so, return 4^k. Otherwise, we calculate the answer with at most four calls to SparseShortLCE$_{4^{k-1}}$. This is possible because $\mathbf{S}(4^k) \subseteq \mathbf{S}(4^{k-1})$ and $\mathbf{S}(4^{k-1})$ is 4^{k-1}-periodic. Finally, we call $\text{Union}_k(i, j)$ if the answer is 4^k; see Algorithm 5.

We again analyze the number of recursive calls to SparseShortLCE$_{4^k}$ counting Union operations. The total number of unions at level k is $|\mathbf{S}(4^k)| \leq (\frac{3}{4})^k$, and in total this sums up to $\mathcal{O}(n)$. The amortized number of Find queries executed by a call to SparseShortLCE$_{4^k}$ is constant if $\text{LCE}(i, j) = 4^k$ and $\mathcal{O}(k+1)$ otherwise. These values also bound the running time of the remaining operations. Hence, by Lemma 3, the total time is as claimed. ◀

▶ **Lemma 14.** *A sequence of q queries ShortLCE$_{4^{k_i}}$ for $i \in \{1, \ldots, q\}$ can be answered online in total time $\mathcal{O}((n+s) \cdot \alpha(n+s, n)) = \mathcal{O}(n \log^* n + s)$ where $s = \sum_{i=1}^{q} (k_i + 1)$.*

k'	SparseShortLCE calls
0	$\text{SparseShortLCE}_{4^0}(1013\mathbf{0}_4, 00101_4) \to \Delta = 00001_4$
1	$\text{SparseShortLCE}_{4^1}(10131_4, 001\mathbf{0}2_4) \to \Delta = 00011_4$
1	$\text{SparseShortLCE}_{4^1}(102\mathbf{0}1_4, 00112_4) \to \Delta = 00021_4$
3	$\text{SparseShortLCE}_{4^3}(1\mathbf{0}211_4, 0\mathbf{0}122_4) \to \Delta = 01021_4$
return call	$\text{SparseShortLCE}_{4^4}(11211_4, 01122_4)$

Figure 5 An execution of $\text{ShortLCE}_{4^4}(i = (10130)_4, j = (00101)_4)$ (assuming $\text{LCE}(i,j) > 4^4$). The numbers are given in base-4 representation. Note that there is no $\text{SparseShortLCE}_{4^2}$ call.

Proof. We calculate $\text{ShortLCE}_{4^k}(i,j)$ using $\mathcal{O}(k)$ SparseShortLCE queries; see Algorithm 6. We iterate through $k' = 0, 1, \ldots, k-1$ maintaining Δ such that $0 \le \Delta \le \text{LCE}(i,j)$ and $i+\Delta, j+\Delta \in \mathbf{S}(4^{k'})$. Before incrementing k', we keep increasing Δ by $4^{k'}$ until $i+\Delta, j+\Delta \in \mathbf{S}(4^{k'})$ or $\Delta > \text{LCE}(i,j)$. The latter condition is checked by calling $\text{SparseShortLCE}_{4^{k'}}(i+\Delta, j+\Delta)$ and terminating if it returns less than $4^{k'}$. The while loop iterates at most twice, because $h_{4^{k'+1}} \in \{0, 4^{k'}, 2\cdot 4^{k'}\}$. Eventually, we either terminate having found the answer, or we can obtain it with a single call to $\text{SparseShortLCE}_{4^k}(i+\Delta, j+\Delta)$.

Let us analyze the total time complexity. Each call to ShortLCE_{4^k} performs up to k $\text{SparseShortLCE}_{4^{k'}}$ queries, but we terminate as soon as we obtain an answer other than $4^{k'}$. In Lemma 13, the last of these queries contributes $\mathcal{O}(k'+1) = \mathcal{O}(k+1)$ to s, while the remaining queries contribute one each. The total contribution of all $\text{SparseShortLCE}_{4^{k'}}$ queries called by a single ShortLCE_{4^k} query is therefore $\mathcal{O}(k+1)$. Hence, the total running time consumed by all $\text{SparseShortLCE}_{4^{k'}}$ queries is $\mathcal{O}((n+s)\cdot\alpha(n+s,n))$ where $s = \mathcal{O}(\sum_{i=1}^{q}(k_i+1))$. It is not hard to see that the remaining time consumed by a single ShortLCE_{4^k} query is $\mathcal{O}(k+1)$. This is partly because checking whether $i+\Delta$ and $j+\Delta$ belong to $\mathbf{S}(4^{k'+1})$ takes constant time, since we know that these indices are in $\mathbf{S}(4^{k'})$. Over all queries this sums up to $\mathcal{O}(s)$, which is dominated by the running time of the $\text{SparseShortLCE}_{4^{k'}}$ queries. The $\mathcal{O}(n \log^* n + s)$ upper bound follows from Lemma 4. ◀

6.2 Final algorithm

We first modify the implementation details for CoarseLCE to reduce the preprocessing time.

Algorithm 5: $\text{SparseShortLCE}_{4^k}(i,j)$: compute $\min(\text{LCE}(i,j), 4^k)$ for $i,j \in \mathbf{S}(4^k)$

```
if Find_k(i) = Find_k(j) then return 4^k
if k = 0 then
    if w[i] = w[j] then ℓ = 1 else ℓ = 0
else
    ℓ = 0
    for p = 0 to 3 do
        ℓ = ℓ + SparseShortLCE_{4^{k-1}}(i + p·4^{k-1}, j + p·4^{k-1})
        if ℓ < (p+1)·4^{k-1} then break
if ℓ = 4^k then Union_k(i, j)
return ℓ
```

Algorithm 6: $\mathrm{ShortLCE}_{4^k}(i,j)$

$\ell = \Delta = 0$
for $k' = 0$ **to** $k-1$ **do**
  while $i+\Delta \notin \mathbf{S}(4^{k'+1})$ **or** $j+\Delta \notin \mathbf{S}(4^{k'+1})$ **do**
    $\ell = \ell + \mathrm{SparseShortLCE}_{4^{k'}}(i+\Delta, j+\Delta)$ ▷ $i+\Delta, j+\Delta \in \mathbf{S}(4^{k'})$
    $\Delta = \Delta + 4^{k'}$
    if $\ell < \Delta$ **then return** $\min(4^k, \ell)$
return $\min(4^k, \Delta + \mathrm{SparseShortLCE}_{4^k}(i+\Delta, j+\Delta))$ ▷ $i+\Delta, j+\Delta \in \mathbf{S}(4^k)$

▶ **Lemma 15.** *For $t = \Omega(\log^6 n)$ we can preprocess a string of length n in $\mathcal{O}(n \log^* n)$ time, so that each $\mathrm{CoarseLCE}_t$ query can be answered in constant time.*

Proof. We set $k = \lceil \frac{1}{2} \log t \rceil$ and lexicographically sort all 4^k-blocks using $\mathrm{ShortLCE}_{4^k}$ queries. The number of blocks is at most $(\frac{3}{4})^k n \le \frac{n}{t^{0.5 \log 0.75}} \le \frac{n}{t^{0.2}}$. By Lemma 14, the sorting time is:

$$\mathcal{O}\left(\frac{n}{t^{0.2}} \log n \log t + n \log^* n\right) = \mathcal{O}\left(n \frac{\log n \log\log n}{\log^{1.2} n} + n \log^* n\right) = \mathcal{O}(n \log^* n).$$

Then we proceed as in the proof of Lemma 10. ◀

By combining Lemma 15 and Lemma 14, we obtain the final theorem.

▶ **Theorem 16.** *A sequence of q LCE queries for a string over a general ordered alphabet can be executed on-line in total time $\mathcal{O}(q \log\log n + n \log^* n)$ making $\mathcal{O}(q+n)$ symbol comparisons.*

7 Final remarks

We gave an $\mathcal{O}(n \log\log n)$-time algorithm for answering on-line $\mathcal{O}(n)$ LCE queries for general ordered alphabet. It is known (see [14]) that the runs of the string can be computed in $\mathcal{O}(T(n))$ time, where $T(n)$ is the time to execute on-line $\mathcal{O}(n)$ LCE queries. Hence our algorithm implies the following result:

▶ **Corollary 17.** *The runs of a string over general ordered alphabet can be computed in $\mathcal{O}(n \log\log n)$ time.*

Our algorithm is a major step towards a positive answer for a question posed by Kosolobov [14], who asked if $\mathcal{O}(n)$ time algorithm is possible.

It is also natural to consider general unordered alphabets, that is, strings where the only allowed operation is checking equality of two characters.

▶ **Theorem 18.** *A sequence of q LCE queries for a string over a general unordered alphabet can be executed in $\mathcal{O}(q \log n + n \log^* n)$ time making $O(n+q)$ symbol equality-tests.*

Proof. We can use the faster $\mathrm{ShortLCE}_{4^k}$ algorithm described in Section 6.1 with $k = \lceil \frac{1}{2} \log n \rceil$. Observe that in this approach we did not use the order of the characters, and thus it still works for unordered alphabets. ◀

Note that for unordered alphabets the reduction by Kosolobov [14] (see also [2]) from computing runs to LCE queries no longer works. Actually, deciding whether a given string is square-free already requires $\Omega(n \log n)$ comparisons, as shown by Main and Lorentz [18]. On the other hand for $\mathcal{O}(n)$ LCE queries $\mathcal{O}(n)$ equality tests always suffice.

References

1 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. A new characterization of maximal repetitions by Lyndon trees. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 562–571. SIAM, 2015. doi:10.1137/1.9781611973730.38.

2 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem, 2015. arXiv:1406.0263v7.

3 Philip Bille, Paweł Gawrychowski, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Longest common extensions in trees. *Theor. Comput. Sci.*, 2015. In press. doi:10.1016/j.tcs.2015.08.009.

4 Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching, CPM 2015*, volume 9133 of *LNCS*, pages 65–76. Springer, 2015. doi:10.1007/978-3-319-19929-0_6.

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

6 Dany Breslauer. *Efficient String Algorithmics*. PhD thesis, Columbia University, 1992. URL: http://www.cs.columbia.edu/~library/theses/breslauer.ps.gz.

7 Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, CPM 2003*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003. doi:10.1007/3-540-44888-8_5.

8 Paweł Gawrychowski, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Universal reconstruction of a string. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures, WADS 2015*, volume 9214 of *LNCS*, pages 386–397. Springer, 2015. doi:10.1007/978-3-319-21840-3_32.

9 Shunsuke Inenaga. A faster longest common extension algorithm on compressed strings and its applications. In Jan Holub and Jan Žďárek, editors, *Prague Stringology Conference 2015*, pages 1–4. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2015. URL: http://www.stringology.org/event/2015/p01.html.

10 Marek Karpiński, Wojciech Rytter, and Ayumi Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.*, 4(2):172–186, 1997.

11 Dmitry Kosolobov. Finding the leftmost critical factorization on unordered alphabet, 2015. arXiv:1509.01018.

12 Dmitry Kosolobov. Lempel-Ziv factorization may be harder than computing all runs. In Ernst W. Mayr and Nicolas Ollinger, editors, *Theoretical Aspects of Computer Science, STACS 2015*, volume 30 of *LIPIcs*, pages 582–593. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.STACS.2015.582.

13 Dmitry Kosolobov. Online detection of repetitions with backtracking. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching, CPM 2015*, volume 9133 of *LNCS*, pages 295–306. Springer, 2015. doi:10.1007/978-3-319-19929-0_25.

14 Dmitry Kosolobov. Computing runs on a general alphabet. *Information Processing Letters*, 116(3):241–244, 2016. doi:10.1016/j.ipl.2015.11.016.

15 Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989. doi:10.1016/0196-6774(89)90010-2.

16 Yury Lifshits. Processing compressed texts: A tractability border. In Bin Ma and Kaizhong Zhang, editors, *Combinatorial Pattern Matching, CPM 2007*, volume 4580 of *LNCS*, pages 228–240. Springer, 2007. doi:10.1007/978-3-540-73437-6_24.

17 Mamoru Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, May 1985. doi:10.1145/214438.214445.

18 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.

19 Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In Alberto Apostolico and Jotun Hein, editors, *Combinatorial Pattern Matching, CPM 1997*, volume 1264 of *LNCS*, pages 1–11. Springer, 1997. doi:10.1007/3-540-63220-4_45.

20 Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975. doi:10.1145/321879.321884.

21 Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984. doi:10.1145/62.2160.

Succinct Online Dictionary Matching with Improved Worst-Case Guarantees

Tsvi Kopelowitz[*1], **Ely Porat**[2], **and Yaron Rozen**[3]

1 **University of Michigan, Ann Arbor, Michigan, USA**
kopelot@gmail.com
2 **Bar Ilan University, Ramat Gan, Israel**
porately@cs.biu.ac.il
3 **Bar Ilan University, Ramat Gan, Israel**
yaron1828@gmail.com

Abstract

In the online dictionary matching problem the goal is to preprocess a set of patterns $D = \{P_1, \ldots, P_d\}$ over alphabet Σ, so that given an online text (one character at a time) we report all of the occurrences of patterns that are a suffix of the current text before the following character arrives. We introduce a succinct Aho-Corasick like data structure for the online dictionary matching problem. Our solution uses a new succinct representation for multi-labeled trees, in which each node has a set of labels from a universe of size λ. We consider lowest labeled ancestor (LLA) queries on multi-labeled trees, where given a node and a label we return the lowest proper ancestor of the node that has the queried label.

In this paper we introduce a succinct representation of multi-labeled trees for $\lambda = \omega(1)$ that support LLA queries in $O(\log\log\lambda)$ time. Using this representation of multi-labeled trees, we introduce a succinct data structure for the online dictionary matching problem when $\sigma = \omega(1)$. In this solution the worst case cost per character is $O(\log\log\sigma + occ)$ time, where occ is the size of the current output. Moreover, the amortized cost per character is $O(1 + occ)$ time.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Succinct indexing, dictionary matching, Aho-Corasick, labeled trees

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.6

1 Introduction

One of the crucial components of Network Intrusion Detection Systems (NIDS) is the ability to detect the presence of viruses and malware in streaming data. This task is typically executed by searching for occurrences of special digital signatures which indicate the presence of harmful intent. While searching for one such signature is often a fairly simple task, NIDS has to deal with the task of searching for many signatures in parallel. In such settings it is required that both the time spent on each packet of data and the total space usage are extremely small. Currently, the task of finding these signatures dominates the performance of such security tools [32], and several practical approaches have been suggested [9, 10]. The theoretical model for this problem is known as the (online) dictionary matching problem, which is a well studied problem [1, 2, 3, 4, 11, 13, 14] and is defined next.

* Work supported in part by NSF grants CCF-1217338, CNS-1318294, and CCF-1514383.

Dictionary matching. In the *dictionary matching problem* the input is a dictionary $D = \{P_1, P_2, ..., P_d\}$ of patterns and a text $T = t_1t_2...t_N$, all over alphabet Σ, where $\sigma = |\Sigma|$. The goal is to list all pairs (i, j) such that $t_{i-|P_j|+1}...t_i = P_j$. Let $n = \sum_{i=1}^{d} |P_i|$, and let $n_{max} = \max_{P \in D}\{|P|\}$. For a dictionary D the *prefix set* of D, denoted by $P(D)$, is the set of all prefixes of patterns in D. Let $m = |P(D)|$ and notice that $m \leq n + 1$. We assume Σ is an integer alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, and that $\sigma \leq m$. The Aho-Corasick (AC) data structure [1] solves the dictionary matching problem using $O(m \log m)$ bits of space and in $O(|T| + occ)$ time, where occ is the size of the output.

Online dictionary matching. In the *online dictionary matching problem* the input is the same as in the dictionary matching problem, but here the text T arrives online (character by character) and the goal is to report all of the occurrences of patterns from D as soon as they appear (before the next character arrives). For a dictionary D and text T let S_i be the longest suffix of $t_1t_2 \ldots t_i$ such that $S_i \in P(D)$. The AC data structure works in the online model by repeatedly finding S_{i+1} from S_i and t_{i+1} (and then also reporting all of the patterns from D that are suffixes of S_{i+1}). The amortized cost for this process, ignoring the work for reporting the output, is constant. However, the worst-case time per character in the AC data structure can be as large at $\Theta(n_{max})$. This may be too large for real-time applications, such as those that occur in NIDS.

One naïve way of tackling this problem is by using an automata with a state for each prefix in $P(D)$, where each state has σ outgoing transitions. However, this approach introduces a blow up in space, which in practice means that the entire data structure cannot fit in fast memory. Moreover, even the $O(m \log m)$ bit implementation of the AC data structure may be too large. Thus, a large body of recent work has focused on succinct representations of the AC data structure.

Succinct data structures. Given a combinatorial object a representation of the object is *succinct* if it uses $z + o(z)$ bits of space where z is the *information theoretic lower bound* for the number of bits representing the object. The main challenge when using a succinct representation is supporting the algorithmic operations with costs that are as efficient as in the non-succinct representation.

A growing trend in recent years has focused on developing succinct representations for the dictionary matching problem; see Table 1. The information theoretic lower bound for a dictionary of size n over alphabet σ is $n \log \sigma$ bits which is significantly less than the $O(m \log m)$ bits used by the AC data structure, when $\sigma << n$. However, much like in the AC data structure, current succinct representations also pay $\Theta(n_{max})$ time per character in the worst-case. We emphasize that Hon et al. [21] presented a solution using $O(m \log \sigma)$ bits (which is not succinct) and the worst-case cost per character is $O(\log \log m)$ time.

1.1 Our Results

In this paper we introduce a new succinct representation of the AC data structure with an implementation that supports low time cost per character in the worst-case. Such a solution addresses the type of constraints that show up in practical settings, such as in NIDS, where the space usage is limited and the worst-case time per character needs to remain low. Our succinct representation is summarized as follows.

▶ **Theorem 1.** *For $\sigma = \omega(1)$ there exists a succinct data structure for the online dictionary matching problem using $m(H_k(D) + 5 + o(1)) + 2\sigma + O(d \log \frac{n}{d})$ bits of space where the worst-*

Table 1 Comparison of the results.

Algorithm	Space	Worst-case Time per Character	Total Time
AC (NFA) [1]	$O(m \log m)$	$O(n_{max})$	$O(\|T\| + occ)$
AC (DFA) [1]	$O(m\sigma \log(m\sigma))$	$O(1)$	$O(\|T\| + occ)$
Chan et al. [12]	$O(m\sigma)$	$O(\log^2 m)$	$O((\|T\| + occ) \log^2 m)$
Hon et al. [21]	$O(m \log \sigma)$	$O(\log \log m)$	$O(\|T\| \log \log m + occ)$
Belazzougui [7]	$m(H_0(D) + 3.443 + o(1)) + O(d \log \frac{n}{d})$	$O(n_{max})$	$O(\|T\| + occ)$
Hon et al. [22]	$m(H_k(D) + 5 + o(1)) + O(d \log \frac{n}{d})$	$O(n_{max})$	$O(\|T\| + occ)$
New ($\sigma = \omega(1)$)	$m(H_k(D) + 5 + o(1)) + 2\sigma + O(d \log \frac{n}{d})$	$O(\log \log \sigma)$	$O(\|T\| + occ)$

case time per character is $O(\log \log \sigma)$*, and the total time for a text query* T *is* $O(|T| + occ)$ *where occ is the size of the output.*

Our main technique is a succinct representation of multi-labeled trees of size n, where each node in the tree has a set of labels drawn from a set $\mathcal{L}$ where $\lambda = |\mathcal{L}|$. The operations of interest on multi-labeled trees are label dependent. In particular we will be interested in lowest labeled ancestor (LLA) queries where given a node u and a label ℓ we need to report the lowest proper ancestor of u that has label ℓ. We show in Sections 4 and 5 how to support such operations for general trees. Strikingly, the type of trees in our implementation of the AC data structure exhibit some special combinatorial properties. Their properties allow an even more succinct representation for these trees which efficiently support LLA queries and other label dependent operations.

In this paper we propose a representation of multi-labeled trees that is succinct when $\lambda = \omega(1)$. Although we mainly consider the *LLA* operation, our representation supports many other operations as well and is succinct for more cases. Moreover, we find our implementation of the *LLA* operation to be simpler than previous approaches (see below).

1.2 Related Work

The notion of succinct data structures was introduced by Jacobson [24] with succinct data structures for bit-arrays, trees and graphs. Many succinct representations for combinatorial objects have since been developed, including succinct representations of sets [24, 26, 30], strings [28, 8], and trees [27, 17].

The first solution for the dictionary matching problem using less than $O(m \log m)$ bits was introduced by Chan et al. in [12]. Their solution also solves the dynamic variant of the problem. Other solutions are based on using suffix trees [23, 21] and are slower than the AC algorithm.

The first representation for the dictionary matching problem in succinct space without a query slowdown was introduced by Belazzougui [7] which was slightly improved by Hon et al. [22]. Succinct representations have also been developed for some variations of the dictionary matching problem, such as dynamic dictionary matching [21, 15], 2D dictionary matching [29], and approximate dictionary matching [21].

Labeled and multi-labeled trees. The problem of representing labeled trees was first considered by Geary, Raman and Raman [17]. However, their solution is succinct only for $\lambda = o(\frac{\log \log n}{\log \log \log n})$. Ferragina et al. [16] proposed a representation of labeled trees based on the XML Burrows–Wheeler transform. However, their representation does not support *LLA* queries. Barbay et al. [5, 6] introduced a representation for labeled trees and multi-labeled

trees supporting a restricted set of operations which does not include LLA queries. Moreover, their representation is succinct only when $\frac{t}{n} = \lambda^{o(1)}$.

The only known representation of labeled trees which supports LLA queries using succinct space are the solutions of He et al. [20] and Tsur [31]. Although these solutions are for the labeled case, they can be extended for multi-labeled trees using the same techniques of Barbay et al. [5], but then they would only be succinct when $\frac{t}{n} = \lambda^{o(1)}$.

2 Preliminaries

2.1 The Aho-Corasick data structure

The Aho-Corasick (AC) data structure [1] is a multi-pattern extension of the KMP data structure [25]. Since the AC data structure is in the core of this paper, we present its internals in some more detail.

The AC data structure is built upon a *trie* storing the patterns in D. The trie edges have the properties that each edge is labeled by a character $\sigma \in \Sigma$, and any two edges leaving the same node have different labels. Thus, there is a bijection between nodes in the trie and prefixes in $P(D)$. For a prefix $u \in P(D)$ let $state(u)$ be the node in the trie corresponding to u. Then u is the concatenation of the edge labels on the path from the root of the trie to $state(u)$. When it is clear from context, we sometimes abuse notation and refer to $state(u)$ as u itself.

The edges of the trie are termed as *forward links*. In addition to the forward links, there are also *failure links* and *report links*. For $u, v \in P(D)$ there is a failure link from node u to node v if and only if v is the longest string in $P(D)$ that is a proper suffix of u. Similarly, for $u \in P(D)$ and $v \in D$ there is a report link from node u to node v if and only if v is the longest string in D that is a proper suffix of u.

In order to solve the online prefix matching problem, we will move from a node u in the AC structure that corresponds to S_i to the node v that corresponds to S_{i+1}. To do this, the AC algorithm first tries to use a forward link from u with the character t_{i+1}. If no such forward link exists, then the algorithm recursively follows failure links until either no failure links are found (in which case v is the root of the trie) or until we reach a node that has a forward link with the character t_{i+1}. One can show that the cost per character of this process is $O(1)$ amortized time. Once v is found we use report links to report the current occurrences.

2.2 Succinct Representation of Trees

Representing ordinal trees. An *ordinal* tree $\mathcal{T}$ with n nodes is a rooted tree where the children of each node are ordered. Each node is given a unique id from $1, \ldots, n$. We use succinct representations of ordinal trees, where each node is given a unique id (the actual tree is not stored). The id is the rank of the node in the pre-order traversal of $\mathcal{T}$.

We use the Balanced Parentheses (BP) representation introduced by Jacobson [24]. In this representation we use parentheses to represent a pre-order traversal of the tree where the first time we visit a node is represented with an open parentheses and the last time we visit a node is represented with a close parentheses. This creates an array of $2n$ bits. For a node u let $open(u)$ and $close(u)$ denote the open and close parentheses of u.

Base set of operations. Munro and Raman [27] showed how to support the following operations in constant time using another $o(n)$ bits on top of the BP representation, for a

total of $2n + o(n)$ bits. By supporting this *base set* of operations on the BP representation one can also support many other common operations in constant time.

- *findclose*(l) – Given an index $l = open(u)$ for some node u, return $close(u)$.
- *findopen*(r) – Given an index $r = close(u)$ for some node u, return $open(u)$.
- *enclose*(i) – Return the pair of indices (l, r) such that: (1) l and r correspond to the same node, (2) $l \leq i \leq r$, and (3) $r - l$ is minimized.
- *pre_rank*(i)/*post_rank*(i) – Return the number of open/close parentheses in the the first i parentheses.
- *pre_select*(i)/*post_select*(i) – Return the index of the i'th open/close parenthesis.

It is important to notice that given an interval $[l, r]$ that corresponds to a node v, the id of v is *pre_rank*(l). Similarly, given the id i of v we have $l =$ *pre_select*(i) and $r =$ *findclose*(l). To simplify these operations we use the notion $v = node([l, r])$ and $[l, r] = interval(v)$. Our algorithms will also make use of the following two properties of the BP representation.

▶ Property 2.1. Let $\mathcal{T}$ be an ordinal tree. Let u be a node in $\mathcal{T}$ whose rank in the pre-order (post-order) on $\mathcal{T}$ is i (j). Then the open (close) parenthesis in the BP representation of $\mathcal{T}$ is i (j).

▶ Property 2.2. Let $\mathcal{T}$ be an ordinal tree and let $[a, b]$ and $[c, d]$ be two subintervals in the BP representation of $\mathcal{T}$ that correspond to two different nodes. Then either one subinterval is completely contained in the other or both subintervals are disjoint.

We emphasize that the $2n + o(n)$ bit representation of Geary, Raman and Raman [17] subsumes the representation of Munro and Raman [27], and in particular supports the base set of operations on the BP representation in constant time.

2.3 Labeled Trees and Multi-Labeled Trees

A *labeled tree* is an ordinal tree where each node has a label drawn from a set $\mathcal{L}$ of size $\lambda = |\mathcal{L}|$. A *multi-labeled tree* is an ordinal tree where each node is associated with a (possibly empty) subset of $\mathcal{L}$. For multi-labeled trees we denote the sum of the sizes of the label subsets by t. We assume without loss of generality that $\lambda \leq t$. Notice that the information-theoretic lower bound for representing a multi-labeled tree is $\log \binom{n\lambda}{t} + \log \binom{2n}{n} + o(n)$.

Our algorithms will make use of lowest labeled ancestor (LLA) queries on multi-labeled trees, where given a node id u and a label ℓ we can quickly return a node id v that is the lowest *proper* ancestor of u which has the label ℓ, or report that no such node exists. This operation is denoted by $v = LLA(\ell, u)$. For succinctness sake, from now on we refer to a node id as the node itself.

Representations supporting same label operations. In order to support fast LLA queries in succinct space we will make use of succinct representations of trees that allow us to compute in constant time some specific operations. These operations are on a label ℓ and a node u where u is also labeled by ℓ. The same label operations that we require are LLA, $pre\_rank_{\mathcal{T}}$ and $post\_rank_{\mathcal{T}}$ queries. We will also want to support $pre\_select_{\mathcal{T}}$ and $post\_select_{\mathcal{T}}$ queries in constant time. For sake of simplicity we refer to all of these operations as same label operations (although the select operations do not have any node as input). See Table 2 for the list of these operations.

In Section 4 we prove the following theorem.

▶ **Theorem 2.** *Assume there is a representation for a multi-labeled tree $\mathcal{T}$ using $f(\mathcal{T})$ bits that supports the same-label operations and the base set operations on the BP representation*

Table 2 Same label operations for multi-labeled trees.

Operation	Description
$LLA(\ell, u)$	The closest proper ancestor of u labeled by ℓ
$pre\_rank_{\mathcal{T}}(\ell, u)$	The rank of u (by the preorder of $\mathcal{T}$) in the set of nodes labeled by ℓ
$post\_rank_{\mathcal{T}}(\ell, u)$	The rank of u (by the postorder of $\mathcal{T}$) in the set of nodes labeled by ℓ
$pre\_select_{\mathcal{T}}(\ell, i)$	The i'th node with label ℓ in the preorder of $\mathcal{T}$
$post\_select_{\mathcal{T}}(\ell, i)$	The i'th node with label ℓ in the postorder of $\mathcal{T}$

in $O(1)$ time each. Then there exists a representation of $\mathcal{T}$ that for any $\lambda = \omega(1)$ uses $f(\mathcal{T}) + o(n+t)$ bits and answers any LLA query in $O(\log\log\lambda)$ time.

We are also able to represent any tree so it can support same-label LLA queries, as long as the label universe is an integer universe $\mathcal{L} = \{1, 2, \ldots, \lambda\}$. This is discussed in Section 5, where combined with Theorem 2 we prove the following theorem.

▶ **Theorem 3.** *For any multi-labeled tree $\mathcal{T}$ with a label set $\mathcal{L} = \{1, 2, \ldots, \lambda\}$ with $\lambda = \omega(1)$, there exists a representation of $\mathcal{T}$ that uses $\lceil \log \binom{n\lambda}{t} \rceil + 2(n+t+\lambda) + o(n+t+\lambda)$ bits and supports LLA queries in $O(\log\log\lambda)$ time.*

3 Dictionary Matching and Same Label Operations

The *c-extended prefix subset* of D, denoted by $P_c(D)$, is the subset of $P(D)$ which contains all $u \in P(D)$ such that $uc \in P(D)$ (the concatenation of u and c).

For each $u \in P(D)$ let u^R be the the string u in reverse order, and let $P(D)^R$ be the set of all reversed prefixes of D. The *suffix-lexicographic order* of $P(D)$ is an ordering of the elements in $P(D)$ where the order is determined by the lexicographic order of the corresponding elements in $P(D)^R$. Thus, for $u \in P(D)$, the rank of u in the suffix-lexicographic order of $P(D)$, denoted by $rank(u)$, is the lexicographic rank of u^R in $P(D)^R$. Since each prefix in $u \in P(D)$ has a unique node $state(u)$ in the AC data structure, let $rank(u)$ be the unique id of $state(u)$. Unless specified otherwise we will abuse notation and assume that $state(u) = rank(u)$.

Belazzougui's data structure. Belazzougui in [7] showed how one can leverage the suffix-lexicographic order of $P(D)$ in order to implement the AC data structure with $n(H_0(D) + 3.443 + o(1)) + O(d \log \frac{n}{d})$ bits. Our solution replaces only one particular component of Belazzougui's data structure which is called the *failure tree*, denoted by $\mathcal{T}_{fail}$. This tree is defined by the failure links in the AC data structure, so that for two nodes $state(u)$ and $state(v)$ we have $fail(state(u)) = state(v)$ if and only if $parent_{\mathcal{T}_{fail}}(state(u)) = state(v)$. An important property of $\mathcal{T}_{fail}$ is that the pre-order traversal of $\mathcal{T}_{fail}$ is exactly the suffix-lexicographic order of $P(D)$. Thus, Belazzougui's data structure uses succinct representations of ordinal trees for representing $\mathcal{T}_{fail}$ that support *parent* operations in constant time, thereby simulating the failure links.

3.1 Final-Failure Links

As discussed above, given some $S_i \in P(D)$ and $c \in \Sigma$ such that $S_i c \notin P(D)$ the time for finding S_{i+1} in the AC algorithm is $\Theta(n_{max})$. This expensive runtime occurs since the AC algorithm may traverse many failure links. However, the traversal stops when the algorithm

reaches a node for which there exists a forward link labeled by c. If such a node exists then this node is the final node in the traversal. We call this node the *final-failure* node for S_i and c, denoted by $ff(c, S_i)$. Notice that $ff(c, u) = state(v)$ where v is the longest suffix of u for which $v \in P_c(D)$. If no such node exists then we say that $ff(c, u) = \perp$. The key idea for improving the time cost per character of the AC algorithm is to find the final-failure node directly instead of traversing all of the failure links. We emphasize that the rest of Belazzougui's data structure remains the same. The only thing we change is the component for finding the final-failure.

In order to support locating the final-failure node we extend the definition of the failure tree. Instead of representing $\mathcal{T}_{fail}$ as an unlabeled ordinal tree, we represent $\mathcal{T}_{fail}$ as a multi-labeled tree. For each node $state(u) \in \mathcal{T}_{fail}$ we say that $state(u)$ is labeled by c if and only if $u \in P_c(D)$. Notice that a node may have many labels, or no labels at all (which is why we use a multi-labeled tree). Now the process of finding the final-failure node for $state(u)$ and character c reduces to finding $LLA(c, state(u))$ in the multi-labeled version of $\mathcal{T}_{fail}$.

Same label operations on $\mathcal{T}_{fail}$. We will now show how the properties of the AC structure and the implementations we consider allow us to support the same label operations in Table 2 on $\mathcal{T}_{fail}$ in constant time. This will allow us to use Theorem 2.

▶ **Lemma 4.** *There exists an implementation of $\mathcal{T}_{fail}$ that supports the parent operation, same-label operations and the base set operations on the BP representation in $O(1)$ time using $m(H_k(D) + 5 + o(1)) + 2\sigma + O(d \log \frac{n}{d})$ bits.*

Proof. Our implementation of $\mathcal{T}_{fail}$ contains two components. The first component is an implementation of the forward links of the AC data structure which is another part of the data structure of Belazzougui [7]. For $u \in P_c(D)$, the forward link from $state(u)$ with character $c \in \Sigma$ is implicitly represented by the ordered pair $(c, state(u))$. Using Belazzougui's implementation we can move from $(c, state(u))$ to $state(uc)$ or backwards in constant time.

The second component is a representation of a slightly modified version of $\mathcal{T}_{fail}$. A key observation with regard to the structure of $\mathcal{T}_{fail}$ is that for any child of the root of $\mathcal{T}_{fail}$, all of the nodes in the subtree of this child correspond to prefixes of the form uc for some $c \in \Sigma$ and $u \in P_c(D)$. However, it is possible that suffixes of the form uc are partitioned among several subtrees of children of the root. For purposes that will be clear later, it is helpful to have all of the nodes corresponding to prefixes ending with character c in one unique subtree of a child of the root. To support this, we add σ new dummy nodes, one for each character in Σ. These nodes will be the only children of the root. The i'th dummy has in its subtree all of the nodes of the form vi for each $P_i(D)$ (recall that $\Sigma = \{1, 2, \ldots, \sigma\}$). This is guaranteed by having each old child of the root become a child of the appropriate new dummy node. Notice that the pre-order and post-order of the nodes in $\mathcal{T}_{fail}$, excluding the dummy nodes, do not change with this modification. Rather, the i'th dummy node is inserted between the nodes corresponding to prefixes ending with $i - 1$ and the nodes corresponding to prefixes ending with i in the pre-order. Thus, for ui we have $pre\_rank_{\mathcal{T}_{fail}}(ui) = pre\_rank_{\mathcal{T}'_{fail}}(ui) - i$. Similarly, the i'th dummy node is inserted between the nodes corresponding to prefixes ending with i and the nodes corresponding to prefixes ending with $i + 1$ in the post-order. For the rest of this proof we refer to this slightly modified tree as $\mathcal{T}'_{fail}$. Notice that $\mathcal{T}'_{fail}$ has $m + \sigma$ nodes.

We represent $\mathcal{T}'_{fail}$ with the data structure of Geary, Raman and Raman [17] using $2m + 2\sigma + o(m)$ bits. Recall that this implementation supports the base set operations on the BP representation in constant time. The particular constant time operations we use with this representation on $\mathcal{T}'_{fail}$ are:

- $parent_{\mathcal{T}'_{fail}}(u) = parent(u)$: Given the id of a node $u \in \mathcal{T}'_{fail}$ return the id of the parent of u in $\mathcal{T}'_{fail}$.
- $child_{\mathcal{T}'_{fail}}(u, i) = child(u, i)$: Given the id of a node $u \in \mathcal{T}'_{fail}$ and a positive integer i, return the id of the i'th child of u in $\mathcal{T}'_{fail}$.
- $pre\_rank_{\mathcal{T}'_{fail}}(u) = pre\_rank(u)$: Given the id of a node $u \in \mathcal{T}'_{fail}$ return its location in the pre-order traversal of $\mathcal{T}'_{fail}$.
- $post\_rank_{\mathcal{T}'_{fail}}(u) = post\_rank(u)$: Given the id of a node $u \in \mathcal{T}'_{fail}$ return its location in the post-order traversal of $\mathcal{T}'_{fail}$.
- $pre\_select_{\mathcal{T}'_{fail}}(i) = pre\_select(i)$: Given an integer $1 \leq i \leq m + \sigma$ return the id of the i'th node in the pre-order traversal of $\mathcal{T}'_{fail}$.
- $post\_select_{\mathcal{T}'_{fail}}(i) = post\_select(i)$: Given an integer $1 \leq i \leq m + \sigma$ return the id of the i'th node in the post-order traversal of $\mathcal{T}'_{fail}$.

We use the parent operations on $\mathcal{T}'_{fail}$ to simulate parent operations on $\mathcal{T}_{fail}$ as follows. Due to the dummy nodes, when invoking the parent operation on u we check if the parent of u is a child of the root (by invoking another call to the parent operation), and if so we treat the root as the parent of u. Otherwise, the parent of u in $\mathcal{T}'_{fail}$ is also the parent of u in $\mathcal{T}_{fail}$.

Same label LLA. For $u, v \in P_c(D)$ we have that $LLA(c, state(u)) = state(v)$ if and only if $parent_{\mathcal{T}_{fail}}(state(uc)) = state(vc)$. This gives lead to supporting same label LLA queries in constant time. To do this, we first move from $state(u)$ to $state(uc)$ in constant time with the forward links structure, then we move from $state(uc)$ to $parent_{\mathcal{T}_{fail}}(state(uc)) = state(vc)$ in constant time using parent operations on $\mathcal{T}'_{fail}$, and then we move from $state(vc)$ to $state(v)$ using the forward links structure (going backwards) in constant time. The transition from $state(vc)$ to $state(v)$ is executed by first finding the pair $c, state(v)$ via a select operation on $state(v, c)$. This pair is represented using $\log m + \log \sigma$ bits. Extracting the $\log m$ bits representing $state(v)$ completes the transition.

Pre-order and post-order rank/select queries. We focus on the details for implementing $pre\_rank_{\mathcal{T}_{fail}}(c, u)$ for some $u \in P_c(D)$ as the rest of the operations are implemented using similar ideas (and the implementations are mostly technical). Recall that by definition, for $u \in P_c(D)$, $pre\_rank_{\mathcal{T}_{fail}}(c, u)$ is exactly the rank of uc in the pre-order $\mathcal{T}_{fail}$, minus $\sum_{c'<c} |P_{c'}(D)|$. Recall that $pre\_rank_{\mathcal{T}_{fail}}(uc) = pre\_rank_{\mathcal{T}'_{fail}}(uc) - c$, so the rank of u in the pre-order of $\mathcal{T}_{fail}$ among the nodes labeled by c can be computed in constant time by invoking $pre\_rank_{\mathcal{T}'_{fail}}(uc)$. Next, let $b = pre\_rank(child(r, c))$ where r is the root of $\mathcal{T}'_{fail}$. Since the c'th child of r is the dummy corresponding to c, then its rank in the pre-order of $\mathcal{T}'_{fail}$ is exactly $\sum_{c'<c} |P_{c'}(D)| + (c - 1)$. So we can compute $pre\_rank_{\mathcal{T}_{fail}}(c, u) = pre\_rank_{\mathcal{T}'_{fail}}(uc) - b$ in constant time.

Space usage. Our data structure uses the same space as Belazzougui's data structure, with the exception that instead of using $2m + o(m)$ bits for representing the failure tree, we use $2m + 2\sigma + o(m)$ bits via the representation of Geary, Raman and Raman [17] (which also supports base set of operations on the BP representation). Thus the total space used is $m(\log \sigma + \frac{2\sigma}{m} + 3.443 + o(1)) + O(d \log \frac{n}{d})$ bits. We further reduce the space usage using the technique of Hon et al. [22] to compress the forward links component into its *k'th order entropy*, thereby achieving a representation that uses $m(H_k(D) + 5 + o(1)) + 2\sigma + O(d \log \frac{n}{d})$ bits. ◀

3.2 Proof of Theorem 1

By combining Lemma 4 and Theorem 2 we obtain a succinct representation of $\mathcal{T}_{fail}$ which supports finding failure links in worst-case constant time and finding a final-failure in worst-case $O(\log\log\sigma)$ time, while using $m(H_k(D)+5+o(1))+2\sigma+O(d\log\frac{n}{d})$ bits.

For the text processing, each time a new character arrives we traverse at most $\log\log\sigma$ failure links. By Lemma 4, each such traversal takes constant time via a parent operation on $\mathcal{T}_{fail}$. If one of these links leads to the final failure, then we are done. Otherwise, we invoke the final failure procedure, which costs another $O(\log\log\sigma)$ time. Thus, the runtime is never worse than the runtime of the AC algorithm, and so the worst-case cost per character is $O(\log\log\sigma)$ (ignoring the cost of reporting the output) and the total cost for the entire text is $O(|T|+occ)$.

4 Solving General LLA With Same Label Operations

In this section we prove Theorem 2.

Successor Search. Recall that by the assumption of Theorem 2, the base set of operations on the BP representation of $\mathcal{T}$ are supported in constant time. For each label ℓ let $I_{\ell,open}$ and $I_{\ell,close}$ be the set of indices in the BP representation of the open and close parentheses, respectively, that correspond to nodes with label ℓ.

Let M be a subset of an ordered universe U. For an element $x\in U$ the successor of x in M is $succ_M(x)=\text{argmin}_{y\in M}\{y>x\}$. For sake of completeness we say that if $x>\max_{y\in M}\{y\}$ then $succ_M(x)=\infty$. In the following we show how successor operations on the sets $I_{\ell,open}$ and $I_{\ell,close}$ are used for answering LLA queries.

▶ **Lemma 5.** *Let $\mathcal{T}$ be a multi-labeled tree over label set $\mathcal{L}$. For a node $u\in\mathcal{T}$ and a label $\ell\in\mathcal{L}$ let $l=succ_{I_{\ell,open}}(close(u))$ and $r=succ_{I_{\ell,close}}(close(u))$. If $r<l$ then $LLA(\ell,u)=v$ where $v=node([findopen(r),r])$. If $r>l$ then $LLA(\ell,u)=LLA(\ell,w)$ where $w=node([l,findclose(l)])$. If $r=l$ then there is no node $LLA(\ell,u)$.*

Proof. Our proof has three cases. In the first case $r<l$, and so by Property 2.2 it must be that $open(u)>findopen(r)$. Therefore, the interval $[findopen(r),r]$ contains the interval $[open(u),close(u)]$ implying that v is an ancestor of u. Since v is labeled with ℓ and $r=close(v)=succ_{I_{\ell,close}}(close(u))$ there is no node on the internal path from v to u in $\mathcal{T}$ that is labeled with ℓ. Thus, $v=LLA(\ell,u)$.

In the second case $l<r$. We first show that $LLA(\ell,u)$ is necessarily an ancestor of $LLA(\ell,[l,findclose(l)])$ and then show that $LLA(\ell,[l,findclose(l)])$ is necessarily an ancestor of $LLA(\ell,u)$. Thus, the two must be the same.

Recall that the interval defined by $enclose(LLA(\ell,u))$ contains the interval $[open(u),close(u)]$. Moreover, since $l<r$ there is no closing parentheses of a node with label ℓ at the indices strictly between $close(u)$ and l. Therefore, the interval corresponding to $LLA(\ell,u)$ must contain the index l. Combining this with Property 2.2 it must be that the interval corresponding to $LLA(\ell,u)$ contains the interval $[open(u),findclose(l)]$ and so $LLA(\ell,u)$ is necessarily an ancestor of $LLA(\ell,[l,findclose(l)])$. For the other direction, by Property 2.2 the interval corresponding to $LLA(\ell,[l,findclose(l)])$ must contain the interval $[l,findclose(l)]$. Since there is no index in $I_{\ell,open}$ between $close(u)$ and l, the interval corresponding to $LLA(\ell,[l,findclose(l)])$ must contain the index $close(u)$. Combining with Property 2.2 the interval corresponding to $LLA(\ell,[l,findclose(l)])$ must contain the interval $[open(u),findclose(l)]$, and so $LLA(\ell,[l,findclose(l)])$ must be an ancestor of $LLA(\ell,u)$.

In the third case $r = l$. Then it must be that $r = l = \infty$ since otherwise we have a single index for both an open and close parentheses. Thus, there is no index of close parentheses in the range $[close(u) + 1, 2n]$ that corresponds to a node labeled with ℓ. If u has a proper ancestor v that is labeled with ℓ, then by Property 2.2 $close(v) > close(u)$. Therefore, there is no such ancestor, and $LLA(\ell, u)$ does not exist. ◀

By Lemma 5, once we perform two successor operations and a constant number of base set operations, we either find a node $v = LLA(\ell, u)$ or we find a node w that is labeled with ℓ such that $LLA(\ell, u) = LLA(\ell, w)$. Computing $LLA(\ell, w)$ in the second case takes $O(1)$ time since w is labeled with ℓ (and so this is a same label LLA query). What remains to be shown is how to execute the two successor queries on the sets of indices.

Successor queries on subsets of indices. Let R be a binary matrix of size $[a] \times [b]$. For integers $1 \leq x \leq a$ and $1 \leq y \leq b$ let $rank_{col}(y, x)$ be the number of 1s in the first x entries of the y'th column of R. For integers $1 \leq j \leq a$ and $1 \leq y \leq b$ let $select_{col}(y, j)$ be the index of the j'th 1 in the y'th column of R.

We focus on $I_{\ell,open}$ as the treatment of $I_{\ell,close}$ is the same. Consider the binary matrix R_{open} of size $[n] \times [\lambda]$, where $R_{open}[i][\ell] = 1$ if and only if $i \in I_{\ell,open}$. Given a node u in $\mathcal{T}$ we can find the row that corresponds to u in R_{open} in constant time by executing a single $pre\_rank(u)$ operation (which is a base set operation). Using the encoding of Barbay et al. [5] on R_{open} we can answer $rank_{col}$ and $select_{col}$ queries in $O(\log\log\lambda)$ and $O(1)$ time respectively. However, this encoding makes use of $O(t \log \lambda)$ bits (since there are t non zero values in the matrix). We reduce this space usage using indirection as follows.

Let $\tau = \log^2 \lambda$. For each set of indices $I_{\ell,open}$ let $\hat{I}_{\ell,open} \subset I_{\ell,open}$ be the indices whose rank in $I_{\ell,open}$ is a multiple of τ. Notice that if $|I_{\ell,open}| < \tau$ then $\hat{I}_{\ell,open} = \emptyset$. The treatment of such cases is discussed after explaining the more challenging case. Consider the binary matrix $\hat{R}_{open}$ of size $[n] \times [\lambda]$, where $\hat{R}_{open}[i][\ell] = 1$ if and only if $i \in \hat{I}_{\ell,open}$. Notice that the number of non-zero entries in $\hat{R}_{open}$ is $t' = O(\frac{t}{\tau})$. We further reduce the matrix $\hat{R}_{open}$ by removing all of the rows that have only zeros, and use another rank and select data structure to move between the row indices of these matrices. This uses another $t' \log \frac{n}{t'} + O(t') + o(n)$ bits [30], which is $o(n + t)$ bits[1].

Thus, we answer rank and select queries on the rows of $\hat{R}_{open}$ using the encoding of Barbay et al. [5] with $O(t' \log \lambda) = o(t)$ bits. Given an index i for which we wish to compute $s = succ_{I_{\ell,open}}(i)$ we first find $\hat{s} = succ_{\hat{I}_{\ell,open}}(i)$ using the data structure on $\hat{R}_{open}$ after finding the appropriate row in the matrix $\hat{R}_{open}$ with a single $rank_{col}$ operation in $O(\log\log\lambda)$ time. If we have successfully found $\hat{s}$ it must be that $|rank_{I_{\ell,open}}(s) - rank_{I_{\ell,open}}(\hat{s})| \leq \tau$. Thus, with $O(\log\tau)$ executions of $pre\_select(\ell, i)$ (each costing $O(1)$ time since it is a same label operation), we perform a binary search to find s in $O(\log\tau)$ time.

Finally, if we were not successful in finding $\hat{s}$ (either $\hat{I}_{\ell,open} = \emptyset$ or $i \geq \max \hat{I}_{\ell,open}$) then there are at most τ possible elements to consider (the last τ elements) and a binary or exponential search with $pre\_select(\ell, i)$ operations finds s in $O(\log\tau)$ time. This completes the proof of Theorem 2

[1] If $t \leq n$ then let $t = n/x$ for some x. Then $t' \log \frac{n}{t'} = \frac{n}{x\tau} \log x\tau = o(n)$. If $t > n$ then $t' \log \frac{n}{t'} = \frac{t}{\tau} \log \frac{n\tau}{t} \leq \frac{t}{\tau} \log \tau = o(t)$.

5 Same Label Operations for General Multi-labeled Trees

In this section we prove Theorem 3. The representation of $\mathcal{T}$ uses two main components. The first component is a *label-ordered tree*, which functions in a way that is similar to the modified failure tree in the dictionary matching data structure. The second component is an implementation of a *transition operator* which functions in a way that is similar to the forward links in the dictionary matching data structure.

Label-ordered tree. The encoding technique for the label-ordered tree is similar to the *tree extraction* technique used in [18, 19, 20]. For each $\ell \in \mathcal{L}$ let $\mathcal{F}_\ell$ be the induced forest obtained by inducing $\mathcal{T}$ on the nodes with label ℓ. By an inducing we mean that for two nodes $u, v \in \mathcal{F}_\ell$, u is the parent of v if and only if both u and v are labeled with ℓ, and $u = LLA(\ell, v)$. Notice that the sum of the sizes of all of the forests is exactly t. For each such forest we create a dummy node and make it the parent of all of the roots of trees in the forest. This adds another λ nodes. Finally, we add a special new root whose children are the dummy nodes, ordered by their labels, thereby creating the label-ordered tree. We denote this tree by $\hat{\mathcal{T}}$. The size of $\hat{\mathcal{T}}$ is $t + \lambda + 1$. We use the data structure of Geary, Raman and Raman [17] to represent $\hat{\mathcal{T}}$ with $2t + 2\lambda + o(t + \lambda)$ bits, while supporting the base set of operations on the BP representation of $\hat{\mathcal{T}}$ in constant time.

Transition operator. The transition operator translates in constant time between the rank of a node u in $\mathcal{T}$ and a label ℓ, and the rank of the copy of u in $\hat{\mathcal{T}}$ which is associated with ℓ. This translation works in both directions. To do so, for each $u \in \mathcal{T}$ and for each label ℓ of u, the transition operator creates the ordered pair $(\ell, pre\_rank_{\mathcal{T}}(u))$. Notice that $rank((\ell, pre\_rank_{\mathcal{T}}(u))) = pre\_rank_{\hat{\mathcal{T}}}(u\ell) - \ell - 1$, where the rank is taken over all ordered pairs. Similarly, one can use a select query to translate from $pre\_rank_{\hat{\mathcal{T}}}(u\ell)$ to $pre\_rank_{\mathcal{T}}(u)$. We use a data structure that supports rank and select queries in constant time [30] using $\lceil \log \binom{n\lambda}{t} \rceil + o(t) + O(\log \log (n\lambda))$ bits.

Same label LLA. Using the above representations, we support same label *LLA* queries exactly like we do in the proof of Lemma 4. Since we used a representation for supporting the base set of operations on the BP representation of $\hat{\mathcal{T}}$ in constant time, again using the ideas in the proof of Lemma 4 we support same label operations and the base set of operations on the BP representation of $\mathcal{T}$ in constant time. Thus, together with Theorem 2 we have completed the proof of Theorem 3.

References

1. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.
2. Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved dynamic dictionary matching. *Inf. Comput.*, 119(2):258–282, 1995.
3. Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000.
4. Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the gap. *CoRR*, abs/1503.07563, 2015.

5 Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theor. Comput. Sci.*, 387(3):284–297, 2007.

6 Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Trans. on Algorithms*, 7(4):52, 2011.

7 Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Combinatorial Pattern Matching, CPM*, pages 88–100, 2010.

8 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014.

9 Anat Bremler-Barr, David Hay, and Yaron Koral. Compactdfa: Generic state machine compression for scalable pattern matching. In *INFOCOM*, pages 659–667. IEEE, 2010.

10 Anat Bremler-Barr, David Hay, and Yaron Koral. Compactdfa: Scalable pattern matching using longest prefix match solutions. *IEEE/ACM Trans. Netw.*, 22(2):415–428, 2014.

11 Gerth Stølting Brodal and Leszek Gasieniec. Approximate dictionary queries. In *Combinatorial Pattern Matching, CPM*, pages 65–74, 1996.

12 Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *Symposium on Discrete Algorithms, (SODA)*, pages 13–22, 2005.

13 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *Europ. Symp. Algorithms, (ESA)*, pages 361–372, 2015.

14 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.

15 Guy Feigenblat, Ely Porat, and Ariel Shiftan. An improved query time for succinct dynamic dictionary matching. In *Combinatorial Pattern Matching, CPM*, pages 120–129, 2014.

16 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

17 Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

18 Meng He, J. Ian Munro, and Gelin Zhou. Path queries in weighted trees. In *International Symposium on Algorithms and Computation, (ISAAC)*, pages 140–149, 2011.

19 Meng He, J. Ian Munro, and Gelin Zhou. Succinct data structures for path queries. In *European Symposium on Algorithms, (ESA)*, pages 575–586, 2012.

20 Meng He, J. Ian Munro, and Gelin Zhou. A framework for succinct labeled ordinal trees over large alphabets. *Algorithmica*, 70(4):696–717, 2014.

21 Wing-Kai Hon, Tsung-Han Ku, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressing dictionary matching index via sparsification technique. *Algorithmica*, 72(2):515–538, 2015.

22 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster compressed dictionary matching. In *String Processing and Information Retrieval, (SPIRE)*, pages 191–200, 2010.

23 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Compressed index for dictionary matching. In *Data Compression Conference (DCC)*, pages 23–32, 2008.

24 Guy Jacobson. Space-efficient static trees and graphs. In *Symposium on Foundations of Computer Science, (FOCS)*, pages 549–554, 1989.

25 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

26 J. Ian Munro. Tables. In Vijay Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.

27 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

28 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

29 Shoshana Neuburger and Dina Sokol. Succinct 2d dictionary matching with no slowdown. In *Algorithms and Data Structures Symposium, (WADS)*, pages 619–630, 2011.

30 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

31 Dekel Tsur. Succinct representation of labeled trees. *TCS*, 562:320–329, 2015.

32 Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *INFOCOM*, 2004.

Graph Motif Problems Parameterized by Dual

Guillaume Fertin[1] and Christian Komusiewicz*[2]

1 **Laboratoire d'Informatique de Nantes-Atlantique, UMR CNRS 6241, Université de Nantes, 2 rue de la Houssinière, 44322 Nantes Cedex 3, France**
guillaume.fertin@univ-nantes.fr

2 **Institut für Informatik, Friedrich-Schiller-Universität Jena, Germany**
christian.komusiewicz@uni-jena.de

Abstract

Let $G = (V, E)$ be a vertex-colored graph, where C is the set of colors used to color V. The Graph Motif (or GM) problem takes as input G, a multiset M of colors built from C, and asks whether there is a subset $S \subseteq V$ such that (i) $G[S]$ is connected and (ii) the multiset of colors obtained from S equals M. The Colorful Graph Motif (or CGM) problem is the special case of GM in which M is a set, and the List-Colored Graph Motif (or LGM) problem is the extension of GM in which each vertex v of V may choose its color from a list $\mathcal{L}(v)$ of colors.

We study the three problems GM, CGM, and LGM, parameterized by $\ell := |V| - |M|$. In particular, for general graphs, we show that, assuming the strong exponential time hypothesis, CGM has no $(2-\epsilon)^{\ell} \cdot |V|^{O(1)}$-time algorithm, which implies that a previous algorithm, running in $O(2^{\ell} \cdot |E|)$ time is optimal [2]. We also prove that LGM is W[1]-hard even if we restrict ourselves to lists of at most two colors. If we constrain the input graph to be a tree, then we show that GM can be solved in $O(4^{\ell} \cdot |V|)$ time but admits no polynomial-size problem kernel, while CGM can be solved in $O(\sqrt{2}^{\ell} + |V|)$ time and admits a polynomial-size problem kernel.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics, G.2.2 Graph Theory

Keywords and phrases NP-hard problem, subgraph problem, fixed-parameter algorithm, lower bounds, kernelization

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.7

1 Introduction

The Subgraph Isomorphism problem is the following pattern matching problem in graphs: given a (typically large) host graph G and a (small) query graph H, return one (or all) occurrence(s) of H in G, where the term occurrence denotes here a subset S of $V(G)$ such that $G[S]$, the subgraph of G induced by S, is isomorphic to H. This type of graph mining problem has numerous applications, notably in biology [20]. Subgraph Isomorphism is a *structural* graph pattern matching problem, where one looks for similar graph structures between H and G. In some biological contexts, however, additional information is provided to the vertices of the graphs, for example their biological function. This can be modeled by labeling each vertex of the graph, for example by giving it one or several colors, each corresponding to an identified function. In the presence of such functional annotation, the structure of a given induced subgraph may be of less importance than the functions it

* Christian Komusiewicz was supported by the DFG, project "Multivariate algorithmics for graph and string problems in bioinformatics" (KO 3669/4-1).

corresponds to. Thus, a new set of *functional* graph pattern matching problems has emerged, starting with the Graph Motif problem [15], which was introduced in the context of the analysis of metabolic networks. Here, what is primarily sought in the host graph is a multiset M of colors that represents the functions of interest.

Graph Motif (GM)
Input: A multiset M built on a set C of colors, an undirected graph $G = (V, E)$, and a coloring $\chi : V \to C$.
Question: Is there a set $S \subseteq V$ such that $G[S]$ is connected and there is a one-to-one mapping f from S to M?

Many variants of the GM problem have been introduced and studied. In particular, List-Colored Graph Motif (or LGM) is a generalization of GM that is used to identify protein complexes in protein interaction networks that are similar to a given protein complex from a different species [7]. In LGM, a list-coloring $\mathcal{L} : V \to 2^C$ is assigned to each vertex of G, and the question asked is the existence of $S \subseteq V$ such that (i) $G[S]$ is connected and (ii) the one-to-one mapping f from S to M we look for satisfies $\forall v \in S : f(v) \in \mathcal{L}(v)$. The special case of GM in which M is a set is called Colorful Graph Motif (or CGM). Many optimization problems related to GM have received interest, including some that are related to tandem mass spectrometry and where the input graph is directed [19]. All these problem variants have given rise to a very abundant literature. CGM, GM, and LGM are NP-hard even in very restricted cases [10]. Consequently, many of the above-mentioned studies have focused on (dis)proving fixed-parameter tractability of the problems (see e.g. [21] for the most recent survey on the topic). In such cases, very often the parameter $k := |M| = |S|$ is considered.

In this paper, we study the parameterized complexity of GM, CGM, and LGM, but we differ from the usual viewpoint by focusing on the *dual* parameter $\ell := |V| - |S|$, that is, ℓ is the number of vertices to be deleted from G to obtain a solution. Although the choice of ℓ may be disputable because it may *a priori* be too large to expect a good behavior in practice, there are several arguments for choosing such a parameter: First, after some initial data reduction, the input may be divided into smaller connected components, where ℓ is not much larger than k. Second, the algorithms for parameter k rely on algebraic techniques or dynamic programming, and in both cases, the worst-case running time is equivalent to the actual running time. In contrast, for example for CGM, the algorithm for parameter ℓ is a search tree algorithm [2], and search tree algorithms can be accelerated substantially via pruning rules. Finally, there are subgraph mining problems where the dual parameter ℓ is usually bigger than the parameter k but leads to the current-best algorithm (in terms of performance on real-world instances) [13]. Hence, parameterization by ℓ may be useful even if ℓ is bigger than k, and thus deserves to be studied.

Related work and our contribution. GM is NP-hard, even when M is composed of two colors [10]. Concerning the parameterized complexity for parameter $k := |M|$, the current-best randomized algorithm has a running time of $2^k \cdot n^{O(1)}$ [3, 18] where $n := |V|$, and there is some evidence that this cannot be improved to a running time of $(2 - \epsilon)^k \cdot n^{O(1)}$ [3]. The current-best running time for a deterministic algorithm is $5.22^k \cdot n^{O(1)}$ [17]. GM on trees can be solved in $n^{O(c)}$ time where c is the number of colors in M [10], but it is W[1]-hard with respect to c [10]. Other parameters, essentially related to the structure of the input graph G, have been studied by Bonnet and Sikora [6]. Finally, concerning parameter ℓ, GM has been shown to be W[1]-hard, even when M is composed of two colors [2].

Table 1 Overview of new and previous results with respect to the dual parameter $\ell := n - k$, where $n := |V|$, $m := |E|$, $k := |M|$, and $\Delta := \max_{v \in V} |\mathcal{L}(v)|$ denotes the maximum list size in G. The lower bound result for CGM assumes the strong exponential time hypothesis (SETH) [14].

	General graphs	Trees
LGM	W[1]-hard [2]	?
LGM, $\Delta = 2$	W[1]-hard (Cor. 4)	?
GM	W[1]-hard [2]	$O(4^\ell \cdot n)$ (Thm. 5) no poly. kernel (Thm. 8)
CGM	$O(2^\ell \cdot m)$ [2], no $(2-\epsilon)^\ell \cdot n^{O(1)}$ (Thm. 1) no poly. kernel (Thm. 2)	$O(\sqrt{2}^\ell + n)$ (Thm. 13), $(2\ell+1)$-vertex kernel (Thm. 10)

Since CGM is a special case of GM, any above-mentioned positive result for GM also holds for CGM. Besides, CGM is NP-hard, even for trees of maximum degree 3 [10], and does not admit a polynomial-size problem kernel with respect to k even if G has diameter two or if G is a comb graph (a special type of tree with maximum degree 3) [1]. Finally, CGM can be solved in $O(2^\ell \cdot m)$ time [2]. The LGM problem is an extension of GM and thus any negative result for GM propagates to LGM. Moreover, LGM is known to be fixed-parameter tractable with respect to k, the current-best algorithm runs in $2^k \cdot n^{O(1)}$ time [18]. Concerning parameter ℓ, LGM has been shown to be W[1]-hard even when M is a set [2].

As mentioned above, we study GM, LGM and CGM with respect to the dual parameter $\ell := n - k$. Since many results in general graphs turn out to be negative, we also chose to focus on the special case where the input graph G is a tree. Our results are summarized in Table 1. In a nutshell, we strengthen previous hardness results for the general case and show that the $O(2^\ell \cdot m)$-time algorithm for CGM is essentially optimal. Then, we show that for GM on trees a fixed-parameter algorithm can be achieved, and that, for CGM on trees, a polynomial problem kernel and better running times than for general graphs can be achieved.

Preliminaries. Throughout the paper, the input graph for our three problems is $G = (V, E)$, and we let $n := |V|$ (resp. $m := |E|$) denote its number of vertices (resp. edges). We use $[n] := \{1, \ldots, n\}$ to denote the set of the integers from 1 through n. The set S of vertices sought for in the three problems is called an *occurrence* of M. If G is vertex-colored, we call a vertex set S *colorful* if $|S| = |M|$ and all vertices in S have pairwise different colors. A vertex v is called *unique* if it is assigned a color c that is assigned to no other vertex in V.

We briefly recall the relevant notions of parameterized algorithmics [8]. A *reduction to a problem kernel*, or *kernelization*, is an algorithm that takes as input an instance (I, k) of a parameterized problem and produces in polynomial time an equivalent instance (I', k') (that is, having the same solution) such that (i) $|I'| \leq g(k)$, and (ii) $k' \leq k$. The instance (I', k') is called *problem kernel* and g is called the *size of the problem kernel*. If g is a polynomial function, then the problem admits a *polynomial-size problem kernelization*. The class W[1] is a basic class of presumed fixed-parameter intractability [8], that is, if a problem is W[1]-hard for parameter k, then we assume that it cannot be solved in $f(k) \cdot n^{O(1)}$ time [8]. The strong exponential time hypothesis (SETH) assumes that CNF-SAT with n variables cannot be solved in time $(2-\epsilon)^n$ for any $\epsilon > 0$ [14].

This work is structured as follows. In Section 2, we present lower bounds for LGM and CGM on general graphs. These negative results motivate our study of the case when G

is a tree; our results for GM on trees and CGM on trees will be presented in Section 3 and Section 4, respectively. Due to lack of space, some proofs are deferred to a full version of the article.

2 Parameterization by Dual in General Graphs: Tight Lower Bounds

CGM can be solved in $O(2^\ell \cdot m)$ time [2]. We show here that this running time bound is essentially optimal.

▶ **Theorem 1.** Colorful Graph Motif *cannot be solved in* $(2-\epsilon)^\ell \cdot n^{O(1)}$ *time unless the strong exponential time hypothesis fails.*

Proof. We present a polynomial-time reduction from CNF-SAT:

> **Input:** A boolean formula Φ in conjunctive normal form with clauses $\mathcal{C}_1, \ldots, \mathcal{C}_q$ over variable set $X = \{x_1, \ldots, x_r\}$.
> **Question:** Is there an assignment β to X that satisfies Φ?

The reduction works as follows. First, for each variable $x_i \in X$ introduce two *variable vertices* v_i^t and v_i^f and color each of the two vertices with color χ_i^x. The idea is that (with the final occurrence) we must select exactly one vertex for this color. This selection will correspond to a truth assignment to X. Now, introduce for each clause $\mathcal{C}_i$ a *clause vertex* u_i, color u_i with a unique color $\chi_i^{\mathcal{C}}$ and make u_i adjacent to vertex v_j^t if x_j occurs nonnegated in $\mathcal{C}_i$ and to vertex v_j^f if x_j occurs negated in $\mathcal{C}_i$. Finally, introduce one further vertex v^* with a unique color χ^*, make v^* adjacent to all variable vertices and let M be the set containing each of the introduced colors exactly once. Note that there are exactly $|X|$ colors that appear twice in G and that all other colors appear exactly once. Hence, $\ell = |X|$. We next show the correctness of the reduction. Let I denote the constructed instance of CGM.

First, assume that Φ is satisfiable and let β be a satisfying assignment of X. For the CGM instance consider the vertex set $S \subseteq V$ that contains all clause vertices, vertex v^*, and for each variable x_i the vertex v_i^t if β sets x_i to 'true' and v_i^f otherwise. Clearly, $|S| = |M|$ and no two vertices of S have the same color. To show that I is a yes-instance of CGM it remains to show that $G[S]$ is connected. First, the subgraph induced by the variable vertices in S plus v^* is a star and thus it is connected. Second, since β is a satisfying assignment each clause vertex in S has at least one neighbor in S (which is by construction a variable vertex). Hence, $G[S]$ is connected.

Conversely, assume that I is a yes-instance of CGM, and let S be a colorful vertex set with $|S| = |M|$ such that $G[S]$ is connected. Since S is colorful, the variable vertices in S correspond to a truth assignment of X. This assignment satisfies X: Indeed, since $G[S]$ is connected, there is a path in $G[S]$ between each clause vertex u_i and v^*, and thus there is a neighbor of u_i that is in S. If this neighbor is v_j^t (resp. v_j^f), then by construction, β assigns 'true' (resp. 'false') to x_j and thus $\mathcal{C}_i$ is satisfied.

Thus, the two instances are equivalent. Now observe that since $\ell = |X| = r$ and $n = 2r+q+1$, any $(2-\epsilon)^\ell \cdot n^{O(1)}$-time algorithm implies a $(2-\epsilon)^r \cdot (r+q)^{O(1)}$-time algorithm for CNF-SAT. This directly contradicts the SETH. ◀

The above reduction also makes the existence of a polynomial-size problem kernel for parameter ℓ unlikely. This is implied by the following two facts. First, CNF-SAT parameterized by the number of variables does not admit a polynomial-size problem kernel unless NP $\subseteq$ coNP/poly [9]. Second, the reduction presented in proof of Theorem 1 is a *polynomial parameter transformation* [5] from CNF-SAT parameterized by the number of

variables to CGM parameterized by ℓ. More precisely, given an input CNF-SAT formula Φ on variable set X, the reduction produces an instance $I = (M, G, \chi)$ of CGM with $\ell = |X|$. Now, any polynomial-size problem kernelization applied to I produces in polynomial time an equivalent CGM instance I' of size $\ell^{O(1)} = |X|^{O(1)}$. Since CNF-SAT is NP-hard, we can now transform this CGM instance in polynomial time into an equivalent CNF-SAT instance that has size $\ell^{O(1)} = |X|^{O(1)}$. Hence, a polynomial-size problem kernel for CGM parameterized by ℓ implies a polynomial-size problem kernel for CNF-SAT parameterized by $|X|$. This implies NP $\subseteq$ coNP/poly [9] (which in turn implies a collapse of the polynomial hierarchy).

▶ **Theorem 2.** Colorful Graph Motif *parameterized by ℓ does not admit a polynomial-size problem kernel unless NP $\subseteq$ coNP/poly.*

We have thus resolved the parameterized complexity of CGM parameterized by ℓ on general graphs and now turn to the more general LGM which is W[1]-hard with respect to ℓ [2]. Here, it would be desirable to obtain fixed-parameter algorithms for the parameter ℓ at least for some restricted inputs. In other words, we would like to further exploit the structure of real-world instances to obtain tractability results. A very natural approach here is to consider the size and structure of the list-colorings $\mathcal{L}(v)$ as additional parameter. Unfortunately, the problem remains W[1]-hard even for the following very restricted case of list-colorings. Herein, the vertex-color graph is the graph with vertex set $V \cup C$ and edge set $\{\{v, c\} \mid c \in \mathcal{L}(v)\}$.

▶ **Theorem 3.** List-Colored Graph Motif *is W[1]-hard with respect to ℓ even if the vertex-color graph is a disjoint union of paths.*

We immediately obtain the following.

▶ **Corollary 4.** List-Colored Graph Motif *is W[1]-hard with respect to ℓ even if $|\mathcal{L}(v)| \leq 2$ for every vertex in G.*

3 Graph Motif on Trees

Motivated by these negative results on general graphs, we now study the special case where the input graph is a tree. For LGM, we were not able to resolve the parameterized complexity with respect to ℓ for this case. Hence, we focus on the more restricted GM problem. We show that GM is fixed-parameter tractable with respect to ℓ if the input graph is a tree. Recall that for general graphs, GM is W[1]-hard for ℓ even if the motif M contains only two colors [2]. Hence, the tree structure helps significantly when parameterizing by ℓ.

3.1 A Dynamic Programming Algorithm

Call a color of M *abundant* if it occurs more often in G than in M. The abundant colors are exactly the ones that have to be "deleted" to obtain a solution S. Let $c_1, \ldots, c_j$ denote the abundant colors of M, and let ℓ_i denote the difference between the number of vertices in V that have color c_i and the multiplicity of c_i in M. This implies in particular that $\sum_{1 \leq i \leq j} \ell_i = \ell$.

The algorithm is a dynamic programming algorithm that works on a rooted representation of G. Thus, obtain a rooted tree T by rooting G at an arbitrary vertex $r \in V$. As usual for dynamic programming on trees, the idea is to combine partial solutions of subtrees. Our algorithm is somewhat similar to a previous dynamic programming algorithm for GM on graphs of bounded treewidth [10] but the analysis and concrete table setup is different.

▶ **Theorem 5.** Graph Motif *can be solved in $O(4^\ell \cdot n)$ time if G is a tree.*

The fixed-parameter tractability of GM on trees also implies the following result for LGM.

▶ **Corollary 6.** LGM *can be solved in* $O(4^\ell \cdot n)$ *time if* G *is a tree and the vertex-color graph* $H = (V \cup C, \{\{v, c\} \mid c \in \mathcal{L}(v)\})$ *is a disjoint union of paths.*

Proof. We describe a reduction of this special case of LGM on trees to GM on trees. Here, we call the vertices of H that are from V the V-vertices of H and those that are from C the C-vertices. Observe that without loss of generality, we can assume that all colors in the lists are contained in M. First, if H has a connected component that contains more C-vertices than V-vertices, then the instance $(M, G, \mathcal{L})$ is a no-instance and can be immediately rejected. Second, for any connected component H' of H that contains at least two C-vertices c_1 and c_2 that have multiplicity two in M, then the instance is also a no-instance: In H', the number of V-vertices exceeds the number of C-vertices by at most one. Hence, if four or more V-vertices are assigned only to c_1 or c_2, then there is some other C-vertex in H' that is assigned to none of the V-vertices. A similar argument applies if H' contains a C-vertex that has multiplicity at least three in M.

If the instances are not rejected because any of the cases described above applies, then each connected component H' of H has at most one C-vertex that has multiplicity two in M and all other C-vertices have multiplicity at most one. We show that in both cases, the constraints of $\mathcal{L}$ for H' can be replaced by simple coloring constraints.

Case 1: Every C-vertex of H' has multiplicity one in M. If H' has the same number of V-vertices as C-vertices (equivalently, H' has an even number of vertices), then every occurrence S of M contains all V-vertices from H'. Otherwise, if H' has more V-vertices than C-vertices (equivalently, H' has an odd number of vertices), then every occurrence S of M contains all except one V-vertex from H'. In both cases, we can replace the constraints as follows. Introduce a color $c_{H'}$, color all V-vertices in H' with color $c_{H'}$ and replace in M every C-vertex of H' by $c_{H'}$. In the first case, the number of vertices with color $c_{H'}$ is exactly the multiplicity of $c_{H'}$ in M, in the second case it is the multiplicity of $c_{H'}$ in M plus one.

Case 2: One C-vertex c of H' has multiplicity two in M. If H' has the same number of V-vertices as C-vertices (equivalently, H' has an even number of vertices), then the instance is a no-instance and can be rejected immediately: any assignment of colors to the V-vertices either fails to assign one of the C-vertices or assigns at most one V-vertex to c. Otherwise, if H' has an odd number of vertices, every occurrence S of M contains all V-vertices of H'. The constraints posed by H' may thus be replaced as follows: Introduce a color $c_{H'}$, color all V-vertices in H' with color $c_{H'}$ and replace in M every C-vertex of H' by $c_{H'}$ (replace c twice). Then the multiplicity of $c_{H'}$ in M is exactly the number of V-vertices in H'.

Applying these replacements exhaustively then results in an equivalent instance of GM on trees which can be solved in the claimed running time due to Theorem 5. ◀

3.2 A Kernelization Lower Bound

We now show that GM does not admit a polynomial-size problem kernel with respect to ℓ even if G is a tree. The proof is based on a cross-composition [4] from the W[1]-hard Multicolored Clique problem [11].

Multicolored Clique
Input: A graph $H = (W, F)$ and a vertex-coloring $\chi : W \to \{1, \ldots, k\}$.
Question: Is there a vertex set $S \subseteq W$ such that S is colorful, that is, $|S| = k$ and the vertices in S have pairwise different colors, and $H[S]$ is a clique?

To avoid confusion between the colors of the Multicolored Clique instance and the GM instance, we refer to the colors of the Multicolored Clique instance as *labels* in the following. Informally, cross-compositions are reductions that combine many instances of one problem into one instance of another problem. The existence of a cross-composition from an NP-hard problem to a parameterized problem Q implies that Q does not admit a polynomial-size problem kernel (unless NP $\subseteq$ coNP/poly) [4].

▶ **Definition 7** ([4]). Let $L \subseteq \Sigma^*$ be a language, let R be a polynomial equivalence relation on Σ^*, and let $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem. An *or-cross-composition of L into Q* (with respect to R) is an algorithm that, given t instances $x_1, x_2, \ldots, x_t \in \Sigma^*$ of L belonging to the same equivalence class of R, takes time polynomial in $\sum_{i=1}^{t} |x_i| + k$ and outputs an instance $(y, k) \in \Sigma^* \times \mathbb{N}$ of Q such that

- the parameter value k is polynomially bounded in $\max_{i=1}^{t} |x_i| + \log t$, and
- the instance (y, k) is a yes-instance for Q if and only if at least one instance x_i is a yes-instance for L .

We present an or-cross composition of Multicolored Clique into GM on trees parameterized by ℓ. The polynomial equivalence relation R will be simply to assume that all the Multicolored Clique instances have the same number of vertices n. The main trick is to encode vertex identities in the graph of the Multicolored Clique instance by numbers of colored vertices in the GM instance; note that this approach was also followed in previous works on GM [10, 6]. Given t instances $(H_1 = (W_1, F_1), \chi_1), H_2 = (W_2, F_2), \chi_2), \ldots, H_t = (W_t, F_t), \chi_t)$ of Multicolored Clique such that $|W_i| = n$ for all $i \in [t]$, we reduce to an instance of GM where the input graph is a tree as follows. Herein, we assume without loss of generality that $t = 2^s$ for some integer s.

The first construction step is to add one vertex r that connects the different parts of the instance and which will be contained in every occurrence of the motif. The vertex r thus receives a unique color that may not be deleted. To this vertex r we attach subtrees corresponding to edges of the input instances. Deleting vertices of such a subtree then corresponds to selecting the endpoints of the corresponding edge.

Instance selection gadget. The technical difficulty in the construction is to ensure that the solution deletes only vertices in subtrees corresponding to edges of the same graph. To achieve this, we introduce $k \cdot (k-1) \cdot \log t$ instance selection colors $\iota[p, q, \tau]$ where $p \in [k]$, $q \in [k] \setminus \{p\}$, and $\tau \in [\log t]$, and demand that the solution deletes exactly one vertex of each instance selection color. To ensure that exactly one instance is selected, we use two further colors ι^+ and ι^-. For each Multicolored Clique instance (H_i, χ_i), attach an *instance selection path* P_i to r that is constructed based on the number i. Let $b(i)$ denote the binary expansion of i and let $b_\tau(i)$, $\tau \in [\log t]$, denote the τth digit of $b(i)$. Construct a path P_i containing first a vertex with color ι^+, then in arbitrary order exactly one vertex of each color in the color set $I_i := \{\iota[p, q, \tau] : b_\tau(i) = 1\}$, and then a vertex with color ι^-. Attach the path P_i to r by making the vertex with color ι^+ a neighbor of r.

The idea of the construction is that exactly one instance selection path P_i is deleted completely and that this will force any solution to delete paths that "complement" P_i (that is, paths which contain all $\iota[p, q, \tau]$ such that $b_\tau(i) = 0$) in the rest of the graph.

Edge selection gadget. To force deletion of subtrees corresponding to exactly $\binom{k}{2}$ edges with different labels, we introduce $2k(k-1)$ label selection colors $\lambda[p, q]^+$ and $\lambda[p, q]^-$ where $p \in [k]$ and $q \in [k] \setminus \{p\}$. These colors will ensure that for each pair of labels p and q the solution deletes exactly one path corresponding to the ordered pair (p, q) and one path corresponding to the pair (q, p).

There are two further sets of colors. One set is used for ensuring vertex consistency of the chosen edges, that is, to make sure that all the selected edges with label pair $(p, \cdot)$ correspond to the same vertex with label p. More precisely, we introduce a color $\omega[p, q]$ for each $p \in [k]$ and each $q \in [k] \setminus \{p\}$, except for the biggest $q \in [k] \setminus \{p\}$.

The final color set is used to check that the edges selected for label pair (p, q) and for label pair (q, p) are the same. To this end, we introduce a set of colors $\varepsilon[p, q]$ for each $p \in [k]$ and each $q \in [k] \setminus \{p\}$ such that $q > p$. To perform the checks of vertex and edge consistency, we encode the identities of vertices and edges into path lengths. More precisely, we assign each vertex $v \in W_i$ a unique (with respect to the vertices of W_i) number $\#(v) \in [n]$.

Now, for each label pair (p, q) and each instance i, attach one path $P_i(u, v)$ to r for each edge $\{u, v\}$ where u has color p and v has color $q \neq p$. The path $P_i(u, v)$

- starts with a vertex with color $\lambda[p, q]^+$ that is made adjacent to r,
- then contains exactly one vertex of each color in $\{\iota[p, q, \tau] : \iota[p, q, \tau] \notin I_i\}$,
- then contains $\#(u)$ vertices of color $\varepsilon[p, q]$ if $p < q$ and $n - \#(v)$ vertices of color $\varepsilon[q, p]$ if $p > q$,
- then, if q is not the biggest label in $[k] \setminus p$, contains $\#(u)$ vertices with color $\omega[p, q]$,
- then, if q is not the smallest label in $[k] \setminus p$, contains $n - \#(u)$ vertices with color $\omega[p, q']$, where q' is the next-smaller label in $[k] \setminus p$ (if $p = q-1$, then $q' = q-2$; otherwise $q' = q-1$), and
- ends with a vertex with color $\lambda[p, q]^-$.

Let $\mathcal{C}$ denote the multiset containing all the vertex colors of all vertices added during the construction with their respective multiplicities. In the correctness proof it will be easier to argue about the colors that are not contained in M. Hence, the construction is completed by setting the multiset D of colors to "delete" to contain each color with multiplicity one except

- the color of r which is not contained in D,
- the vertex consistency colors $\omega[p, q]$ each of which is contained with multiplicity n, and
- the edge selection colors $\varepsilon[p, q]$ each of which is contained with multiplicity n.

The motif M is defined as $M := \mathcal{C} \setminus D$. It remains to show the correctness.

▶ **Theorem 8.** Graph Motif *does not admit a polynomial-size problem kernel with respect to ℓ even if G is a tree.*

4 Colorful Graph Motif on Trees

For the combination of vertex-colored trees as input graphs and motifs that are sets, the problem becomes considerably easier. First, we show that CGM admits a linear-vertex problem kernel in this case. Moreover, we show that this problem kernel can be computed in linear time. The idea for the problem kernelization is based on two simple observations. First, in all graphs, not only in trees, the number of vertices that are not unique is bounded.

▶ **Observation 9.** *Let (M, G, χ) be an instance of* Colorful Graph Motif. *Then at most 2ℓ vertices in G are not unique.*

Proof. Let C^+ denote the set of colors that occur more than once in G and let $\mathrm{occ}(c)$ denote the number of occurrences of a color c in G. We denote $c^+ := |C^+|$, $n^+ := \sum_{c \in C^+} \mathrm{occ}(c)$, and n^- the number of unique vertices in G. By definition, no color is repeated in M, thus $|M| = c^+ + n^-$; moreover, $|V| = n^+ + n^-$. Hence, the number $\ell = |V| - |M|$ of vertices to delete satisfies $\ell = n^+ - c^+$. By definition $n^+ \geq 2c^+$, and thus we conclude that $\ell \geq n^+/2$. ◀

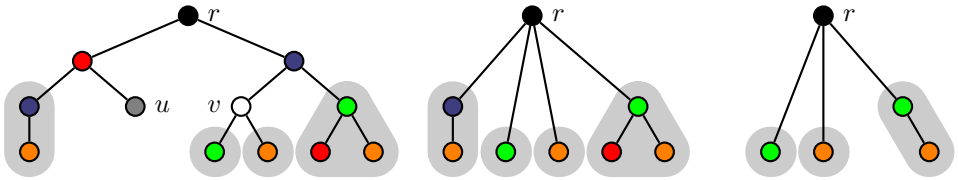


Figure 1 The two phases of the kernelization. Left: The input instance, where r, u, and v have unique colors. The pendant non-unique subtrees are highlighted by the grey background. Middle: after Phase I, all vertices on paths between unique vertices are contracted into r. Right: In Phase II, all vertices with a color that was removed in Phase I are removed together with their descendants.

Second, if there are two vertices that are unique, then the uniquely determined path between these vertices is contained in every occurrence of the motif. The kernelization accordingly removes all the vertices that lie on these paths. More precisely, these vertices are "contracted" into the root r. Afterwards, in a second phase some further vertices are removed because their colors have been used during the contraction. Eventually, this results in an instance which has at most one unique vertex and thus, by Observation 9, bounded size. For an example of the kernelization, see Figure 1. Below, we give a more detailed description.

▶ **Theorem 10.** Colorful Graph Motif *on trees admits a problem kernel with at most* $2\ell + 1$ *vertices that can be computed in* $O(n)$ *time.*

Proof. We first describe the kernelization algorithm, then we show its correctness and finally bound its running time. By Observation 9, the size bound holds if the instance has no unique vertex. Thus, we assume that there is a unique vertex in the following.

Given an instance (G, M, χ) of CGM, first root the input tree G at an arbitrary unique vertex r. Now call a subtree with root v *pendant* if it contains all descendants of v in G. Then, compute in a bottom-up fashion maximal pendant subtrees such that no vertex in this subtree is unique. Call these subtrees the *pendant non-unique subtrees*. By Observation 9, the total number of vertices in pendant non-unique subtrees is at most 2ℓ. Now the algorithm removes vertices in two phases.

Phase I. Remove from G all vertices except r that are not contained in a pendant non-unique subtree. Remove all colors of removed vertices from M. If there is a color c such that two vertices with color c are removed in this step, then return "no". Make r adjacent to the root of each pendant non-unique subtree.

Phase II. In the first step of this phase, for each color c where at least one vertex has been removed in Phase I, remove all vertices from G that have color c. In the second step of this phase, remove all descendants of these vertices. Finally, let M' denote the set of colors that are contained in the remaining instance. This completes the kernelization algorithm; the resulting instance has at most $2\ell + 1$ vertices since all vertices except r are unique. To show correctness, we first observe the following.

Claim: Every occurrence of M in G contains no vertex v that is removed during Phase II of the kernelization. This can be seen as follows. First, every occurrence of M in G contains all vertices removed during Phase I: these vertices are either unique or lie on the uniquely determined path between two unique vertices. Now consider a vertex v removed during Phase II. If v is removed in the first step of Phase II, then v has the same color c as a vertex u

removed during Phase I. Consequently, v is not contained in an occurrence of M: By the above, the occurrence contains u and it contains no other vertex with color c. Otherwise, v is removed in the second step of Phase II, because v is not connected to r. Since every occurrence of M contains r, it thus cannot contain v.

We now show the correctness of the kernelization, that is, the equivalence of the original instance (M, G, χ) and the resulting instance (M', G', χ'). First, assume that (M, G, χ) is a yes-instance. Let S_T be an occurrence of M in G, and let T denote $G[S_T]$; by the above claim, T contains only vertices that are removed during Phase I or that are contained in G'. Consider the subtree T' of G that contains all vertices of T that are not removed during the kernelization. We show that T' is connected in G' and contains all colors of M'. Connectivity can be seen as follows. First, observe that T and T' contain r. Second, any vertex $v \neq r$ of T' is contained in some pendant non-unique subtree of G. Thus, v is in T connected to r via a path that first visits only vertices of T', including the root of the pendant non-unique subtree. The root of the pendant non-unique subtree is in G' adjacent to r. Thus, each vertex $v \neq r$ has in T' a path to r which implies that T' is connected. It remains to prove that T' contains all colors of M'. Consider a color $c \in M'$. Since $c \in M'$, none of the vertices with color c are removed in Phase I of the kernelization. Moreover, since no vertex of T is removed in Phase II of the kernelization, we have that the vertex of T with color c is contained in T'. Thus, T' contains each color of M'. Finally, T' contains each color at most once since T does.

Now assume that (M', G', χ') is a yes-instance and let $S_{T'}$ be an occurrence of M' in G'. Let T denote $G[S_{T'} \cup V_I]$, where V_I is the set of vertices removed during Phase I of the kernelization. We show that T is connected and contains every color of G exactly once. To see that T is connected observe the following: Clearly, $G[\{r\} \cup V_I]$ is connected. Moreover, each vertex $v \neq r$ of T' has in T' a path to r. This path contains a subpath from v to the root r' of the pendant non-unique subtree containing v. In G, r' is adjacent to some vertex of $\{r\} \cup V_I$. Therefore, r' is connected to r in T and thus T is connected. It remains to show that T contains every color of G exactly once. Clearly, T' contains at least one vertex of each color $c \in M'$. Moreover, it also contains at least one vertex of each color $c \in M \setminus M'$ since it contains all vertices of V_I. Besides, it contains each color only once: The vertices of T' have pairwise different colors and different colors than those of the vertices of V_I. Finally, the vertices of V_I have different pairwise colors since the kernelization did not return "no".

The running time can be seen as follows. Determining the pendant non-unique subtrees can be done by a standard bottom-up procedure in linear time. Removing all vertices during Phase I can also be achieved in linear time. After removing a vertex with color c in Phase I, we label c as *occupied*. When we remove a vertex with an occupied color during Phase I, we immediately return "no". After the removal of vertices during Phase I, we can construct M' from M in linear time by removing each occupied color. Finally, we can in linear time add an edge between r and each root of the pendant non-unique subtrees and then remove all remaining vertices that have an occupied color. The final graph G' is obtained by performing a depth-first search from r, in order to include only those vertices still reachable from r. ◀

Now, let us turn to developing fast(er) FPT algorithms for CGM. It can be seen that it is possible to solve CGM in trees in time $1.62^{\ell} \cdot n^{O(1)}$, by 'branching on colors with the most occurrences' until every color appears at most twice. More precisely, for a color c that appears at least three times and some vertex v with color c, we can branch into the two cases to either delete v or to delete the at least two other vertices that have color c. The branching vector[1] for this branching rule is $(1, 2)$ or better. Now, if every color appears at

[1] For an introduction to the analysis of branching vectors, refer to [8, 12].

most twice, then CGM on trees can be solved in polynomial time [10, Lemma 2]. However, by a different branching approach, the above running time can be further reduced.

▶ **Branching Rule 11.** *If there is a color c such that there are two vertices u and v with color c that are both not leaves of the tree G, then branch into the case to delete from G either*

- *the maximal subtree containing u and all vertices w such that the path from v to w contains u, or*
- *the maximal subtree containing v and all vertices w such that the path from u to w contains v.*

Proof of correctness. No occurrence may contain vertices of both subtrees, since in this case it contains u and v which have the same color. ◀

If the rule does not apply, then one can solve the problem in linear time; here, let $\mathrm{occ}(c)$ denote the number of occurrences of a color c in G.

▶ **Lemma 12.** *Let (M, G, χ) be an instance of* Colorful Graph Motif *such that G is a tree and for each color c with $\mathrm{occ}(c) > 1$ at least $\mathrm{occ}(c) - 1$ occurrences of c are leaves of G, then (M, G, χ) can be solved in $O(n)$ time.*

Proof. For each color c with $\mathrm{occ}(c) > 1$, the algorithm simply deletes $\mathrm{occ}(c) - 1$ leaves with color c. This can be done in linear time by visiting all leaves via depth-first search, checking for each leaf in $O(1)$ time whether $\mathrm{occ}(c) > 1$ and deleting the leaf in $O(1)$ time if this is the case. The resulting graph contains each color exactly once, and it is connected since a tree cannot be made disconnected by deleting leaves. ◀

Altogether, we arrive at the following running time.

▶ **Theorem 13.** Colorful Graph Motif *can be solved in $O(\sqrt{2}^{\ell} + n)$ time if G is a tree.*

Proof. The algorithm is as follows. First, reduce the input instance in $O(n)$ time to an equivalent one with $O(\ell)$ vertices using the kernelization of Theorem 10. Now, apply Branching Rule 11. If this rule is no longer applicable, then solve the instance in $O(\ell)$ time (by applying the algorithm behind Lemma 12). Since the graph has $O(\ell)$ vertices, applicability of Branching Rule 11 can be tested in $O(\ell)$ time. Thus, the overall running time is $O(\ell)$ times the number of search tree nodes. Since each application of Branching Rule 11 creates two branches and reduces ℓ by at least two in each branch, the search tree has size $O(2^{\ell/2}) = O(\sqrt{2}^{\ell})$. The resulting running time is $O(\sqrt{2}^{\ell} \cdot \ell + n)$. Furthermore, the factor of ℓ in the running time can be removed by interleaving search tree and kernelization [16], that is, by applying the kernelization algorithm of Theorem 10 in each search tree node. ◀

References

1. Abhimanyu M. Ambalath, Radheshyam Balasundaram, Chintan Rao H., Venkata Koppula, Neeldhara Misra, Geevarghese Philip, and M. S. Ramanujan. On the kernelization complexity of colorful motifs. In *Proc. of the 5th Int'l Symp. on Parameterized and Exact Computation (IPEC'10)*, volume 6478 of *LNCS*, pages 14–25. Springer, 2010.
2. Nadja Betzler, René van Bevern, Christian Komusiewicz, Michael R. Fellows, and Rolf Niedermeier. Parameterized algorithmics for finding connected motifs in biological networks. *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, 8(5):1296–1308, 2011.
3. Andreas Björklund, Petteri Kaski, and Lukasz Kowalik. Constrained multilinear detection and generalized graph motifs. *Algorithmica*, 74(2):947–967, 2016.

4 Hans L. Bodlaender, Bart M. P. Jansen, and Stefan Kratsch. Kernelization lower bounds by cross-composition. *SIAM Journal on Discrete Mathematics*, 28(1):277–305, 2014.

5 Hans L. Bodlaender, Stéphan Thomassé, and Anders Yeo. Kernel bounds for disjoint cycles and disjoint paths. *Theoretical Computer Science*, 412(35):4570–4578, 2011.

6 Edouard Bonnet and Florian Sikora. The graph motif problem parameterized by the structure of the input graph. In *Proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC'15)*, volume 43 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 319–330, 2015.

7 Sharon Bruckner, Falk Hüffner, Richard M. Karp, Ron Shamir, and Roded Sharan. Topology-free querying of protein interaction networks. *Journal of Computational Biology*, 17(3):237–252, 2010. doi:10.1089/cmb.2009.0170.

8 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

9 Holger Dell and Dieter van Melkebeek. Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC'10)*, pages 251–260. ACM, 2010.

10 Michael R. Fellows, Guillaume Fertin, Danny Hermelin, and Stéphane Vialette. Upper and lower bounds for finding connected motifs in vertex-colored graphs. *Journal of Computer and System Sciences*, 77(4):799–811, 2011.

11 Michael R. Fellows, Danny Hermelin, Frances Rosamond, and Stéphane Vialette. On the parameterized complexity of multiple-interval graph problems. *Theoretical Computer Science*, 410(1):53–61, 2009.

12 Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer-Verlag, 1st edition, 2010.

13 Sepp Hartung, Christian Komusiewicz, and André Nichterlein. Parameterized algorithmics and computational experiments for finding 2-clubs. *Journal of Graph Algorithms and Applications*, 19(1):155–190, 2015.

14 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

15 Vincent Lacroix, Cristina G. Fernandes, and Marie-France Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):360–368, 2006.

16 Rolf Niedermeier and Peter Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73(3-4):125–129, 2000.

17 Ron Y. Pinter, Hadas Shachnai, and Meirav Zehavi. Deterministic parameterized algorithms for the graph motif problem. In *Proceedings of the 39th International Symposium on Mathematical Foundations of Computer Science (MFCS'14)*, volume 8635 of *Lecture Notes in Computer Science*, pages 589–600. Springer, 2014. doi:10.1007/978-3-662-44465-8.

18 Ron Y. Pinter and Meirav Zehavi. Algorithms for topology-free and alignment network queries. *J. of Discrete Algorithms*, 27:29–53, 2014. doi:10.1016/j.jda.2014.03.002.

19 Imran Rauf, Florian Rasche, François Nicolas, and Sebastian Böcker. Finding maximum colorful subtrees in practice. *Journal of Computational Biology*, 20(4):311–321, 2013.

20 Roded Sharan and Trey Ideker. Modeling cellular machinery through biological network comparison. *Nature Biotechnology*, 24(4):427–433, 2006.

21 Florian Sikora. An (almost complete) state of the art around the graph motif problem. Technical report, LIGM Université Paris-Est, March 2012. URL: http://www.lamsade.dauphine.fr/~sikora/pub/GraphMotif-Resume.pdf.

Truly Subquadratic-Time Extension Queries and Periodicity Detection in Strings with Uncertainties

Costas S. Iliopoulos[1] **and Jakub Radoszewski**[*2]

1 **Department of Informatics, King's College London, London, UK**
csi@kcl.ac.uk
2 **Department of Informatics, King's College London, London, UK; and Institute of Informatics, University of Warsaw, Warsaw, Poland**
jrad@mimuw.edu.pl

Abstract

Strings with don't care symbols, also called partial words, and more general indeterminate strings are a natural representation of strings containing uncertain symbols. A considerable effort has been made to obtain efficient algorithms for pattern matching and periodicity detection in such strings. Among those, a number of algorithms have been proposed that behave well on random data, but still their worst-case running time is $\Theta(n^2)$. We present the first truly subquadratic-time solutions for a number of such problems on partial words. We show that n longest common compatible prefix queries (which correspond to longest common extension queries in regular strings) can be answered on-line in $\mathcal{O}(n\sqrt{n \log n})$ time after $\mathcal{O}(n\sqrt{n \log n})$-time preprocessing. We also present $\mathcal{O}(n\sqrt{n \log n})$-time algorithms for computing the prefix array and two types of border array of a partial word. We show how our solutions can be adapted to indeterminate strings over a constant-sized alphabet and prove that, unless the Strong Exponential Time Hypothesis is false, the considered problems cannot be solved efficiently over a general alphabet.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases string with don't cares, partial word, indeterminate string, longest common conservative prefix queries, prefix array

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.8

1 Introduction

In this work we deal with different representations of sequential data with uncertainty and imprecision. An (ideal) text is a sequence of symbols from an alphabet Σ. The symbols at some positions may be unknown; in this case they are represented by a *don't care symbol* (sometimes called a *hole* and denoted as $\diamond$) and the resulting sequence is called a *partial word*. In a more general variant, for some positions, instead of a single character from Σ or a hole, a subset of Σ is specified, thus representing a symbol which can be decoded in a number of ways. The presence of such generalised symbols results in a so-called *indeterminate string* (also called a *degenerate string*).

Our main goal here is to develop worst-case efficient algorithms for different variants of pattern matching problem and periodicities detection in the context of strings with uncertainty. The classical pattern matching problem consists in finding all fragments of a given text that match a given pattern. In the presence of uncertainty one needs to specify

* The author is a Newton International Fellow. Supported by the Polish Ministry of Science and Higher Education under the "Iuventus Plus" program in 2015–2016 grant no 0392/IP3/2015/73.

the relation of *matching* (denoted by $\approx$): a don't care symbol matches every other symbol, and a generalised symbol matches every symbol that belongs to the set represented by it (in particular, two generalised symbols match if their sets have a common element). The pattern matching problem is well-studied in the case of partial words [14, 21, 22, 9, 8]. Also if the pattern is an indeterminate string and the text is a regular string, then worst-case efficient [2] or practically efficient [26, 17, 24] algorithms are known.

One of the variants of the pattern matching problem in strings with uncertainty are *longest common compatible prefix queries* (*lccp-queries*), being a natural generalisation of longest common prefix queries in a regular string. Here we are to preprocess a text of length n with uncertain symbols so that the queries for longest matching prefix of any two suffixes of the text can be answered efficiently. They were first defined in [6], where a solution for partial words was presented with $\mathcal{O}(n^2)$ preprocessing time and $\mathcal{O}(1)$ query time for the case of a constant-sized alphabet. A solution with the same complexity for a linearly-sortable alphabet, which works more efficiently in the case that the number of blocks of don't cares in the text is bounded, was shown in [11]. A connected notion is that of a *prefix array*, which stores the answers to the longest common compatible prefix queries between the whole text and all its suffixes. Its $\mathcal{O}(n^2)$ worst-case time (and $\mathcal{O}(n)$ average time) computation for partial words was shown in [18] and for indeterminate strings in [23]. Further combinatorial insights on the prefix array of an indeterminate string have been recently presented in [3, 7].

The basic array of periodicity on strings is the *border array*. It stores, for every prefix of a string, the length of its longest proper border. Its importance stems from applications in pattern matching algorithms and connections with the set of periods of a string; see [10, 13]. There are two different definitions of border on strings with uncertainty; see [16, 23]. A *quantum* border of an uncertain string X is its prefix that matches its suffix. The main weakness of this definition is that if X has a quantum border of length b, there does not necessarily need to exist a solid string S matching X and having a border of length b. For example, this is the case for $X = \mathtt{a} \diamond \mathtt{b}$ which has a quantum border of length 2: $\mathtt{a} \diamond \approx \diamond \mathtt{b}$; however, none of the strings `aab`, `abb` has a border of this length. Therefore, one could be interested in so-called *deterministic* borders: a deterministic border of an uncertain text is defined as a border of some regular string that matches this text. As in the case of regular strings, quantum and deterministic borders correspond to quantum and deterministic *periods* of uncertain texts (the definitions are deferred until Section 2) and thus allow periodicity detection. Quantum periods are also called weak periods and deterministic periods are also called strong periods [5]. Both variants of the border array for a partial word or an indeterminate string can be computed in $\mathcal{O}(n^2)$ worst-case time and $\mathcal{O}(n)$ average time; see [18, 16].

Our Results. In Section 3 we show that, for a partial word of length n, for any $q \in \{1, \ldots, n\}$ one can compute in $\mathcal{O}(n^2 \log n/q)$ time a data structure for answering lccp-queries in $\mathcal{O}(q)$ time. In particular, one can answer n such queries in a partial word in $\mathcal{O}(n\sqrt{n \log n})$ time. In Section 4 we present a construction of the prefix array and both types of a border array – hence, the corresponding types of period array – in the same time complexity. Finally in Section 5 we show that all these results (apart from the deterministic border/period array computation) extend to indeterminate strings over a constant-sized alphabet. Under the word-RAM model the complexities improve by a factor of $\sqrt{\log n}$. We also argue that, under the Strong Exponential Time Hypothesis, none of the considered problems can be solved on indeterminate strings in $\mathcal{O}(n^{2-\epsilon}\sigma^{\mathcal{O}(1)})$ time over an alphabet of size σ, for $\epsilon > 0$.

2 Preliminaries

A *string* S of length $n = |S|$ is a sequence of n letters over a finite alphabet Σ. The letter at the position i, for $1 \le i \le n$, is denoted as $S[i]$. The size of the alphabet is denoted by $\sigma = |\Sigma|$. By $S[i..j]$ we denote a *factor* of S equal to $S[i] \dots S[j]$ (if $i > j$ then it is the empty string ε). The factor is called a *prefix* if $i = 1$ and a *suffix* if $j = n$. The length of the longest common prefix of $S[i..n]$ and $S[j..n]$ is denoted as $\mathsf{lcp}(i, j)$.

If $S[1..b] = S[n-b+1..n]$ then the string $S[1..b]$ is called a *border* of S. A positive integer $p \le n$ is called a *period* of S if $S[i] = S[i+p]$ for all $i = 1, \dots, n-p$. It is known that S has a period p if and only if it has a border of length $n-p$ [10, 13].

For a string S we define the following arrays of length n:

- prefix array π, such that $\pi[i] = \mathsf{lcp}(1, i)$ for $i \ge 2$;
- border array B, such that $B[i]$ is the length of the longest border of $S[1..i]$;
- period array P, such that $P[i]$ is the shortest period of $S[1..i]$.

A *partial word* X of length $n = |X|$ is a sequence of elements $X[1], \dots, X[n]$ from $\Sigma \cup \{\diamond\}$. Here $\diamond \notin \Sigma$ is a special character called a *don't care symbol*. Two characters $a, b \in \Sigma \cup \{\diamond\}$ are said to match (denoted as $a \approx b$) if $a = b$ or $a = \diamond$, or $b = \diamond$. The $\approx$-relation is extended to partial words position by position. Note that $\approx$ is not transitive; for instance, $\mathtt{a} \approx \diamond$ and $\diamond \approx \mathtt{b}$, but $\mathtt{a} \not\approx \mathtt{b}$.

We define a *factor* of X as a partial word $X[i..j] = X[i] \dots X[j]$ (if $i > j$ then it is the empty partial word). A factor is called a *prefix* if $i = 1$ and a *suffix* if $j = n$. The length of the longest common conservative prefix at positions i and j, denoted as $\mathsf{lccp}(i, j)$, is the greatest integer k such that $X[i..i+k-1] \approx X[j..j+k-1]$. Then the prefix array $\pi[2..n]$ of X is defined as $\pi[i] = \mathsf{lccp}(1, i)$.

A *quantum border* of a partial word X is an integer $b \in \{0, \dots, n\}$ such that $X[1..b] \approx X[n-b+1..n]$. A *quantum period* of X is an integer $p \in \{0, \dots, n\}$ such that $X[i] \approx X[i+p]$ for all $i = 1, \dots, n-p$. Those two notions correspond, i.e., if X has quantum period p then it has a quantum border $n-p$ and *vice versa*; see [23]. A *deterministic border* (*deterministic period*) of X is an integer b (p, respectively) such that there exists a string S such that $S \approx X$ and S has a border of length b (a period p, respectively). Here, obviously, we have that if p is a deterministic period of X, then $n-p$ is a deterministic border of X and *vice versa*. Up to the length $\frac{n}{2}$ quantum and deterministic borders of a partial word are the same [16]. However, as we have mentioned before, this does not apply to greater lengths. For partial words we have the following alternative definition of a deterministic period.

▶ **Observation 1.** *A positive integer p is a deterministic period of a partial word X if and only if $X[i] \approx X[j]$ whenever $p \mid i - j$.*

▶ **Example 2.** The partial word

$$\mathtt{a\,b\,a}\diamond\diamond\diamond\mathtt{a}\diamond\mathtt{a\,a}$$

has six quantum periods: 2, 3, 4, 6, 9, 10. For example, 2 is a quantum period because

$$\mathtt{a\,b} \approx \mathtt{a}\diamond \approx \diamond\diamond \approx \mathtt{a}\diamond \approx \mathtt{a\,a}.$$

However, this partial word has only four deterministic periods 3, 6, 9, 10, all corresponding to the solid string

$$\mathtt{aba\,aba\,aba\,a}.$$

As in the case of regular strings, we introduce the border arrays and the period arrays for partial words. By $QB[i]$, $QP[i]$, $DB[i]$, and $DP[i]$ we denote the longest quantum border, shortest quantum period, longest deterministic border, and shortest deterministic period of $X[1..i]$. As we have already mentioned, for every i it holds that $QP[i] = i - QB[i]$ and $DP[i] = i - DB[i]$.

▶ **Example 3.** The following table presents the prefix array and the border arrays of two types of an example partial word.

$X[i]$	a	◇	a	◇	b	a	b	b	b	◇
$\pi[i]$	–	4	2	5	0	2	0	0	0	1
$QB[i]$	0	1	2	3	4	3	4	5	0	1
$DB[i]$	0	1	2	3	2	3	2	0	0	1

We say that a pattern P *occurs* in a text T, both being partial words, at position i if $P \approx T[i..i + |P| - 1]$. Pattern matching on partial words can be done efficiently via convolutions. A line of research lead through alphabet-dependent algorithms and randomized algorithms [14, 21, 22] eventually to an efficient deterministic algorithm; see [9, 8].

▶ **Fact 4.** *Given two partial words P and T of length m and n, respectively, one can find all occurrences of P in T in $\mathcal{O}(n \log m)$ time.*

3 Longest Common Compatible Prefix Queries

Let X be a partial word of length n. In this section we show how to answer lccp-queries for X in $\mathcal{O}(q)$ time after $\mathcal{O}(n^2 \log n/q)$-time preprocessing, for any $q \in \{1, \ldots, n\}$. In the solution we use a dynamic programming approach combined with pattern matching in partial words.

Let us define a family of partial words $X_i = X[(i-1)q + 1..iq]$ for $i = 1, \ldots, \lfloor n/q \rfloor$. Let the array $A[i, j]$ for $i = 1, \ldots, \lfloor n/q \rfloor$ and $j = 1, \ldots, n - q + 1$ be defined as follows: $A[i, j] = 1$ if $X_i \approx X[j..j + q - 1]$, and $A[i, j] = 0$ otherwise.

▶ **Observation 5.** *The array A can be computed in $\mathcal{O}(\frac{n^2 \log n}{q})$ time.*

Proof. Computation of the array is equivalent to pattern matching of each X_i in X. The time complexity follows from Fact 4. ◀

Let the array L for $i = 1, \ldots, \lfloor n/q \rfloor$ and $j = 1, \ldots, n - q + 1$ be defined as follows:

$$L[i, j] = \max\{k \geq 0 : X_i \ldots X_{i+k-1} \approx X[j..j + kq - 1]\}.$$

▶ **Lemma 6.** *The array L can be computed from the array A in $\mathcal{O}(\frac{n^2}{q})$ time.*

Proof. We compute $L[i, j]$ for decreasing values of i and j using a dynamic programming approach. Assume that if $i > \lfloor n/q \rfloor$ or $j > n - q + 1$, then $L[i, j] = 0$. For $i = \lfloor n/q \rfloor, \ldots, 1$ and $j = n - q + 1, \ldots, 1$, if $A[i, j] = 1$, then $L[i, j] = L[i + 1, j + q] + 1$, and otherwise $L[i, j] = 0$. ◀

We answer lccp-queries using the array L. In the query algorithm we use a simple *bounded* lccp routine (denoted as blccp) that for a pair of indices i, j and a length parameter ℓ returns $\min(\mathsf{lccp}(i, j), \ell)$.

▶ **Observation 7.** *blccp(i, j, ℓ) for any i, j, ℓ can be computed in $\mathcal{O}(\ell)$ time.*

```
function lccp(i, j)
k := ⌊i/q⌋;  ℓ := kq − i + 1;
x := blccp(i, j, ℓ);
if x < ℓ then return x;
y := L[k + 1, j + x] · q;
z := blccp(i + x + y, j + x + y, q);
return x + y + z;
```

Figure 1 Function $\mathsf{lccp}(i, j)$.

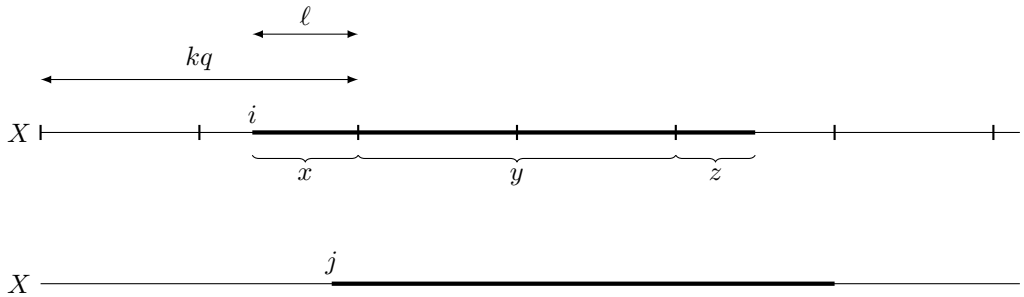


Figure 2 A schematic illustration of the algorithm answering an $\mathsf{lccp}(i, j)$-query. For simplicity the partial word X is depicted twice; the upper copy is divided into fragments of length q. The result of the query is shown in bold.

▶ **Lemma 8.** *Knowing the array L for the partial word X, one can compute $\mathsf{lccp}(i, j)$ for any $i, j \in \{1, \ldots, n\}$ in $\mathcal{O}(q)$ time.*

Proof. The $\mathsf{lccp}(i, j)$ query is answered by the algorithm from the pseudocode in Figure 1. First, we find the smallest ℓ such that $i + \ell \equiv 1 \pmod{q}$. We start with an lccp-query from i and j bounded by ℓ (part x). If the bound is attained, we read the remaining lccp length up to a multiple of q from the array L (part y). The remainder of the result modulo q is computed using a final blccp query (part z); see also Figure 2.

The only non-constant-time operations are two blccp-queries, which can be answered in $\mathcal{O}(q)$ time each by Observation 7. ◀

▶ **Theorem 9.** *Let X be a partial word of length n and $q \in \{1, \ldots, n\}$ be an integer. After $\mathcal{O}(n^2 \log n/q)$-time and $\mathcal{O}(n^2/q)$-space preprocessing one can answer lccp-queries for X in $\mathcal{O}(q)$ time.*

Proof. We use Observation 5 and Lemma 6 for the construction of the data structure and the algorithm of Lemma 8 for answering lccp-queries. ◀

4 Computing Periodicity Arrays

The prefix array of a partial word can be computed via n lccp-queries. By selecting $q = \lfloor \sqrt{n \log n} \rfloor$ in Theorem 9, we obtain $\mathcal{O}(n\sqrt{n \log n})$-time computation of the array. The space usage of this algorithm is $\mathcal{O}(n\sqrt{n/\log n})$. However, we can obtain better space complexity if we refrain from storing the whole array L.

▶ **Corollary 10.** *The prefix array of a partial word of length n can be computed in $\mathcal{O}(n\sqrt{n\log n})$ time and $\mathcal{O}(n)$ space.*

Proof. Consider the array L from the algorithm of Theorem 9. To compute the array π, it suffices to store the values $\ell_j = L[1, j]$ for $j = 2, \ldots, n$ (assuming $L[1, j] = 0$ for $j > n-q+1$). Then

$$\pi[j] \;=\; \ell_j \cdot q \;+\; \mathsf{blccp}(1 + \ell_j \cdot q,\, j + \ell_j \cdot q,\, q),$$

which can be computed in $\mathcal{O}(q)$ time.

The values ℓ_j can be computed with only linear space. Probably the simplest approach is to perform subsequent matching in X of $\lfloor n/q \rfloor$ partial word patterns of the form $X_1 \ldots X_i$ for $i = 1, \ldots, \lfloor n/q \rfloor$. Then as ℓ_j we store the greatest index i such that $X_1 \ldots X_i$ occurs at the position j in X.

By Fact 4, the aforementioned computation of ℓ_j-values takes $\mathcal{O}(n^2 \log n/q)$ time. Knowing those values, we can compute all $\pi[j]$ in $\mathcal{O}(nq)$ time. We select $q = \sqrt{n \log n}$ and obtain an $\mathcal{O}(n\sqrt{n \log n})$-time algorithm. It requires only linear space. ◀

In the case of solid strings one can compute the border array from the prefix array in linear time; see [10, 13]. For partial words we can apply a similar approach to compute the quantum border array. Assume $\pi[n+1] = 0$. We use the following combinatorial observation.

▶ **Observation 11.** *p is a quantum period of $X[1..i]$ if and only if $p \le i \le p + \pi[p+1]$.*

Proof.

(⇒) Assume that p is a quantum period of $X[1..i]$. Then $i - p$ is a quantum border of $X[1..i]$, $X[1..i-p] \approx X[p+1..i]$. Hence, $\pi[p+1] \ge i - p$, i.e., $i \le p + \pi[p+1]$. Obviously, $p \le i$.

(⇐) We have $X[1..\pi[p+1]] \approx X[p+1..p+1+\pi[p+1]-1]$. As $p \le i \le p + \pi[p+1] = p+1+\pi[p+1]-1$, this concludes that $X[1..i-p] \approx X[p+1..i]$. Hence, $i - p$ is a quantum border of $X[1..i]$, so p is a quantum period of $X[1..i]$. ◀

▶ **Lemma 12.** *The quantum border array and the quantum period array of a partial word of length n can be computed in $\mathcal{O}(n)$ time given its prefix array.*

Proof. We focus on computing the array $QP[i]$; the array $QB[i]$ can then be computed in $\mathcal{O}(n)$ time. The algorithm is shown in Figure 3.

In the algorithm we store the last index l for which $QP[l]$ has been computed. For every $p \in \{1, \ldots, n\}$ we set the value of the quantum period to p for positions determined by Observation 11, taking care not to override the previously computed values. As each position in QP is set at most once, the algorithm runs in linear time. ◀

Let us proceed to the computation of deterministic border and period arrays. We will use the following characterisation of a deterministic period of a partial word in terms of its quantum periods, which is a consequence of Observation 1.

▶ **Observation 13.** *A partial word X has a deterministic period p if and only if it has all quantum periods jp for $1 \le j \le \frac{n}{p}$.*

Let us define

$$I_k(p) = [kp, (k+1)p), \quad M_k(p) = \min_{j=1,\ldots,k} (jp + \pi[jp+1]).$$

We combine Observation 11 with Observation 13 to obtain the following criterion.

```
function Compute-QP(X, n)
{ Assume π[n + 1] = 0 }
l := 0;
for p := 1 to n do
    for i := max(p, l + 1) to p + π[p + 1] do
        QP[i] := p;
    l := max(l, p + π[p + 1]);
return QP;
```

Figure 3 Algorithm computing the quantum period array.

▶ **Observation 14.** *If $i \in I_k(p)$, then $X[1..i]$ has a deterministic period p if and only if $i \leq M_k(p)$.*

Using Observation 14 we obtain the following result.

▶ **Lemma 15.** *The deterministic border array and the deterministic period array of a partial word X can be computed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space given its prefix array.*

Proof. First we compute, for every $p \in \{1, \ldots, n\}$, an interval $I(p)$ such that $i \in I(p)$ if and only if $X[1..i]$ has a deterministic period p. For this, notice that the intervals $I_k(p)$ for $k = 1, \ldots, \left\lfloor \frac{n}{p} \right\rfloor$ are pairwise disjoint, their left endpoints are monotonically increasing, whereas the values $M_k(p)$ for $k = 1, \ldots, \left\lfloor \frac{n}{p} \right\rfloor$ are monotonically non-increasing. By Observation 14, we have $I_k(p) \subseteq I(p)$ as long as $M_k(p) \geq (k+1)p - 1$. The last interval included in $I(p)$ is $I_k(p) \cap [1, M_k(p)]$ for the smallest k such that $M_k(p) < (k+1)p - 1$, if such a value of k exists. The computation of $I(p)$ takes $\mathcal{O}(\frac{n}{p})$ time, which gives $\mathcal{O}\left(\sum_{p=1}^{n} \frac{n}{p}\right) = \mathcal{O}(n \log n)$ time in total.

The final step consists in computing the smallest deterministic period of each $X[1..i]$. This is equivalent to the min-variant of the Manhattan skyline problem: for a family of intervals $I(p)$ with heights p we are to compute, for every i, the smallest height of an interval that covers it. Using the linear-time nested union/find data structure [15] this problem can be solved in $\mathcal{O}(n)$ time (see also Section 5.1 in [12]). ◀

We plug Corollary 10 into Lemmas 12 and 15 to arrive at the following final result.

▶ **Theorem 16.** *The prefix array, the quantum border array, the quantum period array, the deterministic border array, and the deterministic period array of a partial word of length n can all be computed in $\mathcal{O}(n\sqrt{n \log n})$ time and $\mathcal{O}(n)$ space.*

▶ Remark. In [18] it is mentioned that all quantum periods/borders of the whole partial word can be computed via a single run of pattern matching, i.e., in $\mathcal{O}(n \log n)$ time. Therefore, by Observation 13, all deterministic periods (hence, borders) of the whole partial word can also be computed in $\mathcal{O}\left(\sum_{p=1}^{n} \frac{n}{p}\right) = \mathcal{O}(n \log n)$ time (and linear space).

5 The Case of Constant Alphabet and Indeterminate Strings

An *indeterminate string* X of length $|X| = n$ over an alphabet Σ of size σ is a sequence of nonempty sets $X[1], \ldots, X[n]$ with $X[i] \subseteq \Sigma$. Two subsets A, B of Σ are said to match

(denoted as $A \approx B$) if they contain at least one letter in common. Under this matching relation one can transfer all notions of pattern matching and periodicity from partial words to indeterminate strings [16, 23]. In this section we show that the majority of the results from the previous sections extend to indeterminate strings over a constant-sized alphabet. Due to large constants hidden in the time complexities, the resulting algorithms are plausible in practice only for a small σ. The most common alphabet over which indeterminate strings are considered is $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Such indeterminate strings occur, e.g., in the FASTA format.

In the data structure of Section 3 we used an efficient pattern matching routine on partial words. The state-of-the-art algorithm for pattern matching on indeterminate strings works in $\mathcal{O}(\sigma n \log n)$ time or in $\mathcal{O}(n\sqrt{n \log n})$ time [2], however, only if the text is a *regular string*. If both the pattern and the text are indeterminate, we obtain an efficient solution for $\sigma = \mathcal{O}(1)$.

▶ **Lemma 17.** *Given two indeterminate strings P and T of length m and n, respectively, over a constant-sized alphabet, one can find all occurrences of P in T in $\mathcal{O}(n \log m)$ time.*

Proof. For every $A \subseteq \Sigma$ we perform the following procedure. Construct a binary string P' of length m such that $P'[i] = 1$ if and only if $P[i] = A$. Construct a binary string T' of length n such that $T'[i] = 1$ if and only if the sets $T[i]$ and A are disjoint. Use an FFT convolution to count, for every alignment of P' and T', the number of common 1s at the corresponding positions of P' and a factor of T'.

In the end we report all alignments for which no common 1 was found in any of the steps. The algorithm works in 2^σ steps, each taking $\mathcal{O}(n \log m)$ time. ◀

Another building block of the lccp data structure are the blccp queries. For indeterminate strings with $\sigma = \mathcal{O}(1)$ they can be implemented in $\mathcal{O}(\ell)$ time just as in Observation 7. We can also answer them slightly faster using standard properties of the word-RAM model.

▶ **Fact 18.** *For an indeterminate string X of length n over an alphabet of size $\sigma = \mathcal{O}(1)$, after $\mathcal{O}(n)$-time and space preprocessing one can compute $\mathsf{blccp}(i, j, \ell)$ in $\mathcal{O}(\ell / \log n)$ time.*

Proof. Consider any $\epsilon > 0$. Let $c = (2 + \epsilon)\sigma$ and $L = \max\left(\left\lfloor \frac{\log n}{c} \right\rfloor, 1\right)$. The number of indeterminate strings of length L over the alphabet of size σ is:

$$2^{\sigma L} \leq 2^{\frac{\sigma \log n}{(2+\epsilon)\sigma}} = n^{\frac{1}{2+\epsilon}} < \sqrt{n},$$

so each of them can be assigned an integer identifier between 1 and $\lfloor \sqrt{n} \rfloor$. For every pair of indeterminate strings of length L we precompute their lccp. There are $2^{2L\sigma}$ such pairs, and the result for each of them can be computed in $\mathcal{O}(L)$ time. All the results can be stored in an array of size $2^{2L\sigma}$. In total this precomputation takes

$$\mathcal{O}(2^{2L\sigma} L) = \mathcal{O}(n^{\frac{1}{1+\epsilon/2}} \log n) = o(n)$$

time.

For every factor of X of length L we then compute its integer identifier. This can be done in $\mathcal{O}(n)$ time if the identifiers are determined by Rabin-Karp-style polynomials with the rolling property; see [13]. Finally a $\mathsf{blccp}(i, j, \ell)$ query is answered by cutting the factors of length ℓ into factors of length L and using the precomputed answers. ◀

▶ Remark. For a partial word over a constant-sized alphabet a much better constant $c = (2 + \epsilon)\log(\sigma + 1)$ would suffice.

Using Lemma 17 and Fact 18 we obtain an implementation of lccp-queries on indeterminate strings.

▶ **Theorem 19.** *Let X be an indeterminate string of length n over a constant-sized alphabet and $q \in \{1, \ldots, \lfloor n/\log n \rfloor\}$ be an integer. After $\mathcal{O}(n^2/q)$-time preprocessing one can answer* lccp*-queries for X in $\mathcal{O}(q)$ time.*

Now the computation of the prefix array and quantum border/period array is the same as in partial words. However, the computation of the deterministic border and period array does not generalise, since Observation 1, and consequently Observation 13, does not hold for indeterminate strings. For example, consider an indeterminate string X of length 3 over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ such that $X[1] = \{\mathtt{a}, \mathtt{b}\}$, $X[2] = \{\mathtt{a}, \mathtt{c}\}$, $X[3] = \{\mathtt{b}, \mathtt{c}\}$. It has a quantum period 1 and $X[1] \approx X[2] \approx X[3] \approx X[1]$. However, it does not have a deterministic period 1 since there is no $s \in \Sigma$ that would match $X[1]$, $X[2]$, and $X[3]$ simultaneously. Therefore we obtain only the following result for indeterminate strings.

▶ **Corollary 20.** *The prefix array, the quantum border array, and the quantum period array of an indeterminate string of length n over a constant-sized alphabet can be computed in $\mathcal{O}(n\sqrt{n})$ time and $\mathcal{O}(n)$ space.*

The time complexities of the algorithms of Corollary 20 have exponential dependency on the alphabet size σ. We will now show that, under some well-known hypotheses, no truly subquadratic algorithms with polynomial dependency on σ exist for any of the considered problems.

The *Orthogonal Vectors Problem* is defined as follows: given two sets A and B containing N vectors from $\{0,1\}^d$ each, does there exist a pair of vectors $\alpha \in A$ and $\beta \in B$ that is orthogonal, i.e., $\sum_{h=1}^{d} \alpha[h]\beta[h] = 0$? The following conjecture is known to be implied (see [25]) by the Strong Exponential Time Hypothesis (SETH), see [19, 20], which asserts that for any $\epsilon > 0$ there is an integer $k > 3$ such that k-SAT cannot be solved in $2^{(1-\epsilon)n}$ time. This conjecture has already been applied to prove hardness results of stringology problems [1, 4].

▶ **Conjecture 21.** *There is no $\epsilon > 0$ and an algorithm that solves the Orthogonal Vectors Problem in $\mathcal{O}(N^{2-\epsilon} \cdot d^{\mathcal{O}(1)})$ time.*

We will show that, under this conjecture, pattern matching on indeterminate strings of length n and $2n$, respectively, both over an alphabet of size σ, cannot be solved in $\mathcal{O}(n^{2-\epsilon} \cdot \sigma^{\mathcal{O}(1)})$ time.

▶ **Theorem 22.** *The Orthogonal Vectors Problem can be reduced to pattern matching of an indeterminate pattern of length n in an indeterminate text of length $2n$, where $n = N$, over an alphabet of size $\sigma = d$.*

Proof. Let $A = \{\alpha_1, \ldots, \alpha_N\}$ and $B = \{\beta_1, \ldots, \beta_N\}$ be the two sets of vectors in $\{0,1\}^d$. Consider an alphabet $\Sigma = \{1, \ldots, \sigma\}$. For a vector $\alpha \in \{0,1\}^d$, by $f(\alpha)$ we denote the subset of Σ defined as: $s \in f(\alpha) \Leftrightarrow \alpha[s] = 1$. Under this mapping, two vectors α and β are orthogonal if and only if the sets $f(\alpha)$ and $f(\beta)$ are disjoint, i.e., the indeterminate symbols $f(\alpha)$ and $f(\beta)$ *do not* match.

We construct an indeterminate pattern $P = f(\alpha_1) \ldots f(\alpha_N)$ and an indeterminate text $T = f(\beta_1) \ldots f(\beta_N) f(\beta_1) \ldots f(\beta_N)$. Then the Orthogonal Vectors Problem for the sets A and B has a positive answer if and only if P *does not* occur in T at any of the positions $1, \ldots, n$. ◀

Let P and T be the indeterminate pattern and text of Theorem 22 and S be the concatenation of P and T. As the pattern matching can be solved by computing the prefix array of S or any of the border arrays of S, or answering n lccp-queries in S, we obtain the following conditional lower bound.

▶ **Corollary 23.** *The prefix array and any of the border arrays of an indeterminate string of length n cannot be computed in $\mathcal{O}(n^{2-\epsilon} \cdot \sigma^{\mathcal{O}(1)})$ time unless SETH fails. Also the problem of answering n lccp-queries in an indeterminate string of length n cannot be solved in $\mathcal{O}(n^{2-\epsilon} \cdot \sigma^{\mathcal{O}(1)})$ time unless SETH fails.*

6 Conclusions and Final Remarks

We have presented a worst-case efficient framework for answering longest common compatible prefix queries in a partial word. We have then shown how we can compute the prefix array and two types of border/period arrays of a partial word basically as fast as answering n lccp-queries. In some cases lccp-queries can be answered faster than using our approach – e.g., if the number of don't care symbols is small or the number of groups of consecutive don't care symbols is small, see [11] – which automatically yields more efficient algorithms for computing the aforementioned arrays.

Then we have presented extensions of all the results apart from the construction of the deterministic border and period array to indeterminate strings over a constant-sized alphabet. We have also argued that, for general alphabets, efficient solutions to any of the considered problems for indeterminate strings would violate the Strong Exponential Time Hypothesis. This, in particular, justifies the usage of heuristic approaches for these problems. As an open question we leave the computation of deterministic periods of an indeterminate string over a constant-sized alphabet in $\mathcal{O}(n^{2-\epsilon})$ time.

Acknowledgements. The authors thank an anonymous referee for a number of helpful suggestions.

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 Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987. doi:10.1137/0216067.

3 Ali Alatabbi, M. Sohel Rahman, and William F. Smyth. Inferring an indeterminate string from a prefix graph. *J. Discrete Algorithms*, 32:6–13, 2015. doi:10.1016/j.jda.2014.12.006.

4 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 51–58. ACM, 2015. doi:10.1145/2746539.2746612.

5 Francine Blanchet-Sadri and Robert A. Hegstrom. Partial words and a theorem of Fine and Wilf revisited. *Theor. Comput. Sci.*, 270(1-2):401–419, 2002. doi:10.1016/S0304-3975(00)00407-2.

6 Francine Blanchet-Sadri and Justin Lazarow. Suffix trees for partial words and the longest common compatible prefix problem. In Adrian Horia Dediu, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*

– *7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings*, volume 7810 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2013. doi:10.1007/978-3-642-37064-9_16.

7 Manolis Christodoulakis, P. J. Ryan, William F. Smyth, and Shu Wang. Indeterminate strings, prefix arrays & undirected graphs. *Theor. Comput. Sci.*, 600:34–48, 2015. doi:10.1016/j.tcs.2015.06.056.

8 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.

9 Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 592–601. ACM, 2002. doi:10.1145/509907.509992.

10 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

11 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, and Tomasz Waleń. A note on the longest common compatible prefix problem for partial words. *J. Discrete Algorithms*, 34:49–53, 2015. doi:10.1016/j.jda.2015.05.003.

12 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theor. Comput. Sci.*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

13 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2003.

14 Michael J. Fischer and Michael S. Paterson. String matching and other products. In Richard Karp, editor, *Proceedings of the 7th SIAM-AMS Complexity of Computation*, pages 113–125, 1974. doi:10.1007/978-3-540-89097-3_14.

15 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

16 Jan Holub and William F. Smyth. Algorithms on indeterminate strings. In *Proceedings of 14th Australasian Workshop on Combinatorial Algorithms*, pages 36–45, 2003.

17 Jan Holub, William F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008. doi:10.1016/j.jda.2006.10.003.

18 Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina Perdikuri, William F. Smyth, and Athanasios K. Tsakalidis. String regularities with don't cares. *Nord. J. Comput.*, 10(1):40–51, 2003.

19 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.

20 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.

21 Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *39th Annual Symposium on Foundations of Computer Science, FOCS'98, November 8-11, 1998, Palo Alto, California, USA*, pages 166–173. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743440.

22 Adam Kalai. Efficient pattern-matching with don't cares. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 655–656. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545468.

23 William F. Smyth and Shu Wang. New perspectives on the prefix array. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *String Processing and Information Retrieval,*

15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10-12, 2008. Proceedings, volume 5280 of *Lecture Notes in Computer Science*, pages 133–143. Springer, 2008. doi:10.1007/978-3-540-89097-3_14.

24 William F. Smyth and Shu Wang. An adaptive hybrid pattern-matching algorithm on indeterminate strings. *Int. J. Found. Comput. Sci.*, 20(6):985–1004, 2009. doi:10.1142/S0129054109007005.

25 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.

26 Sun Wu and Udi Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings USENIX Winter 1992 Technical Conference*, page 153–162, 1992. URL: https://www.usenix.org/legacy/publications/library/proceedings/wu.pdf.

Estimating Statistics on Words Using Ambiguous Descriptions

Cyril Nicaud

Université Paris-Est, LIGM (UMR 8049), F77454 Marne-la-Vallée, France
cyril.nicaud@u-pem.fr

Abstract

In this article we propose an alternative way to prove some recent results on statistics on words, such as the expected number of runs or the expected sum of the run exponents. Our approach consists in designing a general framework, based on the symbolic method developed in analytic combinatorics. The descriptions obtained in this framework are built in such a way that the degree of ambiguity of an object O (i.e., the number of different descriptions corresponding to O) is exactly the value of the statistic under study for O. The asymptotic estimation of the expectation is then done using classical techniques from analytic combinatorics. To show the generality of our method, we not only apply it to obtain new proofs of known results, but also extend them from the uniform distribution to any memoryless distribution.

1998 ACM Subject Classification G.2.1 Combinatorics.

Keywords and phrases random words, runs, symbolic method, analytic combinatorics.

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.9

1 Introduction

In this article we propose an alternative way to prove some recent results on statistics on words, such as the expected number of runs or the expected sum of the run exponents. Studying statistics on words is a classical topic in discrete probabilities, which has many fundamental applications in computer science, for instance in the fields of bioinformatics, information theory and average case analysis of algorithms.

We specially focus on statistics related to the runs in a random word (see Section 2.1 for the definition). Bounding the maximal number of runs in a word is a fundamental question in combinatorics of words, with consequences in text algorithms. Kolpakov and Kucherov proved that it is in $\mathcal{O}(n)$ in their seminal paper [12], and they conjectured that it is at most n. Banai and his coauthors proved this conjecture very recently [1]. Several other statistics, such as the total run length or the sum of exponents, have also been studied in the literature. Besides tightening lower and upper bounds in the worst case [4, 5, 8, 14, 16, 17, 18, 1], works have been done on the expected values of those statistics, for uniform distributions on words [15, 13, 11, 3]. It is the kind of questions we propose to study in this article.

Our main contribution is to provide a general framework, which proves quite useful to obtain asymptotic equivalents to the expectations of statistics related to runs. We follow and adapt the main ideas developed in the field of *analytic combinatorics* (see the textbook of Flajolet and Sedgewick [6]): First we explain how to build the formal power series $L_\chi(z)$ that corresponds to the statistic χ directly from a combinatorial specification on sets of words. Then, we use the techniques of complex analysis to estimate the expectation $\mathbb{E}_n[\chi]$ of χ for uniform random words of length n. The main difference with the classical framework is that the combinatorial specifications we use are *ambiguous*. Usually, unambiguity is mandatory for this combinatorial method to apply. However, if the degree of ambiguity of the specification

27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016).
Editors: Roberto Grossi and Moshe Lewenstein; Article No. 9; pp. 9:1–9:12

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

for a word w, i.e. the number of ways to produce w, is exactly $\chi(w)$, then we can directly get an expression of $L_\chi(z)$, or an equation it satisfies.

The net gain of this method is that once $L_\chi(z)$ is known, no tedious computations are needed to get the asymptotic equivalent of $\mathbb{E}_n[\chi]$. The tools from analytic combinatorics apply and directly yield the result. Moreover, this framework can be used to go beyond uniform distributions, since it can easily be extended to memoryless distributions, where each letter is chosen independently with some fixed probability on the alphabet.

The technique we propose is quite natural, and there are hints of its use, for instance, in [6, A.7.] and also in the study of hidden words [7]. However, it lacks a general framework, which is what we propose and illustrate in this article. This introduction is continued in Section 3, where we present the method on three basic examples, after the required notations given in Section 2. This is done in an informal way, but it should gives a fair picture of our method. The formalism of weighted sets is then introduced in Section 4. In Section 5, we propose alternative proofs to some results of the literature. Finally, we explain in Section 6 how to generalize them to memoryless distributions.

2 Preliminaries

For any two nonnegative integers i, j, let $[i, j]$ denote the integer interval $\{i, \dots, j\}$. By convention, $[i, j] = \emptyset$ if $j < i$. Let also $[i]$ denote the integer interval $[1, i]$.

The *mobius function* $\mu : \mathbb{Z}_{\geq 1} \to \{-1, 0, 1\}$ is defined as follows. If $n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$ is the decomposition of a positive n into prime numbers, then $\mu(n) = (-1)^k$ if all the α_i's are equal to 1, and $\mu(n) = 0$ otherwise. The main property of this function is that f and g are two functions from $\mathbb{Z}_{\geq 1}$ such that $f(n) = \sum_{d|n} g(d)$, then $g(n) = \sum_{d|n} \mu\left(\frac{n}{d}\right) f(d)$, where $d|n$ means that d ranges over the divisors of n.

2.1 Words and Probabilities on Words

In the sequel we consider words on a finite alphabet A, of cardinality $\ell \geq 2$. We assume the reader is familiar with the classical definitions on words, such as prefixes, suffixes, factors, subwords ... For $w \in A^*$ of length n and $i \in [n]$, let w_i (or $w[i]$) denote the i-th letter of w, with the convention that positions start at 1. The last letter of w is therefore $w_{|w|}$. Let also $w[i, j] = w_i \cdots w_j$ denote the factor of w that starts at position i and ends at position j.

Recall that a word w is *not primitive* when there exists a word v and an integer $k \geq 2$ such that $w = v^k$, and that it is *primitive* otherwise. Let $\mathcal{P}$ denote the set of all primitive words. A word w of length n is *periodic with period* $p \geq 1$ when $w[i] = w[i + p]$, for every $i \in [n - p]$. The *period* of a word is its smallest period. If w is periodic with period p, then its *exponent* is $\frac{|w|}{p}$. The exponent is not necessarily an integer: for instance the exponent of *ababa* is 5/2. A *run of period* p in a word w is a factor $w[i, j]$ of w with least period p, such that $p \geq 2$ and $w[i - 1, j]$ and $w[i, j + 1]$, when they exist, are not of period p (the factor $w[i, j]$ is "maximal" for the period p). We identify such a run by the triplet (i, j, p). Let $\textsc{runs}(v)$ denote the set of all runs in the word v.

The *uniform distribution* on a finite set E is the probability p defined for all $e \in E$ by $p(e) = \frac{1}{|E|}$. By a slight abuse of notation, we will speak of the *uniform distribution on* A^* to denote the sequence $(p_n)_{n\geq 0}$ of uniform distributions on A^n. For instance, if $A = \{a, b, c\}$, then each element of A^n has probability 3^{-n} under this distribution.

Another very classical distribution on A^n is the *memoryless distribution of probability* p, where p is a probability on the alphabet A. Under this distribution, the probability of a word

$w = w_1 \cdots w_n \in A^n$ is $\mathbb{P}_p(w) = p(w_1) \cdots p(w_n)$. This distribution consists in generating each letter of the word independently, following p.

2.2 Elements of Analytic Combinatorics

We only present the parts of this well-established theory that will be needed in the sequel. For more information, the reader is referred to the book of Flajolet and Sedgewick [6].

A set $\mathcal{E}$ with a size function $s : \mathcal{E} \to \mathbb{N}$ is a *combinatorial set* if for every $n \in \mathbb{N}$, $\mathcal{E}_n := s^{-1}(n)$ is finite. The *generating series* $E(z)$ of $\mathcal{E}$ is defined by $E(z) := \sum_{e \in \mathcal{E}} z^{s(e)} = \sum_{n \geq 0} e_n z^n$, with $e_n = |\mathcal{E}_n|$. We will also use the notation $[z^n]E(z) := e_n$ to denote the n-th coefficient $E(z)$. If $\mathcal{E}$ and $\mathcal{F}$ are two combinatorial sets of size functions s and t, $\mathcal{E} \times \mathcal{F}$ is also a combinatorial set for the size function $r((e, f)) = s(e) + t(f)$, for every $e \in \mathcal{E}$ and $f \in \mathcal{F}$. This construction extends naturally to $\mathcal{E}_1 \times \cdots \times \mathcal{E}_k$ and to $\mathcal{E}^k$, for every $k \geq 2$.

The *symbolic method* consists in a dictionary to directly translate unambiguous combinatorial specifications into equations on generating series. In particular:

▶ **Theorem 1** ([6]). *For $\mathcal{E}$ and $\mathcal{F}$ two combinatorial sets of generating series $E(z)$ and $F(z)$:*

- *If $\mathcal{E}$ and $\mathcal{F}$ are two disjoint sets, then $\mathcal{G} = \mathcal{E} \dot\cup \mathcal{F}$ implies that $G(z) = E(z) + F(z)$.*
- *If $\mathcal{G} = \mathcal{E} \times \mathcal{F}$, then $G(z) = E(z)F(z)$.*
- *If $\mathcal{E}_0 = \emptyset$ and $\mathcal{G} = \mathcal{E}^* := \cup_{k \geq 0} \mathcal{E}^k$, then $G(z) = \frac{1}{1-E(z)}$.*

There are other basic constructions, but we will not need them in this article. However, there is a more advanced tool that is particularly useful for us: If $\mathcal{E}_0 = \emptyset$, a tuple of elements of $\mathcal{E}$ is *primitive* when, it is primitive as a word on the alphabet $\mathcal{E}$. From [6, A.4] we get that if $\mathcal{F}$ is the set of primitive tuples of elements of $\mathcal{E}$, then

$$F(z) = \sum_{k \geq 1} \frac{\mu(k)\, E(z^k)}{1 - E(z^k)}. \tag{1}$$

As an illustration, observe that the generating series of the alphabet is $A(z) = \ell z$, as there are ℓ letters, each of size 1. Since a word is a tuple of letters, the generating series of all words[1] is $\frac{1}{1-A(z)} = \frac{1}{1-\ell z}$. Moreover, by Equation 1, the generating series $P(z)$ of the set $\mathcal{P}$ of primitive words on A is

$$P(z) = \sum_{k \geq 1} \frac{\mu(k)\, \ell z^k}{1 - \ell z^k}. \tag{2}$$

The second part of the theory consists in considering generating series as analytic functions from $\mathbb{C}$ to $\mathbb{C}$, and then in using the powerful techniques of this field. We referred the reader to [6] for the classical definitions of the theory of analytic functions. In the sequel, we will only use the following theorem, which is a simplified version of the classical Transfer Theorem [6, p.393]. The full version is much more powerful, but it requires some analytic conditions that are too long to introduce for this extended abstract.

▶ **Theorem 2** (Simplified Transfer Theorem [6]). *Let r be a positive real number. Let f be a function from $\mathbb{C}$ to $\mathbb{C}$, which is analytic at 0, with radius of convergence greater than r. For any $k \in \mathbb{Z}_{\geq 1}$, we have the following asymptotic equivalent as n tends to infinity,*

$$[z^n] \frac{f(z)}{(1 - z/r)^k} \sim \frac{f(r)\, n^{k-1}}{(k-1)!\, r^n}.$$

[1] This elementary result can of course be obtained directly.

We will also use Theorem 2 the following way in the sequel: if $f_1, \ldots, f_k$ are analytic at 0 and of radius of convergence greater than r, then applying the theorem to each term yields

$$[z^n]\left(\frac{f_1(z)}{1-z/r} + \frac{f_2(z)}{(1-z/r)^2} + \ldots + \frac{f_k(z)}{(1-z/r)^k}\right) \sim \frac{f_k(r)\, n^{k-1}}{(k-1)!\, r^n}, \tag{3}$$

since the other terms are negligible when n tends to infinity.

Extracting the n-th coefficient of Equation (2) yields the well known fact that if P_n denote the number of primitive words, then $P_n = \sum_{d|n} \ell^{n/d} \sim \ell^n$. Hence, $P(z)$ is analytic at 0 and its radius of convergence is $1/\ell$. This simple fact will be quite useful in the sequel.

If χ is a statistic on a combinatorial set $\mathcal{E}$, i.e. a mapping from $\mathcal{E}$ to $\mathbb{R}$, the *cumulative generating series of* χ is the formal power series $L_\chi(z) = \sum_{e\in\mathcal{E}} \chi(e) z^{|e|}$. Observe that the expectation of χ for uniform random elements of $\mathcal{E}_n$ is given by $\mathbb{E}_n[\chi] = [z^n]L_\chi(z)/[z^n]E(z)$. Since we focus on statistics on words in this article, we will always have $[z^n]E(z) = \ell^n$, the number of words of length n, except in Section 6 where we directly work with probabilities.

3 Three Introductory Examples

In this section we study three basic examples, to illustrate how some statistics on random words can be estimated using ambiguous specifications. We will not be fully formal, the rigorous framework will be presented in the next section.

We start with the classical question of estimating the expected number occurrences of a fixed pattern v of length m in a uniform random word w of length n. Occurrences may overlap: aaa has two occurrences of aa in our settings. Let α_v be the random variable that counts the number of occurrences of v in w. The classical probabilistic analysis of the expectation $\mathbb{E}_n[\alpha_w]$ of α_w for the uniform distribution on A^n is the following: for any $i \in [n]$ let X_i be the random variable that values 1 if there is an occurrence of v in w starting at position i and that values 0 otherwise. Then we have $\alpha_v = \sum_{i=1}^n X_i$. The X_i's are not independent, but since the expectation is linear, we have $\mathbb{E}_n[\alpha_v] = \sum_{i=1}^n \mathbb{E}[X_i]$. As a consequence, $\mathbb{E}[Z_n] = (n-m+1)\ell^{-m} \sim n\ell^{-m}$, as v is fixed in our settings.

As we are working with the uniform distribution, the probabilistic proof can also be established in a purely combinatorial manner: We just count the number of words of length n having an occurrence of v at position i, and get exactly the same computations.

There is another, more advanced, way to obtain this result using combinatorics. The symbolic method described in Section 2.2 works when one starts with an unambiguous combinatorial specification. If the regular expression is ambiguous, then applying blindly the rules of transformation does not produce the correct generating series. Nonetheless, the resulting series $L(z)$ can still be useful: roughly speaking, if $\kappa(w)$ denote the number of different ways that the word w can be parsed in the expression (we call this quantity the *degree of ambiguity of* w), then $L(z) = \sum_w \kappa(w) z^{|w|}$. We can take advantage of this property, provided we can design an ambiguous expression such that for every word, *the value of the statistic is equal to its degree of ambiguity.* Back to our example, it is not difficult to see that for the ambiguous expression $\mathcal{L} = A^* v A^*$, each word w can be parsed in a number of ways equal to the number of occurrences of v in w. Hence, using the dictionary of the symbolic method, we get that $L_{\alpha_v}(z) = \frac{z^m}{(1-\ell z)^2}$. From this expression we obtain:

$$\sum_{|w|=n} \alpha_v(w) = [z^n]\frac{z^m}{(1-\ell z)^2} = [z^{n-m}]\frac{1}{(1-\ell z)^2} = (n-m+1)\ell^{n-m}.$$

We just have to divide by ℓ^n to get the expectation of α_v. Instead, we can use the Simplified Transfer Theorem directly on $\frac{z^m}{(1-\ell z)^2}$ to obtain that $\mathbb{E}_n[\alpha_v] \sim n\ell^{-m}$. It is probably too

complicated to use analytic combinatorics here, but in many situations, we will not want to find an exact expression for the n-th coefficient, if it can be avoided. Using the Transfer Theorem, we can find asymptotic equivalents without first computing the coefficients.

Let us consider another simple example. Assume that we are now interested in the number $\beta_v(w)$ of occurrences of v as a subword of w. The expectation of β_v for random words of length n can be established using probabilities and the linearity of the expectation as for α_v. However, we want to illustrate the use of analytic tools once more. It is not difficult to verify that the ambiguous expression $\mathcal{L} = A^*v_1A^*v_2A^* \cdots A^*v_mA^*$ corresponds to our needs. Its associated generating series is $L(z) = \frac{z^m}{(1-\ell z)^{m+1}}$, which satisfies the conditions of the Simplified Transfer Theorem. This yields that $[z^n]L(z) \sim \frac{\ell^{n-m}n^m}{m!}$. As there are ℓ^n words of length n, the expected number of occurrences of v as a subword of a random word of length n is asymptotically equivalent to $\frac{n^m}{m!\ell^m}$. See [7] for more information on statistics related to subwords.

We conclude this section with a last elementary example. Let $\pi(w)$ denote the length of the largest word v such that $w \in vA^*\overline{v}$, where $\overline{v}$ denote the reverse (or mirror) of v. The description $\mathcal{L} = \cup_{v\in A^+} vA^*\overline{v}$ is ambiguous, but a word w is in exactly $\pi(w)$ sets of this union, since the number of nonempty prefixes of a word is equal to its length. The specification $\mathcal{L}$ can be rewritten $\mathcal{E}\times A^*$, where $\mathcal{E}$ is the set of pairs $(v, \overline{v})$ for nonempty v. The generating series of $\mathcal{E}$ is $E(z) = \frac{\ell z^2}{1-\ell z^2}$, and the symbolic method yields that $L(z) = \frac{E(z)}{1-\ell z}$. As $E(z)$ is analytic at 0 with radius of convergence $\frac{1}{\sqrt{\ell}} > \frac{1}{\ell}$, the Simplified Transfer Theorem applies and yields that $[z^n]L(z) \sim E(\ell^{-1})\ell^n = \frac{1}{\ell-1}\ell^n$. Hence, the expected value of π tends to $\frac{1}{\ell-1}$.

In the sequel, we define a framework on *sets of weighted words* to formalize what we did for our three introductory examples. It is directly inspired from the simple remarks we just made, on how ambiguity can be used to estimate statistics. However, this is done in a more sophisticate way. We will be able, for instance, to handle non-integer degrees of ambiguity, which will prove useful in Section 5.

4 Combinatorics of Sets of Weighted Words

In this section we introduce the framework that will be used throughout this article. The idea is to formalize the notion of "number of time an ambiguous expression is parsed", and to do it in a way similar to the symbolic method. For this purpose, we have to introduce some formalism on sets of weighted words. The definitions we propose are natural extensions of the classical ones on sets.

Consider the two sets of words $\mathcal{E} = \{a, ab, aa\}$ and $\mathcal{F} = \{\varepsilon, a, b\}$. We interpret them as "each word of $\mathcal{E}$ has weight 1", and the same for $\mathcal{F}$. Since a is in both $\mathcal{E}$ and $\mathcal{F}$, we would like a to have weight two in $\mathcal{E} \cup \mathcal{F}$. Similarly, since $ab = a \cdot b = ab \cdot \varepsilon$, we would like ab to have weight two in $\mathcal{E} \cdot \mathcal{F}$. Finally, since $aaa = a \cdot a \cdot a = a \cdot aa = aa \cdot a$, we would like aaa to have weight three in $\mathcal{E}^*$. A relevant way to handle this is to use multisets, that is, sets where an element may appear more than once. We will need a bit more in the sequel, and thus allow the weights to take any real positive value in the definitions below.

Formally, if $\mathcal{E}$ be a nonempty set, a *weighted set*[2] on $\mathcal{E}$ is a mapping $\mathcal{M}$ from $\mathcal{E}$ to $\mathbb{R}_{\geq 0}$. For $e \in \mathcal{E}$, we say that e is in $\mathcal{M}$ (written $e \in \mathcal{M}$) if $\mathcal{M}(e) \neq 0$, and we write $e \notin \mathcal{M}$ otherwise. A set $\mathcal{M}$ is viewed as a weighted set where every element of e has weight 1: for every $e \in \mathcal{E}$, $\mathcal{M}(e) = 1$ if $e \in \mathcal{M}$ and $\mathcal{M}(e) = 0$ otherwise.

[2] We use the terminology "weighted set on $\mathcal{E}$" for "set of weighted elements of $\mathcal{E}$", as a weighted graph is a graph of weighted vertices.

If $\mathcal{E}$ is a combinatorial set of size function s, we define *the generating series* $M(z)$ *of a weighted set* $\mathcal{M}$ *on* $\mathcal{E}$ by $M(z) = \sum_{e\in\mathcal{E}} \mathcal{M}(e)z^{s(e)}$. Observe that if $\mathcal{M}$ is a set, then the generating series of $\mathcal{M}$ viewed as a weighted set or as a set coincide.

From now on, we only work on weighted sets of words on A. To simplify the notations, we will sometimes write $\mathcal{M} = \{a \mapsto \frac{1}{2}, ba \mapsto 3, baba \mapsto 11\}$ for the weighted set defined by $\mathcal{M}(a) = \frac{1}{2}$, $\mathcal{M}(ba) = 3$, $\mathcal{M}(baba) = 11$, and $\mathcal{M}(x) = 0$ for every $x \notin \{a, ba, baba\}$.

If $\mathcal{M}$ and $\mathcal{M}'$ are two weighted sets of words, the *sum* $\mathcal{M} \oplus \mathcal{M}'$ is the weighted set $\mathcal{N}$ defined by $\mathcal{N}(w) = \mathcal{M}(w) + \mathcal{M}'(w)$, for every $w \in A^*$. The *concatenation* $\mathcal{M} \odot \mathcal{M}'$ of the weighted sets $\mathcal{M}$ and $\mathcal{M}'$ is defined by

$$\mathcal{M} \odot \mathcal{M}' = \bigoplus_{\substack{v\in\mathcal{M}\\ v'\in\mathcal{M}'}} \{vv' \mapsto \mathcal{M}(v)\mathcal{M}'(v')\}.$$

That is, every pair (v, v') contributes additively to $\mathcal{M}(v)\mathcal{M}(v')$ to the weight of the word vv'. For instance, if $\mathcal{M} = \{a \mapsto 1/2, ab \mapsto 3\}$ and $\mathcal{M}' = \{\varepsilon \mapsto 5, b \mapsto 7\}$, then their concatenation is $\mathcal{M} \odot \mathcal{M}' = \{a \mapsto 5/2, ab \mapsto 37/2, abb \mapsto 21\}$.

If $\varepsilon \notin \mathcal{M}$, the *star* $\mathcal{M}^\star$ is defined by $\mathcal{M}^\star = \oplus_{k\geq 0}\mathcal{M}^k$, where $\mathcal{M}^0 = \{\varepsilon \mapsto 1\}$ and $\mathcal{M}^{k+1} = \mathcal{M}^k \odot \mathcal{M}$ for every $k \geq 0$. Observe that if $\varepsilon \in \mathcal{M}$, then this operation is not well defined, as ε is in every $\mathcal{M}^k$ and therefore has infinite weight in $\mathcal{M}^*$.

The following proposition extends the symbolic method to weighted sets of words.

▶ **Proposition 3.** *If* $\mathcal{M}$ *and* $\mathcal{M}'$ *are two weighted sets of words, then*

$$\begin{aligned}
\mathcal{N} = \mathcal{M} \oplus \mathcal{M}' \quad &\Rightarrow \quad N(z) = M(z) + M'(z),\\
\mathcal{N} = \mathcal{M} \odot \mathcal{M}' \quad &\Rightarrow \quad N(z) = M(z)M'(z),\\
\mathcal{N} = \mathcal{M}^\star \quad &\Rightarrow \quad N(z) = \frac{1}{1 - M(z)}, \qquad (\text{if } \varepsilon \notin \mathcal{M}).
\end{aligned}$$

In the sequel, we will implicitly use the following lemma, which was already presented informally in Section 3.

▶ **Lemma 4.** *Let* $\alpha_v(w)$ *denote the number of occurrences of* v *as a factor of* w*. The generating series of the weighted set* $A^* \odot \{v \mapsto 1\} \odot A^*$ *(the weighted set version of* A^*vA^**) is equal to* $L_{\alpha_v}(z)$*, the cumulative generating series of the statistic* α_v*.*

Proof. As A^*v is a unambiguous expression, every element of $A^* \odot \{v \mapsto 1\}$ has weight 1, and the same holds for A^*. Thus, by definition, if $\mathcal{N} = (A^* \odot \{v \mapsto 1\}) \odot A^*$, then $\mathcal{N}(w) = |\{(w_1, w_2) \in A^* \times A^* : w = w_1 v \cdot w_2\}|$, which is exactly $\alpha_v(w)$, as announced. ◀

5 Application to Run Statistics

5.1 The Expected Number of Runs

For any given word v, let $\rho(v)$ denote its number of runs. In [15], Puglisi and Simpson established the following result.

▶ **Theorem 5** ([15])**.** *The expected number of runs in a word of length* n *on an alphabet of size* ℓ *satisfies asymptotically*

$$\mathbb{E}_n[\rho] \sim \left(\frac{\ell - 1}{\ell} \sum_{k\geq 1} \frac{\mu(k)}{\ell^{2k-1} - 1}\right) n.$$

To prove Theorem 5, they proceed as follows. For every given p, they compute the total number of runs of period p in the set of all words of length n. Then, they sum these values for all possible p. Finally, they obtain an asymptotic equivalent of this quantity using elementary, but technical, computations.

In this section, we propose an alternative proof of Theorem 5 using our framework. Recall that $\mathcal{P}$ is the set of all primitive words and that $P(z)$ is its associated generating series. Let $\mathcal{C} = \{ww \mapsto 1 : w \in \mathcal{P}\}$ and let $\mathcal{D} = \{aww \mapsto 1 : w \in \mathcal{P} \text{ and the last letter of } w \neq a\}$.

▶ **Lemma 6.** *The generating series of the weighted set $(\mathcal{C} \odot A^*) \oplus (A^* \odot \mathcal{D} \odot A^*)$ is the cumulative generating series of the statistic ρ.*

Proof. For the weighted set $\mathcal{M} = \mathcal{C} \odot A^* = \oplus_{w\in\mathcal{P}}\{ww \mapsto 1\} \odot A^*$, $\mathcal{M}(w)$ is the number of prefixes of the form ww for $w \in \mathcal{P}$, that is, $\mathcal{M}$ counts the number of runs at the beginning of the word. Similarly, for $\mathcal{N} = A^* \odot \mathcal{D} \odot A^* = \oplus_{w\in\mathcal{D},a\neq w[|w|]} A^* \odot \{aww \mapsto 1\} \odot A^*$, $\mathcal{N}(w)$ is the number of runs of w that does not start at the first position, since each run is identified by the factor aww. Hence, $\mathcal{M} \oplus \mathcal{N}$ counts the number of runs, concluding the proof. ◀

The generating series of $\mathcal{C}$ and $\mathcal{D}$ are $C(z) = P(z^2)$ and $D(z) = (\ell - 1)zP(z^2)$, respectively. Hence, the cumulative generating series $L_\rho(z)$ of the number of runs can be obtained using Proposition 3:

$$L_\rho(z) = \frac{P(z^2)}{1-\ell z} + \frac{(\ell-1)zP(z^2)}{(1-\ell z)^2}.$$

Since the radius of convergence of $P(z^2)$ is $\frac{1}{\sqrt{\ell}} > \frac{1}{\ell}$, we are in the settings of Equation (3) and the Simplified Transfer Theorem yields that $[z^n]L_\rho(z) \sim n\frac{\ell-1}{\ell}P(\ell^{-2})\ell^n$. Dividing by ℓ^n gives another expression for the result of Theorem 5:

$$\mathbb{E}_n[\rho] \sim \frac{\ell-1}{\ell}\, P\left(\frac{1}{\ell^2}\right) n. \tag{4}$$

In particular, the infinite sum of Theorem 5 is just $P(\ell^{-2})$. Indeed, by Equation (2) we have

$$P\left(\frac{1}{\ell^2}\right) = \sum_{k\geq 1} \frac{\mu(k)\,\ell\cdot\ell^{-2k}}{1-\ell\cdot\ell^{-2k}}.$$

Multiplying the numerator and denominator by ℓ^{2k-1} yields the formula of Theorem 5.

5.2 The Expected Total Run Length

The *total run length* of a word is the sum of the lengths of its runs. We denote by $\tau(w)$ the total run length of w. In [11], Glen and Simpson proved the following result.

▶ **Theorem 7** ([11])**.** *The expected total run length of a uniform random word of length n asymptotically satisfies*

$$\mathbb{E}_n[\tau] \sim \left(\sum_{k\geq 1} P_k \frac{2k(\ell-1)+1}{\ell^{2k+1}}\right) n,$$

where P_k is the number of primitive words of length k.

Their techniques follows the steps of the proof of Theorem 5 given in Section 5.1.

In order to prove Theorem 7 with our framework, we first focus on another statistic. For any word w, let $\delta(w)$ denote the sum of the periods of the runs of w. We are interested in the expected value of δ for uniform random words of length n. Consider the weighted set $\overline{\mathcal{C}} = \{ww \mapsto |w| : \ w \in \mathcal{P}\}$, where the weight of each ww is the length of w. A direct computation yields that the generating series of $\overline{\mathcal{C}}$ is $\overline{C}(z) = z^2P'(z^2)$. Similarly the generating series of the weighted set $\overline{\mathcal{D}} = \{aww \mapsto |w| : \ w \in \mathcal{P}$ and the last letter of $w \neq a\}$ is $\overline{D}(z) = (\ell-1)z^3P'(z^2)$.

We can now reuse the ambiguous specification of Lemma 6, with $\overline{\mathcal{C}}$ and $\overline{\mathcal{D}}$ instead of $\mathcal{C}$ and $\mathcal{D}$, and get that the cumulative generating series of δ is

$$L_\delta(z) = \frac{z^2P'(z^2)}{1-\ell z} + \frac{(\ell-1)z^3P'(z^2)}{(1-\ell z)^2}, \text{ with } P'(z) = \frac{d}{dz}P(z).$$

By Equation 3, from this expression of $L_\delta(z)$ we directly get the following proposition.

▶ **Proposition 8.** *The expected sum of the periods of the runs in a uniform random word of length n asymptotically satisfies $\mathbb{E}_n[\delta] \sim \frac{\ell-1}{\ell^3}P'(\ell^{-2})\, n$.*

We can now proceed with our proof of Theorem 7. Consider the ambiguous specification $\mathcal{L} = \cup_{w\in\mathcal{P}} A^*wwA^*$. Observe that a run $r = (i, j, p)$ in a word v matches the expression of $\mathcal{L}$ exactly once for every $w = v[k, k+p-1]$, with $k \in \{i, \dots, j-2p+1\}$. That is, the pair (v, r) matches the specification exactly $|r| - 2p + 1$ times. In other words, the generating series of the weighted set $A^* \odot \mathcal{C} \odot A^*$ is the cumulative generating series of the statistic $\tau - 2\delta + \rho$ (recall that τ is the total run length, δ is the sum of periods and ρ is the number of runs). Thus, Proposition 3 directly yields:

$$\frac{P(z^2)}{(1-\ell z)^2} = L_\tau(z) - 2L_\delta(z) + L_\rho(z) \Rightarrow L_\tau(z) = \frac{P(z^2)}{(1-\ell z)^2} + 2L_\delta(z) - L_\rho(z).$$

Theorem 2 applies and we obtain that

$$\mathbb{E}_n[\tau] = \frac{1}{\ell^n}[z^n]L_\tau(z) \sim \left(\frac{2(\ell-1)}{\ell^3}P'\left(\frac{1}{\ell^2}\right) + \frac{1}{\ell}P\left(\frac{1}{\ell^2}\right)\right) n, \tag{5}$$

which is another formulation of Theorem 7. Indeed, since $P(z) = \sum_{k\geq 1} P_k z^k$, we have

$$\frac{1}{\ell}P\left(\frac{1}{\ell^2}\right) = \frac{1}{\ell}\sum_{k\geq 1}\frac{P_k}{\ell^{2k}} = \sum_{k\geq 1}\frac{P_k}{\ell^{2k+1}}.$$

Moreover, $P'(z) = \sum_{k\geq 1} kP_k z^{k-1}$, and thus

$$\frac{2(\ell-1)}{\ell^3}P'\left(\frac{1}{\ell^2}\right) = \frac{2(\ell-1)}{\ell^3}\sum_{k\geq 1}\frac{kP_k}{\ell^{2k-2}} = \sum_{k\geq 1}P_k\frac{2k(\ell-1)}{\ell^{2k+1}}.$$

Summing the two terms yields the formula of Theorem 7.

5.3 The Expected Sum of Exponents

For any word $v \in A^*$, let $\gamma(v)$ denote the sum of the exponents of the runs of v. In [13], Kusano, Matsubara, Ishino and Shinohara proved the following result.

▶ **Theorem 9** ([13]). *The expected sum of the exponents of runs for uniform random words of length n satisfies asymptotically:*

$$\mathbb{E}_n[\gamma] \sim \left(\sum_{k\geq 1} \mu(k) \left(\frac{2(\ell-1)}{\ell^{2k}-\ell} + \frac{1}{k\ell} \log\left(\frac{\ell^{2k}}{\ell^{2k}-\ell} \right) \right) \right) n.$$

We follow the analysis of the previous section: A run $r=(i,j,p)$ in a word v matches the expression $\mathcal{L} = \cup_{w\in\mathcal{P}} A^* w w A^*$ exactly $|r|-2p+1$ times. Since we want to compute the statistic γ, we have to divide the contribution of each run (i,j,p) by p.

Let $\tilde{\mathcal{C}} = \{ww \mapsto \frac{1}{|w|} : w \in \mathcal{P}\}$ and let $\tilde{\mathcal{D}} = \{aww \mapsto \frac{1}{|w|} : w \in \mathcal{P} \text{ and } w_{|w|} \neq a\}$. Let $\tilde{C}(z)$ and $\tilde{D}(z)$ denote their generating series. By Proposition 3, the generating series of $\tilde{\mathcal{L}} = A^* \odot \tilde{\mathcal{C}} \odot A^*$ is $\tilde{L}(z) = \frac{\tilde{C}(z)}{(1-\ell z)^2}$. Moreover, it satisfies:

$$\tilde{L}(z) = \sum_{v\in A^*} \sum_{\substack{r\in \text{RUNS}(v) \\ r=(i,j,p)}} \frac{|r|-2p+1}{p} z^{|v|} = L_\gamma(z) - 2L_\rho(z) + \sum_{v\in A^*} \sum_{\substack{r\in \text{RUNS}(v) \\ r=(i,j,p)}} \frac{z^{|v|}}{p} \tag{6}$$

Let $\xi(v)$ be the sum of $\frac{1}{p}$ for every $(i,j,p) \in \text{RUNS}(v)$. Using exactly the same idea as in Section 5.1, its cumulative series is $L_\xi(z) = \frac{\tilde{C}(z)}{1-\ell z} + \frac{\tilde{D}(z)}{(1-\ell z)^2}$. Hence, Equation (6) rewrites

$$L_\gamma(z) = 2L_\rho(z) + \frac{\tilde{C}(z)-\tilde{D}(z)}{(1-\ell z)^2} - \frac{\tilde{C}(z)}{1-\ell z}.$$

Since the radius of convergence of both $\tilde{C}(z)$ and $\tilde{D}(z)$ is $1/\sqrt{\ell}$, the Simplified Transfer Theorem applies. We obtain that the expected value of γ asymptotically satisfies

$$\mathbb{E}_n[\gamma] \sim \left(\frac{2(\ell-1)}{\ell} P\left(\frac{1}{\ell^2}\right) + \frac{1}{\ell} Q\left(\frac{1}{\ell^2}\right) \right) n, \tag{7}$$

where the function $Q(z) = \int_0^z P(t)t^{-1}dt$ naturally appears when simplifying $\tilde{C}(\ell^{-1}) - \tilde{D}(\ell^{-1})$.

One can check that Equation (7) is just another formulation of Theorem 9. Indeed, we have

$$2\frac{\ell-1}{\ell} P\left(\frac{1}{\ell^2}\right) = \sum_{k\geq 1} \mu(k) \frac{2(\ell-1)}{\ell(\ell^{2k-1}-1)} = \sum_{k\geq 1} \mu(k) \frac{2(\ell-1)}{\ell^{2k}-\ell}.$$

And since everything is normally convergent,

$$\left(\frac{1}{\ell^2}\right) = \int_0^{1/\ell^2} P(t)t^{-1}dt = \int_0^{1/\ell^2} \sum_{k\geq 1} \frac{\mu(k)}{t} \frac{\ell\, t^k}{1-\ell\, t^k} dt = \sum_{k\geq 1} \mu(k) \int_0^{1/\ell^2} \frac{\ell\, t^{k-1}}{1-\ell\, t^k} dt.$$

Observe that the derivative of $t \mapsto -\log(1-\ell\, t^k)$ is $t \mapsto \frac{k\ell\, t^{k-1}}{1-\ell\, t^k}$. Thus

$$Q\left(\frac{1}{\ell^2}\right) = \sum_{k\geq 1} \frac{\mu(k)}{k} \log \frac{1}{1-\ell^{1-2k}} = \sum_{k\geq 1} \frac{\mu(k)}{k} \log \frac{\ell^{2k}}{\ell^{2k}-\ell}.$$

This gives the announced result.

6 Generalization to Memoryless Sources

In this section, we show how our formalism can be used to generalize the results to memoryless sources (see Section 2.1 for the definition). From now on, the alphabet is $A = \{a_1, \dots, a_\ell\}$ and we have a probability function p on A that charges at least two letters:[3] $p(a_i) < 1$ for every $i \in [\ell]$. Let $\vec{p}$ be the vector $\vec{p} = (p(a_1), \dots, p(a_\ell))$.

6.1 Multivariate Generating Series and Memoryless Sources

For $v \in A^*$ and $i \in [\ell]$, let $|v|_i$ denote the number of occurrences of the letter a_i in v. In our settings, multivariate generating series are formal power series on the formal variables z, u_1, $\dots$, u_ℓ. When needed, we will use the vector $\vec{u} = (u_1, \dots, u_\ell)$ to simplify the notations. For any positive integer k, let $\vec{u}^k$ denote the vector $(u_1^k, \dots, u_\ell^k)$, and let $N_k(\vec{u}) = u_1^k + \ldots + u_2^k$.

The *multivariate generating series* $L(z, \vec{u})$ of a language $\mathcal{L}$ is defined by

$$L(z, \vec{u}) := \sum_{v \in A^*} z^{|v|} \prod_{i=1}^{\ell} u_i^{|v|_i} = \sum_{n, k_1, \dots, k_\ell \geq 0} L(n, k_1, \dots, k_\ell) z^n u_1^{k_1} \cdots u_\ell^{k_\ell},$$

where $L(n, k_1, \dots, k_\ell)$ is the number of words of length n of $\mathcal{L}$ with exactly k_i occurrences of a_i, for every $i \in [\ell]$.

Multivariate generating series are widely use in combinatorics and analytic combinatorics. In particular, when the parameters controlled by the u_i's are additive, the symbolic method can be extended, giving efficient techniques to build the series. We refer the interested reader to [6, Ch. III] for more information on this topic. Interestingly, we can also extend our framework to multivariate generating series, when the u_i's are associated with the number of occurrences of the letters. First, the definition is extended to a weighted set $\mathcal{M}$ by weighting each word: $M(z, \vec{u}) := \sum_{v \in A^*} \mathcal{M}(v)\, z^{|v|} \prod_{i=1}^{\ell} u_i^{|v|_i}$. Proposition 3 is then directly generalized: if $\mathcal{M}$ and $\mathcal{N}$ are two weighted sets then the multivariate series of $\mathcal{M} \oplus \mathcal{N}$ is $M(z, \vec{u}) + N(z, \vec{u})$, the one of $\mathcal{M} \odot \mathcal{N}$ is $M(z, \vec{u})N(z, \vec{u})$, and the one of $\mathcal{M}^\star$ is $\frac{1}{1 - M(z, \vec{u})}$.

The main reason to consider multivariate series is the following: if $L(z, \vec{u})$ is the series of a language $\mathcal{L}$, then if we instantiate every formal variable u_i with the value $p(a_i)$, which we simply write $L(z, \vec{p})$, then we obtain a univariate series such that $[z^n]L(z, \vec{p})$ is exactly the probability that a word of length n belongs to $\mathcal{L}$, for the memoryless model of probability p. Similarly, if the generating series $M(z)$ of the weighted set $\mathcal{M}$ is the cumulative generating series of a statistic χ (for the uniform distribution), then $\mathbb{E}_n[\chi] = [z^n]M(z, \vec{p})$ for the memoryless distribution of probability p. The proofs of these facts are completely straightforward. However, together with the symbolic method, this provides a useful framework to deal with statistics on random words for memoryless distributions.

As an example, let us consider our first introductory statistic, the number of occurrences of the pattern v in a word. We use the weighted set description $A^* \odot \{v \mapsto 1\} \odot A^*$. The multivariate series of $A^\star$ is $\frac{1}{1 - z\, N_1(\vec{u})}$, since it is the weighted star of A, whose multivariate series is $A(z, \vec{u}) = u_1 z + \ldots + u_\ell z = N_1(\vec{u})\, z$. The multivariate series of $\{v \mapsto 1\}$ is $V(z, \vec{u}) = z^{|v|} u_1^{|v|_1} \cdots u_\ell^{|v|_\ell}$. Hence, the multivariate series of the number of occurrences of v is $\frac{V(z, \vec{u})}{(1 - N_1(\vec{u})z)^2}$. For $\vec{u} = \vec{p}$, we have $N_1(\vec{p}) = 1$, since p is a probability, and $V(z, \vec{p}) = \mathbb{P}_p(v) z^{|v|}$, by definition of a memoryless model. Hence, the multivariate series for $\vec{u} = \vec{p}$ is equal to $\frac{\mathbb{P}_p(v)}{(1-z)^2}$. The Simplified Transfer Theorem yields that the expected number of occurrences of v in a word of length n is asymptotically $\mathbb{P}_p(v)n$, for this memoryless distribution.

[3] Everything is trivial if $p(a_i) = 1$ for some i, as the only word of A^n with positive probability is a_i^n.

6.2 Expected Number of Runs for Memoryless Sources

We start as in Section 5.1, and use the weighted set $(\mathcal{C} \odot A^*) \oplus (A^* \odot \mathcal{D} \odot A^*)$ to count the number of runs, with $\mathcal{C} = \{ww \mapsto 1 : w \in \mathcal{P}\}$ and $\mathcal{D} = \{aww \mapsto 1 : w \in \mathcal{P} \text{ and } w_{|w|} \neq a\}$. The associated multivariate series is therefore $L(z, \vec{u}) = \frac{C(z,\vec{u})}{1-N_1(\vec{u})z} + \frac{D(z,\vec{u})}{(1-N_1(\vec{u})z)^2}$, where $C(z, \vec{u})$ and $D(z, \vec{u})$ are the multivariate series of $\mathcal{C}$ and $\mathcal{D}$.

At this point we have to compute the multivariate generalization $P(z, \vec{u})$ of $P(z)$, the series of primitive words. We will also need to compute $P_i(z, \vec{u})$, the multivariate series of the primitive words that ends by a_i. This is done using Equation (1), which readily extends to multivariate series in our case, yielding

$$P(z, \vec{u}) = \sum_{k\geq 1} \frac{\mu(k) z^k N_k(\vec{u})}{1 - z^k N_k(\vec{u})} \text{ and } P_i(z, \vec{u}) = \sum_{k\geq 1} \frac{\mu(k) z^k u_i^k}{1 - z^k N_k(\vec{u})}.$$

Moreover, $C(z, \vec{u}) = P(z^2, \vec{u}^2)$ and it is easy to compute from $P_i(z, \vec{u})$ that

$$D(z, \vec{u}) = \sum_{i\in[\ell]} z v_i P_i(z^2, \vec{u}^2) = \sum_{k\geq 1} \mu(k) z^{2k+1} \frac{\sum_{i=1}^{\ell} v_i u_i^{2k}}{1 - N_{2k}(\vec{u}) z^{2k}}, \text{ with } v_i = \sum_{\substack{j\in[\ell] \\ j\neq i}} u_i.$$

This formula looks complicated, but it simplifies when evaluating it at $z = 1$, the dominant singularity, and at $\vec{u} = \vec{p}$. In particular, if $\vec{u} = \vec{p}$, then $v_i = 1 - p(a_i)$ and $\sum_{i=1}^{\ell} v_i u_i^k = N_k(\vec{p}) - N_{k+1}(\vec{p})$. Hence, applying the Simplified Transfer Theorem to the expression of $L(z, \vec{p})$ yields the following result.

▶ **Theorem 10.** *For the memoryless distribution of probability p, the expected number of runs in a random word of length n satisfies asymptotically*

$$\mathbb{E}_n[\rho] \sim D(1, \vec{p})n = \left(\sum_{k\geq 1} \mu(k) \frac{N_{2k}(\vec{p}) - N_{2k+1}(\vec{p})}{1 - N_{2k}(\vec{p})} \right) n.$$

7 Conclusions

As illustrated throughout this article, the framework we propose is quite useful to study some statistics on random words. We choose to focus on presenting the technique itself in this extended abstract, to try to convince the reader that it is a precious tool to estimate the expectation of various parameters on words.

Due to the lack of space, we only generalized the result on the expected number of runs to memoryless distributions, but the other theorems of Section 5 can also be extended in a similar way. Some other kinds of generalizations can also be obtained. For instance, the expected number of cubic-runs (runs of exponent at least 3) is asymptotically equivalent to $\frac{\ell-1}{\ell} P(\ell^{-3})n$, which can be obtained as in Section 5.1. More generally, all results can readily be generalized to k-runs. Other known statistics can be studied using this method: as a last example, the expected number of squares χ in a word, i.e. the number of factors of the form vv for nonempty v was studied in [3]. In our framework, this corresponds to the weighted set $\oplus_{v\in A+} A^* \odot \{vv \mapsto 1\} \odot A^*$, thus $L_\chi(z) = \frac{\ell z^2}{(1-\ell z)^2(1-\ell z^2)}$ and $\mathbb{E}_n[\chi] \sim \frac{n}{\ell-1}$.

A natural extension of this work would be to provide similar tools to deal with higher moments, in particular with the variance. However, what we did in this article is related to the linearity of the expectation, and the variance is not linear. To compute higher moments, we have to handle dependencies between runs in a word, which is much more complicated. It would also be interesting to revisit some other probabilistic studies of the literature, such as [9, 2, 10], to see if they can be included in the framework of sets of weighted words.

References

1 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *CoRR*, abs/1406.0263, 2014.

2 Manolis Christodoulakis, Michalis Christou, Maxime Crochemore, and Costas S. Iliopoulos. Abelian borders in binary words. *Discrete Applied Mathematics*, 171:141–146, 2014.

3 Manolis Christodoulakis, Michalis Christou, Maxime Crochemore, and Costas S. Iliopoulos. On the average number of regularities in a word. *Theoretical Computer Science*, 525:3–9, 2014.

4 Maxime Crochemore and Lucian Ilie. Maximal repetitions in strings. *Journal of Computer and Systems Sciences*, 74(5):796–807, 2008.

5 Maxime Crochemore, Lucian Ilie, and Liviu Tinta. The "runs" conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011.

6 Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2008.

7 Philippe Flajolet, Wojciech Szpankowski, and Brigitte Vallée. Hidden word statistics. *Journal of the ACM*, 53(1):147–183, 2006.

8 Frantisek Franek and Qian Yang. An asymptotic lower bound for the maximal number of runs in a string. *Intern. Journal of Foundations Computer Science*, 19(1):195–203, 2008.

9 Kimmo Fredriksson and Szymon Grabowski. Average-optimal string matching. *Journal of Discrete Algorithms*, 7(4):579–594, 2009.

10 Pawel Gawrychowski, Gregory Kucherov, Benjamin Sach, and Tatiana A. Starikovskaya. Computing the longest unbordered substring. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval – 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 246–257. Springer, 2015.

11 Amy Glen and Jamie Simpson. The total run length of a word. *Theoretical Computer Science*, 501:41–48, 2013.

12 Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the 1999 Symposium on Foundations of Computer Science (FOCS'99), New York (USA)*, pages 596–604, New-York, October 17-19 1999. IEEE Computer Society.

13 Kazuhiko Kusano, Wataru Matsubara, Akira Ishino, and Ayumi Shinohara. Average value of sum of exponents of runs in a string. *Intern. Journal of Foundations of Computer Science*, 20(06):1135–1146, 2009.

14 Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, and Ayumi Shinohara. New lower bounds for the maximum number of runs in a string. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2008, Prague, Czech Republic, September 1-3, 2008*, pages 140–145, 2008.

15 Simon J. Puglisi and Jamie Simpson. The expected number of runs in a word. *Australasian Journal of Combinatorics*, 42:45–54, 2008.

16 Simon J. Puglisi, Jamie Simpson, and William F. Smyth. How many runs can a string contain? *Theoretical Computer Science*, 401(1-3):165–171, 2008.

17 Wojciech Rytter. The number of runs in a string. *Information and Computation*, 205(9):1459–1469, 2007.

18 Jamie Simpson. Modified Padovan words and the maximum number of runs in a word. *Australasian Journal of Combinatorics*, 46:129–145, 2010.

Reconstruction of Trees from Jumbled and Weighted Subtrees

Dénes Bartha[1], Péter Burcsi[2], and Zsuzsanna Lipták[3]

1 **Dept. of Computer Algebra, Eötvös Loránd University, Budapest, Hungary**
`denesb@gmail.com`
2 **Dept. of Computer Algebra, Eötvös Loránd University, Budapest, Hungary**
`bupe@compalg.inf.elte.hu`
3 **Dip. di Informatica, University of Verona, Italy**
`zsuzsanna.liptak@univr.it`

Abstract

Let T be an edge-labeled graph, where the labels are from a finite alphabet Σ. For a subtree U of T, the *Parikh vector* of U is a vector of length $|\Sigma|$ which specifies the multiplicity of each label in U. We ask when T can be reconstructed from the multiset of Parikh vectors of all of its subtrees, or all of its paths, or all of its maximal paths. We consider the analogous problems for weighted trees. We show how several well-known reconstruction problems on labeled strings, weighted strings and point sets on a line can be included in this framework. We present reconstruction algorithms and non-reconstructibility results, and extend the polynomial method, previously applied to jumbled strings [Acharya *et al*, SIAM J on Discr. Math, 2015] and weighted strings [Bansal *et al*, CPM 2004], to deal with general trees and special tree classes.

1998 ACM Subject Classification F.2.2 [Nonnumerical Algorithms and Problems] Computations on discrete structures, G.2.2 [Graph Theory] Graph labeling, Trees

Keywords and phrases trees, paths, Parikh vectors, reconstruction problems, homometric sets, polynomial method, jumbled strings, weighted strings

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.10

1 Introduction

Let T be an unrooted tree T with labeled edges, where the labels come from a finite ordered alphabet Σ. For a subtree U of T, the *Parikh vector* of U is a vector of length $|\Sigma|$ which specifies the multiplicity of each label in U. If the labels are positive reals or integers, we refer to them as *weights*, and define the weight of U as the sum of weights of the edges in U. (It is common to refer to a subtree as *jumbled* if only its Parikh vector is known, and as *weighted* if only its weight is known.) Given a subtree property $\mathcal{A}$, we refer to the multiset of Parikh vectors of all subtrees with property $\mathcal{A}$ as $MP_{\mathcal{A}}(T)$, and to the multiset of weights of all subtrees with property $\mathcal{A}$ as $MW_{\mathcal{A}}(T)$. For example, $MW_{\text{PATH}}(T)$ is the multiset of path weights in a weighted tree T.

Consider the two edge-labeled trees in Fig. 1, with labels from the alphabet $\Sigma = \{a, b\}$. The two trees are non-isomorphic, but the multisets of Parikh vectors of their *subtrees* are the same, $MP_{\text{SUBTREE}}(T_1) = MP_{\text{SUBTREE}}(T_2)$, as can be easily checked. At the same time, the multisets of Parikh vectors of their *paths* are not the same, $MP_{\text{PATH}}(T_1) \neq MP_{\text{PATH}}(T_2)$, since, for instance, T_2 has a path with Parikh vector $(1, 3)$ and T_1 does not.

These multisets can be described with the help of polynomials. Let variable x represent label a, and variable y label b. Then the polynomial describing the *subtrees* of both T_1 and T_2

Figure 1 Two MP_{SUBTREE}-equivalent trees.

is $2x + 3y + 4xy + y^2 + x^2y + 3xy^2 + 2x^2y^2 + xy^3 + x^2y^3$, where the interpretation e.g. of the term $3xy^2$ is that there are 3 (the coefficient) subtrees that contain 1 letter a and 2 letters b (the exponents). In a similar way, polynomials can be used to describe the multisets of *paths* or of *maximal paths*. Moreover, they can be used to describe the *weights* of certain subtrees when the edges are labeled with positive integers. We will give more precise definitions later.

In this paper, we are interested in the following questions:

- **Computation:** How can we compute the polynomials describing the jumbled or weighted subtrees, paths, or maximal paths?
- **Reconstruction:** Can trees be uniquely reconstructed from the multiset of jumbled or weighted subtrees, paths, or maximal paths? I.e. are there non-isomorphic trees with the same multisets? – We split this problem into two sub-problems:
  1. **Large Unjumble:** Is the unlabeled tree (i.e., its topology) uniquely determined by the multiset of jumbled or weighted subtrees, paths, or maximal paths?
  2. **Small Unjumble:** Given the topology of the tree, is the labeling uniquely determined by the multiset of jumbled or weighted subtrees, paths, or maximal paths?

The method of using polynomials to describe multisets of Parikh vectors or of weights has been successfully applied in the past to strings. In [8] polynomials were used for representing the multiset of weights of substrings (there called *submasses*) of a weighted string, i.e. where each character is assigned a positive integer weight. Fast Fourier Transform was employed to compute this polynomial, and several algorithms were proposed for finding substrings with a given query weight, using this polynomial.

In [2] the authors describe a similar polynomial representation of the multiset of Parikh vectors of substrings, and study the class of strings having the same multiset (there called *confusable*), using algebraic methods based on this polynomial. The method was originally employed in [28] for the related *turnpike problem*: Given n unknown points on a line, reconstruct the positions of these points from the multiset of interpoint distances. Indeed, in [1], an algorithm was given for reconstruction of all confusable strings from the multiset of Parikh vectors of substrings, an adaptation of an algorithm given in [28]. Note that the turnpike problem itself can be viewed as a problem on an edge-weighted tree (a path), where the vertices are the points, the edges are weighted by the distances between consecutive points, and the input is the multiset of path weights.

In this paper, we show how the polynomial method can be extended to trees. But generalizing the substructure of *substring* to trees can result either in *subtrees*, or in *paths*. We show that the method works for both types of substructures, as well as for *maximal paths* (i.e. paths between leaves). Note that equivalence w.r.t. one does not imply equivalence w.r.t. the other.

In the case of strings, both for jumbled and for weighted substrings, the polynomial can be computed via convolution from a very easily computable polynomial with 0/1 coefficients

(called *generating polynomial* in [2] and *prefix polynomial* in [8]), essentially using the fact that the Parikh vector (resp. weight) of a substring is the difference of the Parikh vectors (resp. weights) of two prefixes. We show how to compute the polynomials for trees in a similar manner, recursively from the polynomials of subtrees, but using both multiplication and addition of polynomials. Since strings can be represented as edge-labeled paths, our framework encompasses the known results on strings. Of course, if the tree T is a path, then the multisets of subtrees and of paths coincide.

The related problem of *jumbled pattern matching*, finding one or all occurrences of substructures with a given Parikh vector, has been studied recently extensively on strings, most recently in [13, 5, 25, 4, 15, 22, 24, 7, 21, 27, 29, 12, 11]; and also on vertex-colored graphs and trees [20, 14, 17]. On graphs, the problem is also called *motif search*, and it is NP-hard to decide whether a match exists, even when G is a tree [26]. When the number of colors is constant, the problem is fixed-parameter tractable w.r.t. treewidth [20].

Note that the variant of our problem where the subtrees are restricted to maximal paths is closely related to the problem of *distance-based phylogenetic reconstruction*, see e.g. [16], where a distance matrix between the leaves of a tree is given, and the task is to reconstruct the tree. The problem there is well-understood: such a tree exists if the input matrix has a certain property (called *additivity*), and an efficient algorithm exists for reconstructing the tree [30], which runs in cubic time in the number of leaves. The difference here is that we are given the input numbers without assignment to the pairs of leaves.

Following [28], we call two weighted trees T_1 and T_2 *homometric* if the multisets of pairwise distances between vertices is the same for both trees, or equivalently in our terms, $MW_{\text{PATH}}(T_1) = MW_{\text{PATH}}(T_2)$. We note that even though trees, and more generally, graphs, do appear in the literature on homometric sets [19, 6], those papers consider homometric vertex sets within one tree rather than homometric pairs of trees, while the papers [18, 3] treat quite different problems from the present ones.

Most proofs are omitted due to space limitations, and will be included in the full version.

2 The polynomial representation of Parikh multisets and weight multisets

Let Σ be a finite alphabet with elements $a_1, a_2, \ldots, a_\sigma$. Consider the polynomial ring over the integers in σ indeterminates, i.e. $\mathbb{Z}[x_1, x_2, \ldots, x_\sigma]$. When the alphabet is binary, we will denote the indeterminates by x and y.

If we interpret a Parikh vector $(k_1, k_2, \ldots, k_\sigma)$ as a multidegree, we can assign to it the monomial $x_1^{k_1} x_2^{k_2} \cdots x_\sigma^{k_\sigma}$. Note that the total degree of the polynomial equals the sum of entries of the Parikh vector. A *multiset* of Parikh vectors can then be represented as the sum of the monomials of its elements; multiplicities become coefficients. The power of this viewpoint is that disjoint union of (multi)sets corresponds to the product of the two monomials associated to the sets.

If we work with weights rather than arbitrary labels, then a single indeterminate suffices: to a weighted edge e with weight $w(e)$, we associate the polynomial $x^{w(e)}$. If we have a set U of edges and take the product of the monomials corresponding to the elements, then we get $x^{\sum_{e \in U} w(e)}$. The primary focus of the present paper are Parikh multisets and weight multisets of a tree T obtained by taking Parikh vectors or weights of subtrees of T satisfying some condition.

▶ **Definition 1.** Let T be a tree and $\mathcal{A}$ be a property of subtrees. Let the edges of T be labeled by an σ-element alphabet Σ. The $MP_{\mathcal{A}}$-polynomial of T, denoted by $f_{\mathcal{A}}(T)$ is

the σ-variable polynomial associated to the multiset of Parikh vectors of all subtrees of T satisfying condition $\mathcal{A}$.

▶ **Definition 2.** Let T be a tree and $\mathcal{A}$ be a property of subtrees. Let the edges of T be weighted by positive integers. The $MW_{\mathcal{A}}$-polynomial of T, denoted by $g_{\mathcal{A}}(T)$ is the 1-variable polynomial associated to the multiset of weights of all subtrees of T satisfying condition $\mathcal{A}$.

The main reason for using polynomials to represent multisets is that we have additional algebraic structure, while all information about the multiset is still preserved. This is a crucial property used throughout (sometimes implicitly), so we state it as an observation.

▶ **Observation 3.** *Let T_1, T_2 be trees and $\mathcal{A}$ a subtree property. Then $MP_{\mathcal{A}}(T_1) = MP_{\mathcal{A}}(T_2)$ if and only if $f_{\mathcal{A}}(T_1) = f_{\mathcal{A}}(T_2)$. Similarly $MW_{\mathcal{A}}(T_1) = MW_{\mathcal{A}}(T_2)$ if and only if $g_{\mathcal{A}}(T_1) = g_{\mathcal{A}}(T_2)$.*

The following observation is also straightforward and means that $MP_{\mathcal{A}}(T)$ contains all the information for computing $MW_{\mathcal{A}}(T)$.

▶ **Observation 4.** *If the letters of the alphabets are positive integers, then they can be interpreted as weights. Then the $MW_{\mathcal{A}}$-polynomial of a tree can be calculated from the $MP_{\mathcal{A}}$-polynomial by substituting x^{a_i} into the variable x_i.*

▶ **Example 1.** *Let $\mathcal{A} = PATH$. Let $\Sigma = \{a, b\}$ and let the indeterminate x correspond to a, and y to b. The tree T_1 in Figure 1 has the MP_{PATH}-polynomial $2x+3y+4xy+y^2+x^2y+2xy^2+2x^2y^2$, while the MP_{PATH}-polynomial of T_2 is $2x+3y+4xy+y^2+x^2y+2xy^2+x^2y^2+xy^3$. If we let $a = 3$ and $b = 2$, then the MW_{PATH}-polynomial of T_1 is $3t^2+2t^3+t^4+4t^5+2t^7+t^8+2t^{10}$. This is obtained from its MP_{PATH}-polynomial by letting $x = t^3$ and $y = t^2$. (We used a new letter t to avoid confusion.)*

In what follows, we discuss how $MP_{\mathcal{A}}$-polynomials and $MW_{\mathcal{A}}$-polynomials of a tree can be computed. We will restrict our attention to the case of $\mathcal{A} =$ SUBTREE, where all subtrees are considered, $\mathcal{A} =$ PATH, where only paths between pairs of vertices are considered and $\mathcal{A} =$ MAXPATH, where only maximal paths are considered. The theorems will be stated for $MP_{\mathcal{A}}$-polynomials, but are valid in the same form for $MW_{\mathcal{A}}$-polynomials.

Unless otherwise specified, the labels or weights are always on the edges rather than the vertices. The computation methods for vertex labeled and vertex weighted graphs are obtained by adapting the computations, which we will not state as separate theorems. Our examples of $MP_{\mathcal{A}}$-equivalent families are proved using the polynomial method. We present recursive computation methods for the three kinds of subtree properties in the following sections (the base cases for the recursion are left to the reader).

To conclude this section, we propose a new algorithmic application of $MP_{\mathcal{A}}$-polynomials (resp. $MW_{\mathcal{A}}$-polynomials) for randomized testing of $MP_{\mathcal{A}}$-equivalence (resp. $MW_{\mathcal{A}}$-equivalence) of trees. The method is based on randomized equality testing for polynomials using the Schwartz-Zippel lemma [32, 34]. The computation methods presented later all allow an efficient substitution into the polynomials, even in the case when we consider subtrees, where the size of the MP_{SUBTREE}-set and thus the number of coefficients of the polynomial can be exponential in the input. For the substitution we do not need the sequence of coefficients, we can use the recursive methods for evaluating the polynomial. Finally note that using modular arithmetic, calculations can be further sped up.

3 Subtrees

3.1 Computation of $f_{\text{SUBTREE}}(T)$

We first consider the case when all subtrees are considered in the Parikh multiset or the weight multiset. Although we work on free trees (i.e. unrooted trees), for the computations it is convenient to consider rooted trees. We root the tree T in an arbitrary vertex v and define an auxiliary polynomial $r(T, v)$, called the rooted MP-polynomial of T with root v, as the polynomial representing the Parikh multiset of all subtrees containing v. We have the following theorem.

▶ **Theorem 5.** *Let T be a rooted tree with root v. Let $v_1, v_2, \ldots, v_k$ be the children of v. Denote the subtrees rooted at v_i by T_i for $i = 1, \ldots, k$. Denote the index in Σ of the label on the edge connecting v and v_j by l_j. We have the following equations.*

$$r(T, v) = \prod_{j=1}^{k} (1 + x_{l_j} \cdot r(T_j, v_j)) \quad \text{and} \quad f(T) = r(T, v) + \sum_{j=1}^{k} f(T_j)$$

Note that Theorem 5 generalizes the computation of MP-polynomials or MW-polynomials of strings presented in e.g. [8, 28, 2] since a string can be interpreted as an edge-labeled path. The theorem also generalizes the subtree size multiset presented in [9].

3.2 Reconstructibility – Large Unjumble

For a general labeled tree T, one can ask if the unlabeled version of the tree (the topology) can be uniquely reconstructed from $MP_{\text{SUBTREE}}(T)$ or $MW_{\text{SUBTREE}}(T)$. This is already impossible from $MP_{\text{SUBTREE}}(T)$ for a trivial (i.e. one-element) alphabet, which also implies that $MW_{\text{SUBTREE}}(T)$ does not determine the isomorphism class of the unlabeled tree either.

If one puts the same label (resp. weight) on each edge, then the Parikh vector (resp. weight) of a subtree simply counts the number of edges in that subtree. It was proved in [9] that knowing the number of subtrees with k edges for all k, that is, in our terms, knowing $MP_{\text{SUBTREE}}(T)$ for one-letter alphabets is not generally enough for unique reconstruction of the tree up to isomorphism.

▶ **Proposition 6** ([9]). *Let $\Sigma = \{1\}$. There exist infinitely many pairs of trees T_1, T_2, such that if we label each edge with the only element of Σ, then $MP_{\text{SUBTREE}}(T_1) = MP_{\text{SUBTREE}}(T_2)$ and $MW_{\text{SUBTREE}}(T_1) = MW_{\text{SUBTREE}}(T_2)$.*

In the positive direction, we mention the following reconstructibility result from the same paper. A *spider* is a tree with one vertex of degree at least 3 and all others with degree at most 2 (called *star-like trees* in that paper).

▶ **Theorem 7** ([9]). *Let $|\Sigma| = 1$, and T_1, T_2 be two edge-labeled spiders with labels from Σ. If $MP_{\text{SUBTREE}}(T_1) = MP_{\text{SUBTREE}}(T_2)$, then T_1 and T_2 are isomorphic.*

3.3 Reconstructibility – Small Unjumble

When the alphabet is non-trivial, there are several non-isomorphic labelings of a typical tree. We consider reconstructibility of the labels for a fixed unlabeled tree. Note that the problem of reconstructing a string from its substring compositions [2] is a special case: a string can be represented as a path of equal length where the edge labels correspond to individual characters in the string.

The problem of reconstructing a 1-dimensional point set from interpoint distances considered in [28] is also a special case of the reconstruction of a tree from $MW(T)$: the weights are the distances between neighboring points on a line. Since every subtree of a path is a (sub)path, this remark also applies for reconstructibility from path Parikh vectors (resp weights), addressed in the following section. The above two problems can also be reduced to the case of vertex labeled paths.

We give non-reconstructibility examples for trees that are not a path. The smallest pair of non-isomorphic MP_{SUBTREE}-equivalent edge labeled trees are on six vertices.

▶ **Example 2.** *Let P be a path of length 4, whose vertices are called $v_1, v_2, \ldots, v_5$ and the edges $v_1v_2, \ldots$ are labeled with a, b, a, b. Construct T_1 by attaching a 6th vertex v_6 to v_4 with an edge labeled by b. Construct T_2 from P by attaching a 6th vertex to v_2 with an edge labeled by b. See the example in Fig. 1.*

It is also possible to attach a larger tree instead of the sixth vertex, which gives larger examples of MP_{SUBTREE}-equivalent pairs. We remark that the smallest such example for vertex labeled trees is on 7 vertices. We also give a construction that yields an infinite family of MP_{SUBTREE}-equivalent examples (similar constructions work for vertex labeled trees).

▶ **Proposition 8.** *Let s_1 and s_2 be two MP_{SUBTREE}-equivalent strings of length k over a binary alphabet Σ_1. Create two MP_{SUBTREE}-equivalent edge-labeled paths P_1, P_2 by using characters of the strings as labels. Let U be an edge labeled rooted tree with labels from a disjoint alphabet Σ_2. Create T_j ($j = 1, 2$) from P_j by joining $k+1$ copies of U to each vertex of P_j, identifying the vertex on the path and the root of U. Then T_1 and T_2 are not isomorphic as labeled trees, but $MP_{\text{SUBTREE}}(T_1) = MP_{\text{SUBTREE}}(T_2)$.*

Finally, we present a result stating that, unsurprisingly, MP-equivalence does not generally follow from MW-equivalence, already for 2-letter alphabets. We have an infinite family already for paths.

▶ **Proposition 9.** *Let $k \leq n$ an integer, $\Sigma = \{1, 2\}$. Let P_1 be a path of length $14 + 5k$, edge labeled with elements of the sequence $s_1 = 21211112222122(12122)^k$. Let P_2 be a path of length 14, edge labeled with elements of the sequence $s_2 = 22111121222212(12212)^k$. Then $MP_{\text{SUBTREE}}(P_1) \neq MP_{\text{SUBTREE}}(P_2)$, but $MW_{\text{SUBTREE}}(P_1) = MW_{\text{SUBTREE}}(P_2)$.*

4 Paths

4.1 Computation of $f_{\text{PATH}}(T)$

Let $f(T) = f_{\text{PATH}}(T)$ be the MP_{PATH}-polynomial of T. Root T is an arbitrary vertex v. We denote by $r(T, v)$ the polynomial corresponding to all paths at least one of whose endpoints is v. We include 0-length paths in the computation, since it makes the formulae simpler, this adds a constant n (the number of vertices) to the polynomial.

▶ **Theorem 10.** *Let T be a rooted tree with root v. Let $v_1, v_2, \ldots v_k$ be the children of v in T. Denote the subtrees rooted at v_1 (resp. v_2 etc.) by T_1 (resp. T_2 etc.). Denote the index in Σ of the label on the edge connecting v and v_j by l_j. We have the following equalities:*

$$r(T, v) = 1 + \sum(x_{l_j} \cdot r(T_j, v_j)), f(T) = r(T, v) + \sum_{j=1}^{k} f(T_j) + \sum_{1 \leq i < j \leq k} (x_{l_i} x_{l_j} r(T_i, v_i) r(T_j, v_j))$$

Proof. For the statement on r note that a path starting in v either stops there immediately, or contains exactly one of the v_j and thus a path from v_j to a vertex of T_j as a subpath. To understand the identity for f, observe that a path in T either contains v as one of its endpoints or is entirely contained in one of the T_j, or else it is the union of two paths which both have v as one endpoint and their respective other endpoints in distinct T_i and T_j. ◀

4.2 Reconstructibility – Large Unjumble

For a general labeled tree T, one can ask if the unlabeled version of the tree can be uniquely reconstructed from $MP_{\text{PATH}}(T)$ or $MW_{\text{PATH}}(T)$. We show that this is already impossible from $MP_{\text{PATH}}(T)$ for a one-element alphabet, which also implies that $MW_{\text{PATH}}(T)$ does not determine the isomorphism class of the unlabeled tree either.

In the following, we give infinitely many examples of pairs of unlabeled trees that are homometric. This can be considered as a special case of MP_{PATH}-equivalence (resp. MW_{PATH}-equivalence) when $|\Sigma| = 1$ (resp. we use the same weight everywhere).

▶ **Proposition 11.** *For $n \geq 11$ and odd, let T_1 be a tree constructed from a 5-star by adding respectively $1, 1, (n-5)/2$ and $(n-9)/2$ new vertices joined to the star's leaves, obtaining a tree on n vertices. Construct T_2 similarly, by adding 0, 2, $(n-7)/2$ and $(n-7)/2$ new vertices adjacent to the star's leaves. Then T_1 and T_2 are homometric but are not isomorphic.*

For $n \geq 12$ and even, let T_1 be a tree constructed from a 6-star by adding respectively $1, 1, 1$, $(n-6)/2$ and $(n-10)/2$ new vertices to the star's leaves, obtaining a tree on n vertices. Construct T_2 similarly, but add $0, 1, 2$, $(n-8)/2$ and $(n-8)/2$ new vertices. Then T_1 and T_2 are homometric but are not isomorphic.

We remark that all one has to do is check the number of paths of length $1, 2, 3$ and 4 since the constructed trees have diameter 4. The calculation is straightforward, and the idea behind it is that if we add k_1, k_2, k_3 and k_4 vertices to the 5-star as above, then the number of 1-paths (resp. 2-paths, 3-paths and 4-paths) is already determined by their sum and the sum of their squares, and these values are identical for the two trees. One can compose such trees by solving instances of the Prouhet-Tarry-Escott problem, see e.g. [10], Chap. 11.

4.3 Reconstructibility – Small Unjumble

We now turn to the problem of unique reconstructibility of the labeling, once the unlabeled version of the tree is known. Again, if we take the viewpoint of strings being (either edge or vertex) labeled graphs, then this problem contains as a special case the problem of string reconstructibility from MW_{PATH} or MP_{PATH}. We thus focus on reconstructibilty for other trees. First remark that the contruction in Proposition 8 also yields infinitely many MP_{PATH}-equivalent pairs of non-isomorphic trees.

We now give a family of pairs that are *vertex labeled* PM-equivalent trees.

▶ **Proposition 12.** *Let $k \geq 1$ an integer. Let P_{base} be a path of length 3 with alternating vertex labels $0, 1, 0, 1$, and P_{k-1} be a path on k vertices, al labeled by 0. Construct T_1 by attaching two copies of P_{k-1} to P_{base} with two edges: one is attached to the leaf with 0 label, and the other to the neighboring vertex on P_{base}. We get a tree on $2k+4$ vertices. The construction of T_2 is similar, but P_{base} is reversed. Then T_1 and T_2 are two different labelings of the same tree, and $MP_{\text{PATH}}(T_1) = MP_{\text{PATH}}(T_2)$.*

Class sizes for MP-equivalence. For paths, the size of an equivalence class of MW_{PATH}-equivalent paths (resp. MP_{PATH}-equivalent paths) is always a power of 2, as it was proved in [28] (resp. [2]). This result no longer holds for other classes of trees, as illustrated by the example below.

▶ **Example 3.** *Let T be a spider on 11 vertices with 5 legs of length 2. Then the following 3 weightings of T form an MP_{PATH}-equivalence class of size 3 (we give the weighting as 5-tuples of weight pairs on the legs from the center outwards).* $T_1 : [1,3],[2,3],[3,5],[4,1],[6,1]$, $T_2 : [1,3],[2,5],[3,1],[5,1],[5,3]$, $T_3 : [1,5],[2,1],[4,1],[4,3],[5,3]$.

Finally note that Proposition 9 also applies for $\mathcal{A} = \text{PATH}$.

5 Maximal paths

5.1 Computation of $f_{\text{MAXPATH}}(T)$

Let $f(T) = f_{\text{MAXPATH}}(T)$ be the MP_{MAXPATH}-polynomial of T. Let $r(T,v)$ denote the MP-polynomial corresponding to all paths with one endpoint in v and another one in a leaf. Finally, let $t(T,v)$ be the MP-polynomial for all maximal paths that have v as one of their endpoints. Note that $t(T,v) = 0$ if v is not a leaf in T.

▶ **Theorem 13.** *Let T be a rooted tree with root v, and let $v_1, v_2, \dots v_k$ be the children of v in T. Denote the subtrees rooted at v_1 (resp. v_2 etc.) by T_1 (resp. T_2 etc.). Denote the index in Σ of the label on the edge connecting v and v_j by l_j. We have the following equalities:*

$$
\begin{aligned}
r(T,v) &= \sum (x_{l_j} \cdot r(T_j, v_j)) \\
f(T) &= t(T,v) + \sum_{j=1}^{k} (f(T_j) - t(T_j, v_j)) + \sum_{i<j} (x_{l_i} x_{l_j} r(T_i, v_i) r(T_j, v_j)) \\
t(T,v) &= \begin{cases} r(T,v) & \text{if } k{=}1 \\ 0 & \text{if } k{>}1 \end{cases}
\end{aligned}
$$

5.2 Reconstructibility – Small Unjumble

We only consider reconstructibility for weighted graphs. Let us fix the topology of T as an n-star. We have the following reconstructibility result for edge-weighted n-stars.

▶ **Theorem 14.** *Let T_1 and T_2 be two n stars s.t. $n-1$ is not a power of 2. Then $MW_{\text{MAXPATH}}(T_1) = MW_{\text{MAXPATH}}(T_2)$ implies that T_1 and T_2 are isomorphic as edge weighted trees. If $n = 2^k + 1$ for some $k \leq 0$, then there are non-isomorphic edge labeled n-stars that are MW_{MAXPATH}-equivalent.*

The theorem is an easy consequence of Theorem 1 and Theorem 2 from [33], about the reconstructibility of numbers from the multiset of their pairwise distances. These two theorems are also proved in [23, 31] using the polynomial representation of sumsets, which is more in the spirit of the present paper. To see how it follows, simply observe that the weights of maximal paths are the pairwise sums of edge labels.

6 Reconstruction Algorithms

In this section, we treat reconstruction of edge-labeled trees from *weighted paths*, where the topology of the tree is given (Small Unjumble). Note that we assume that $\mathcal{S}$ is given sorted.

First let us note that for the case where T is a star, a simple greedy algorithm will solve the problem exactly in time loglinear in the input size $|\mathcal{S}| = \binom{n}{2}$. Denote by X the multiset of weights of the edges, then $MW_{\text{PATH}}(T) = X \cup (X + X)$ (where by $X + X$ we denote the multiset of sums of two elements from the multiset X). Clearly, the two smallest numbers in $\mathcal{S}$ are necessarily in X, which means that their sum is necessarily in $X + X$. The algorithm starts with an empty X, iteratively chooses the smallest remaining number in $\mathcal{S}$, adds it to X, and eliminates it and its sums with those already in X from $\mathcal{S}$. We touch each of the $\binom{n}{2}$ input numbers exactly once; getting the next smallest one takes constant time, while finding the corresponding elements from $X + X$ takes $\log n$ time each.

▶ **Example 4.** *Let T be a 6-star, i.e. $|V(T)| = 6$ with one vertex of degree 5 and 5 leaves, and let $\mathcal{S} = \{2, 3, 5, 5, 7, 8, 9, 10, 11, 12, 12, 13, 14, 15, 19\}$. Necessarily $2, 3 \in X$, and this eliminates also $5 = 2 + 3$ from the input set. The next remaining smallest number is 5: this must again be an edge label, thus $5 \in X$, eliminating $7 = 5 + 2$ and $8 = 5 + 3$ from our input set. Continuing, we get that $9 \in X$, eliminating $11, 12, 14$, and finally, that $10 \in X$, eliminating $12, 13, 15, 19$. So we see that the 5 edges are labeled with $2, 3, 5, 9$, and 10 respectively.*

In particular, if T is a star, then if there is a solution, it is necessarily unique. Thus we have proved the following:

▶ **Proposition 15.** *If T is a star, then the Greedy Algorithm correctly reconstructs its labeling from $MW_{\text{PATH}}(T)$ in time $O(n^2 \log n)$. Moreover, for any instance $\mathcal{S}$, either $\mathcal{S}$ is uniquely reconstructable, or there is no solution.*

Now let's turn to a general tree topology. In the following we will generalize the algorithm given in [28] for the turnpike problem to any tree. To this end, we define the *path poset* of a tree T as the set of all paths in T, together with the inclusion order. We give an example below (Ex. 5). Note that the input $MW_{\text{PATH}}(T)$ consists precisely of the weights of all elements of the path poset. So the task is to fill in the values from $\mathcal{S}$ into the path poset. The following is immediate:

▶ **Lemma 16.** *For any tree T, the path poset of T is exactly the union of the path posets of its maximal paths.*

▶ **Example 5.** *Let T be as in Fig. 2, input $\mathcal{S} = \{1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 6, 7, 8, 9, 11\}$. In the same figure, we show the three pyramids with the unique solution (up to exchanging the labels of d and e).*

Following [28], we will refer to the above representation of the values of the path poset of a maximal path as a *pyramid*. If $\pi = (v_1, \ldots, v_s)$ is a maximal path in T, then in its pyramid Δ, row k will hold all values of subpaths of π of length k. Let us refer to d_{ij} as the sum of the weights on the path from v_i to v_j. As was shown in [28], the following relationships hold within one pyramid:

▶ **Lemma 17** ([28])**.** *$d_{ij} + d_{k\ell} = d_{i\ell} + d_{kj}$ for $1 \le i \le k \le \ell \le j$.*

This property is then used in [28] for a backtracking algorithm which takes the next largest remaining value, guesses its position in the pyramid, and fills in all other values which are implied by it. When a choice implies a value not present in the input, the algorithm backtracks. We, however, need to fill in all pyramids concurrently. For this, the following lemma will be useful. We omit the proof for lack of space.

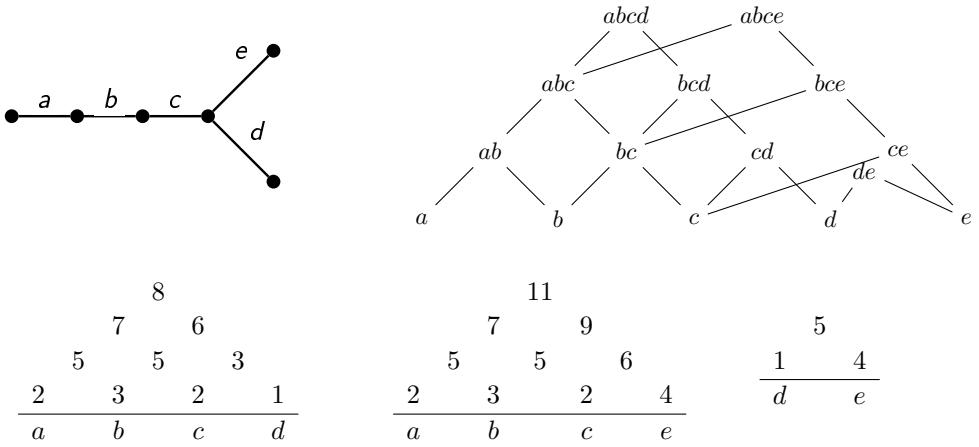


Figure 2 Example 5: A tree, its path poset, and the path posets of its three maximal paths (in the latter we omit the edges for clarity), with the values of the input set filled in.

▶ **Lemma 18.** *Let $\pi = (v_1, \ldots, v_r)$ and $\pi' = (u_1, \ldots, u_{r'})$ be two maximal paths in T with non-empty intersection ρ. Let Δ be the pyramid for π, with entries d_{ij}, and Δ' the pyramid for π', with entries d'_{ij}. If $\rho = (v_i, \ldots, v_{i+\ell}) = (u_{i'}, \ldots, u_{i'+\ell})$, then the following relationships hold between Δ and Δ':*

1. *for $k \le i, k' \le i'$: $d_{k,i+s} - d'_{k',i'+s} = d_{k,i+t} - d'_{k',i'+t}$ for all $0 \le s, t \le \ell$, and*
2. *for $k \ge i + \ell, k' \ge i' + \ell$: $d_{i+s,k} - d'_{i'+s,k'} = d_{i+t,k} - d'_{i'+t,k'}$ for all $0 \le s, t \le \ell$.*

Our algorithm proceeds as follows. In each step, it takes the next largest value still in $\mathcal{S}$ and places it in one of the maximal free places, i.e. in a free place that has no larger free place in any of the pyramids. It then fills in all implied positions according to Lemma 17 and 18. If at some point it encounters a value not present among the yet unused values, it backtracks. For example, in Example 5, for the first value 11 there are three possible choices, namely the tops of the three pyramids. Say we have already placed values 11 and 9 in their respective places as in the final solution. Now placing 8 on the top of the first pyramid will force the difference for all values on the right sides of the first and second pyramids to be 3, an application of Lemma 17.

▶ **Lemma 19.** *Every maximal free place is either on top of a pyramid, or on the side of a pyramid.*

▶ **Theorem 20.** *There is a $O((2\Gamma)^{(\Gamma+n)} n^2 \log n)$ algorithm for finding all possible labelings of a given tree T from the multiset of $\binom{n}{2}$ path weights, where n is the number of vertices of T, and Γ the number of maximal paths in T.*

Although the algorithm has exponential running time, it compares well to the simple exhaustive search if the number of leaves is small, since trying all possible labelings of the edges would give $O(n^{2n}/2^n)$ running time. Note that parameter Γ is quadratic in the number of leaves. So essentially the algorithm performs well on trees which are close to strings, and badly on trees that are close to stars, i.e. have many leaves. Indeed, as can be seen, the Greedy algorithm for stars applies the opposite strategy, namely filling in the path poset from below; this makes sense when the higher levels are more populous than the lower levels, while starting from above is appropriate when the form is pyramid-like. Moreover, the analysis is very pessimistic and does not so far take advantage of the improvements given by the

pruning due to Lemmas 17 and 18. In practice, we expect that many branches will be pruned by these implications. For the special case of the turnpike problem, if we consider random instances then incorrect branches are pruned almost immediately, see [28].

7 Conclusion and Open Problems

Our reconstruction algorithm is purely combinatorial, and it seems a challenging problem to find a reconstruction algorithm based on MP-polynomials, similar to the ones presented in [28, 2]. We would also be interested in proving further results about unique reconstructibility with algebraic techniques.

Another intriguing task is connecting the Large Unjumble Problem for weighted maximal paths to the distance-based phylogeny problem: Note that if we had an assignment of the input numbers to the Γ leaf pairs, then a reconstruction, if it exists, is unique, and can be found in $O(\Gamma^{3/2})$ e.g. using the Neighbor Joining algorithm [30] (or it can be shown that no such reconstruction exists).

Further open problems include the complexity status of the reconstruction problems introduced, in particular in which of the cases Large Unjumble is computationally hard.

References

1 Jayadev Acharya, Hirakendu Das, Olgica Milenkovic, Alon Orlitsky, and Shengjun Pan. Quadratic-backtracking algorithm for string reconstruction from substring compositions. In *2014 IEEE Int. Symp. on Information Theory (ISIT 2014)*, pages 1296–1300, 2014. doi:10.1109/ISIT.2014.6875042.

2 Jayadev Acharya, Hirakendu Das, Olgica Milenkovic, Alon Orlitsky, and Shengjun Pan. String reconstruction from substring compositions. *SIAM J. Discrete Math.*, 29(3):1340–1371, 2015. doi:10.1137/140962486.

3 Tatsuya Akutsu, Daiji Fukagawa, Jesper Jansson, and Kunihiko Sadakane. Inferring a graph from path frequency. *Discrete Applied Mathematics*, 160(10-11):1416–1428, 2012. doi:10.1016/j.dam.2012.02.002.

4 Amihood Amir, Ayelet Butman, and Ely Porat. On the relationship between histogram indexing and block-mass indexing. *Philosophical Transactions of The Royal Society A: Mathematical Physical and Engineering Sciences*, 372(2016), 2014. doi:10.1098/rsta.2013.0132.

5 Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In *41st Int. Coll. on Automata, Languages, and Programming (ICALP 2014)*, volume 8572 of *Lecture Notes in Computer Science*, pages 114–125. Springer, 2014. doi:10.1007/978-3-662-43948-7_10.

6 Maria Axenovich and Lale Özkahya. On homometric sets in graphs. *Electronic Notes in Discrete Mathematics*, 38:83–86, 2011. doi:10.1016/j.endm.2011.09.014.

7 Golnaz Badkobeh, Gabriele Fici, Steve Kroon, and Zsuzsanna Lipták. Binary Jumbled String Matching for Highly Run-Length Compressible Texts. *Information Processing Letters*, 113:604–608, 2013. doi:10.1016/j.ipl.2013.05.007.

8 Nikhil Bansal, Mark Cieliebak, and Zsuzsanna Lipták. Efficient algorithms for finding submasses in weighted strings. In *Proc. of the 15th Ann. Symp. on Combinatorial Pattern Matching (CPM 2004)*, volume 3109 of *Lecture Notes in Computer Science*, pages 194–204. Springer, 2004. doi:10.1007/978-3-540-27801-6_14.

9 Dénes Bartha and Péter Burcsi. Reconstructibility of trees from subtree size frequencies. *Stud. Univ. Babeş-Bolyai Math.*, 59:435–442, 2014.

10 Peter B. Borwein. *Computational excursions in analysis and number theory*. CMS books in mathematics. Springer, New York, Berlin, Heidelberg, 2002.

11 Péter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Algorithms for Jumbled Pattern Matching in Strings. *Int. Journal of Foundations of Computer Science*, 23:357–374, 2012. doi:10.1142/S0129054112400175.

12 Péter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On Approximate Jumbled Pattern Matching in Strings. *Theory of Computing Systems*, 50:35–51, 2012. doi:10.1007/s00224-011-9344-5.

13 Timothy M. Chan and Moshe Lewenstein. Clustered integer 3SUM via additive combinatorics. In *Proc. of 47th Annual ACM Symposium on Theory of Computing (STOC 2015)*, pages 31–40, 2015. doi:10.1145/2746539.2746568.

14 Ferdinando Cicalese, Travis Gagie, Emanuele Giaquinta, Eduardo Sany Laber, Zsuzsanna Lipták, Romeo Rizzi, and Alexandru I. Tomescu. Indexes for jumbled pattern matching in strings, trees and graphs. In *20th Int. Symp. on String Processing and Information Retrieval (SPIRE 2013)*, volume 8214 of *Lecture Notes in Computer Science*, pages 56–63. Springer, 2013. doi:10.1007/978-3-319-02432-5_10.

15 Ferdinando Cicalese, Eduardo Sany Laber, Oren Weimann, and Raphael Yuster. Approximating the maximum consecutive subsums of a sequence. *Theoret. Comput. Sci.*, 525:130–137, 2014. doi:10.1016/j.tcs.2013.05.032.

16 Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme J. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. CUP, 1998.

17 Stephane Durocher, Robert Fraser, Travis Gagie, Debajyoti Mondal, Matthew Skala, and Sharma V. Thankachan. Indexed geometric jumbled pattern matching. In *Proc. of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*, volume 8486 of *Lecture Notes in Computer Science*, pages 110–119. Springer, 2014. doi:10.1007/978-3-319-07566-2_12.

18 Tomás Feder and Rajeev Motwani. On the graph turnpike problem. *Inf. Process. Lett.*, 109(14):774–776, 2009. doi:10.1016/j.ipl.2009.03.024.

19 Radoslav Fulek and Slobodan Mitrovic. Homometric sets in trees. *Eur. J. Comb.*, 35:256–263, 2014. doi:10.1016/j.ejc.2013.06.008.

20 Travis Gagie, Danny Hermelin, Gad M. Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. *Algorithmica*, 73(3):571–588, 2015. doi:10.1007/s00453-014-9957-6.

21 Emanuele Giaquinta and Szymon Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14–16):538–542, 2013. doi:10.1016/j.ipl.2013.04.013.

22 Danny Hermelin, Gad M. Landau, Yuri Rabinovich, and Oren Weimann. Binary jumbled pattern matching via all-pairs shortest paths. *CoRR*, abs/1401.2065, 2014. URL: http://arxiv.org/abs/1401.2065.

23 Ross Honsberger. *In Polya's Footsteps: Miscellaneous Problems and Essays (Dolciani Mathematical Expositions)*. The Mathematical Association of America, October 1997.

24 Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In *21st Annual European Symposium on Algorithms (ESA 2013)*, volume 8125 of *Lecture Notes in Computer Science*, pages 625–636. Springer, 2013. doi:10.1007/978-3-642-40450-4_53.

25 Eduardo Laber, Wilfredo Bardales, and Ferdinando Cicalese. On lower bounds for the maximum consecutive subsums problem and the $(min, +)$-convolution. In *Proceedings of the 2013 IEEE Int. Symp. on Information Theory (ISIT 2013)*. IEEE, 2014.

26 Vincent Lacroix, Cristina G. Fernandes, and Marie-France Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):360–368, 2006. doi:10.1109/TCBB.2006.55.

27 Lap-Kei Lee, Moshe Lewenstein, and Qin Zhang. Parikh matching in the streaming model. In *19th Int. Symp. on String Processing and Information Retrieval (SPIRE 2012)*, volume 7608 of *Lecture Notes in Computer Science*, pages 336–341. Springer, 2012. doi:10.1007/978-3-642-34109-0_35.

28 Paul Lemke, Steven S. Skiena, and Warren D. Smith. *Discrete and Computational Geometry: The Goodman-Pollack Festschrift*, chapter Reconstructing Sets From Interpoint Distances, pages 597–631. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. doi:10.1007/978-3-642-55566-4_27.

29 Tanaeem M. Moosa and M. Sohel Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discr. Algorithms*, 10:5–9, 2012. doi:10.1016/j.jda.2011.08.003.

30 N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol Biol Evol*, 4(4):406–425, 1987.

31 Svetoslav Savchev and Titu Andreescu. *Mathematical Miniatures*, volume 43 of *Anneli Lax New Mathematical Library*. The Mathematical Association of America, 2003.

32 Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980. doi:10.1145/322217.322225.

33 J. L. Selfridge and E. G. Straus. On the determination of numbers by their sums of a fixed order. *Pacific J. Math.*, 8(4):847–856, 1958.

34 Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. of Symbolic and Algebraic Computation (EUROSAM'79)*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer, 1979.

A 7/2-Approximation Algorithm for the Maximum Duo-Preservation String Mapping Problem

Nicolas Boria[1], Gianpiero Cabodi[2], Paolo Camurati[3], Marco Palena[4], Paolo Pasini[5], and Stefano Quer[6]

1 **Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy,** nicolas.boria@polito.it
2 **Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy,** gianpiero.cabodi@polito.it
3 **Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy,** paolo.camurati@polito.it
4 **Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy,** marco.palena@polito.it
5 **Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy,** paolo.pasini@polito.it
6 **Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy,** stefano.quer@polito.it

Abstract

This paper presents a simple 7/2-approximation algorithm for the MAX DUO-PRESERVATION STRING MAPPING (MPSM) problem. This problem is complementary to the classical and well studied MIN COMMON STRING PARTITION problem (MCSP), that computes the minimal edit distance between two strings when the only operation allowed is to shift blocks of characters. The algorithm improves on the previously best known 4-approximation algorithm by computing a simple local optimum.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Polynomial approximation, Max Duo-Preservation String Mapping Problem, Min Common String Partition Problem, Local Search

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.11

1 Introduction

Within the field of stringology, string comparison is one of the central problems, as its applications range from data compression to bioinformatics. There are various ways to measure the similarity of two strings, however the most common measure is the so called *edit distance* that counts the minimum number of edit operations that must be performed in order to transform one string into the other. In the specific field of biology, the edit-distance may provide some measure of the kinship between different species based on the similarities of their DNA, as each edit operation can be considered as a single mutation. In data compression, it may help to store efficiently a set of similar yet different data (e.g., different versions of the same object). Indeed, when a set of elements all have a short edit-distance towards a single "base element", an efficient way to compress the whole set of data might be to store only the "base" element of the set, and then record all the other elements as series of edit operations.

Obviously, the concept of edit distance changes definition based on the set of edit operations that are allowed. We tackle the classical case where the only edit operation that

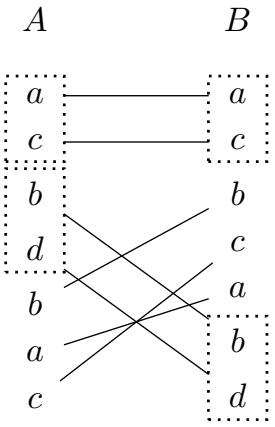


Figure 1 A mapping π that preserves 2 duos.

is allowed is to shift a block of characters, that is, to change the order of the characters in the string by modifying the position of some substring. In this case, the edit distance can be measured by solving the MIN COMMON STRING PARTITION (MCPS).

The MCSP is a fundamental and widely studied problem in the field of string comparison, which applications in the field of bioinformatics are described in [7, 13]. Given a string A let $\mathcal{P}_A$ denote a *partition* of A, that is, a set of substrings whose concatenation results in A. Consider two strings A and B, both with n characters, such that B is a permutation of A. The MCSP Problem introduced in [13] and [18] asks for two partitions $\mathcal{P}_A$ of A and $\mathcal{P}_B$ of B of minimum cardinalities such that $\mathcal{P}_A$ is a permutation of $\mathcal{P}_B$. The $k-$MCSP denotes a natural restriction of the problem where each character of the alphabet has at most k occurrences in each string. In [13], it is shown that this problem is NP-Hard and even APX-Hard. This holds also when the number of occurrences of each character is at most 2, and the result follows from a reduction to MAX INDEPENDENT SET (note that the problem is trivial when the maximal number of occurrences of each character is at most one). Since its introduction in [13], the problem has been intensively studied in various frameworks, such as polynomial approximation [7, 8, 9, 13, 15, 16] and parametric computation [3, 4, 10, 14]. Regarding polynomial approximation, the best results known so far are an $O(\log n \log^* n)$-approximation algorithm for the general version of the problem [9], and an $O(k)$-approximation for $k-$MCSP [16]. Regarding parametric computation, the problem was proved to be Fixed Parameter Tractable (FPT), first with respect to both k and the cardinality ϕ of an optimal partition [3, 10, 14], and more recently, with respect to ϕ only [4].

In [6], the symmetrical (maximization) version of the problem is introduced and denoted by MAX DUO-PRESERVATION STRING MAPPING (MPSM). A *duo* is defined as a couple of consecutive characters in a given string. It is clear that when a couple of partitions $(\mathcal{P}_A, \mathcal{P}_B)$ are a solution for a given instance of MIN COMMON STRING PARTITION that partition A and B into ϕ substrings, this solution is equivalent to a mapping π from characters of A to characters of B that preserves exactly $n - \phi$ duos. A duo is considered *preserved* when its two consecutive characters are mapped to two consecutive characters in the other string. Hence, given two strings A and B, the MPSM problem asks for a mapping π from A to B that preserves a maximum number of duos. An example of mapping that preserves 2 duos is provided in Figure 1.

Reminding that MCSP is NP-Hard [13], its maximization version MPSM is also NP-Hard. However, it is likely that these two problems have different behaviours in terms of ap-

proximation, inapproximability, and parameterized complexity. Among many others, MAX INDEPENDENT SET and MIN VERTEX COVER provide a perfect example of two symmetrical problems having different characteristics: on the one hand, MIN VERTEX COVER is easily 2-approximable in polynomial time by taking all endpoints of a maximal matching [12], and is FPT [5], while on the other hand MAX INDEPENDENT SET is inapproximable within ratio $n^{\varepsilon-1}$ for a given $\varepsilon \in (0,1)$ unless $\mathbf{P} = \mathbf{NP}$ [17], and is $W[1]$-Hard [11].

In [6], some approximation results are presented for MPSM with the following method. A graph problem called CONSTRAINED MAXIMUM INDUCED SUBGRAPH (CMIS) is defined and proved to be a generalization of MPSM. Using a solution to the linear relaxation of CMIS, it is then shown that a randomized rounding provides a k^2 expected approximation ratio for k-CMIS (and thus for k-MPSM), and a 2 expected approximation ratio for 2-CMIS (and thus for 2-MPSM). In [2], these results were improved by introducing and analysing two simple approximation algorithms: the first guarantees a 4-approximation ratio (regardless of the value of k), while the second ensures an approximation ratio $8/5$ when $k = 2$ and ratio 3 when $k = 3$. Moreover, the problem is shown to be APX-Hard. Very recently, the problem was shown to be FPT with respect to the number of duos preserved [1].

In what follows, we present further improvements on the latter results, namely a polynomial $7/2$-approximation algorithm based on a local search technique. In Section 2, we present briefly a graph generalization of MPSM called MAX CONSECUTIVE BIPARTITE MATCHING. Then, we describe our local search algorithm in Section 3 for which we provide complexity analysis (Section 4) and bound on the approximation ratio (Section 5). We finally provide some perspective for future works and possible further improvements on the approximation guarantee in Section 6.

2 Graph translation of the Problem

We are interested in improving on the best known approximation algorithm for MAX DUO-PRESERVATION STRING MAPPING problem, that has approximation ratio 4. In [2], the problem is shown to be a particular case of the following graph problem, which we denote as MAX CONSECUTIVE BIPARTITE MATCHING. Given a bipartite graph where vertices on both sides are ordered : $A = (a_1, ..., a_n)$, $B = (b_1, ..., b_n)$, the MAX CONSECUTIVE BIPARTITE MATCHING problem asks for the maximum matching M such that if $(a_i, b_j) \in M$, then a_{i+1} can only be matched to b_{j+1}, and b_{j+1} can only be matched to a_{i+1}. In other words, sets of matched consecutive vertices on one side must be matched to consecutive vertices on the other side.

Let us recall briefly why MAX DUO-PRESERVATION STRING MAPPING is a particular case of MAX CONSECUTIVE BIPARTITE MATCHING. Strings A and B of any instance of MAX DUO-PRESERVATION STRING MAPPING can be translated as ordered duo sets D^A and D^B (for example, if $A =$ "abc" and $B =$ "bac", then $D^A = ((ab), (bc))$, and $D^B = ((ba), (ac))$).

Consider the bipartite graph $G(I)$ built in the following way (an example is provided in Figure 2):

- each vertex on the left-hand side represents a duo of the set D^A, and each vertex on the right-hand side represents a duo of D^B.
- edges exist between two vertices if and only if they represent the same duo (same couple of characters in the same order)

It is shown in [2] that any feasible solution M for MAX CONSECUTIVE BIPARTITE MATCHING in the graph $G(I)$ yields a mapping π between strings A and B that preserves at least $|M|$ duos (and exactly $|M|$ duos if M is inclusion-wise maximal).

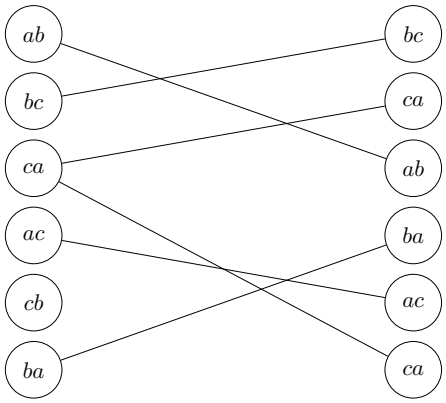


Figure 2 Graph $G(I)$ when I consists of $A =$ "$abcacba$" and $B =$ "$bcabaca$".

Indeed, such a matching can be seen as a partial mapping, and the number of edges in the matching is equal to the number of duos that the mapping preserves. The partial mapping can then be completed in an arbitrary way, since the set of non-mapped characters in A is a permutation of non-mapped vertices in B.

In the rest of the paper, we will refer only to the MAX CONSECUTIVE BIPARTITE MATCHING problem, bearing in mind that any approximation result that holds for MAX CONSECUTIVE BIPARTITE MATCHING also holds MAX DUO-PRESERVATION STRING MAPPING.

We call two edges *conflicting* if they cannot be both part of the same solution, either because they share a common endpoint or because their endpoints are consecutive on one side of the graph but not on the other.

In the following, we present an algorithm that produces such a partial mapping based on local search technique.

3 Local search algorithm

Local search algorithms produce solutions that are defined as local optima. A local optimum of an optimization problem is a solution that is optimal (either maximal or minimal) within a neighbouring set of candidate solutions. Starting from any feasible solution, the algorithm searches an improving solution in the neighbouring set, and repeatedly moves to an improving neighbouring solution as long as such a solution exists. When no improving neighbouring solution can be found, then the current solution is by definition a local optimum.

The quality of the local optimum obviously depends on the definition of the neighbouring set.

We devise a local search algorithm denoted `LOCAL`, which is based on a neighbourhood structure $\mathcal{N}$. Given a matching M that is a feasible solution for the problem, the neighbourhood of M, called $\mathcal{N}(M)$, contains all feasible solutions M' such that $|M \setminus M'| \leq 1$. In other words M' must contain all edges of M apart from possibly one.

While searching for an improving solution in the neighbouring set, the algorithm `LOCAL` will first try to improve the solution without removing any edge from the current solution M. On the one hand, if M is not inclusion-wise maximal, then there is an edge that can be added to the current solution M without having to remove any edge from it. If on the other hand M is inclusion-wise maximal, then the algorithm scans every matching $M \setminus \{v\}$ (for each $v \in M$) and checks if at least two edges can be added to one of these matchings. The pseudocode of algorithm `LOCAL` is provided in Section 4.

Algorithm 1 ALGORITHM LOCAL

Input: $G = (V, E)$
Output: M

```
1:  M = ∅, M' = ∅
2:  if ∃v ∈ E then
3:     M' ← {v}
4:  end if
5:  while M ≠ M' do
6:     M' ← M
7:     for each v ∈ E do
8:        if M ∪ {v} is feasible then
9:           M ← M ∪ {v}
10:          continue
11:       end if
12:    end for
13:    for each v ∈ M do
14:       for each (u, w) ∈ E × E do
15:          if M \ {v} ∪ {u, w} is feasible then
16:             M ← M \ {v} ∪ {u, w}
17:             break
18:          end if
19:       end for
20:    end for
21: end while
22: return M
```

4 Complexity analysis

We prove that the algorithm runs indeed in polynomial time. First of all, even starting from an empty solution, the algorithm will increment the value of its solution by at least one at each step, so that it will conclude after at most $|\text{SOL}| \leq n$ steps.

At each step, the algorithm first scans all edges that are not in SOL and checks if one of them does not conflict with any edge of SOL. This is done in $O(n^2)$ time. If such an edge is found, the current step is finished. Otherwise, for each edge u of SOL, the algorithm considers all sets of at most 6 non-solution edges conflicting with u, and checks if they can be added to the matching $\text{SOL} \setminus \{u\}$ without generating any conflict. This is done in $O(n^6)$ time for each edge u of the current solution: each edge of the solution conflicts with $O(n)$ non-solution edges, so that there are $O(n^6)$ candidate combinations of at most 6 non-solution edges to consider. Considering that, at each step, the current solution has $O(n)$ edges, the complexity of a single step is $O(n^7)$.

In all, the algorithm finishes after at most n steps, each step running in $O(n^7)$ time, so that the overall complexity is $O(n^8)$.

The complexity of LOCAL can actually been brought down to $O(n^4)$ thanks to the following observation. If an improvement incrementing the cardinality of the solution by at least one can be made at some step (by removing an edge u of SOL and adding a set X of at least two non conflicting edges to SOL), then an improvement incrementing this value by exactly one is also possible (by removing the same edge u of SOL and adding exactly any couple of edges of the set X). Thus, instead of scanning all sets of at most 6 non-solution edges conflicting

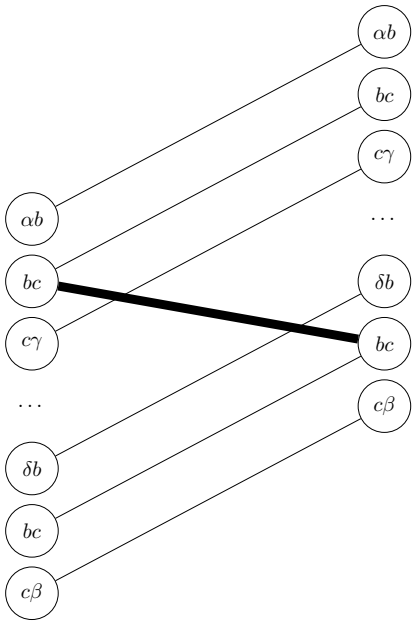


Figure 3 Conflicts among SOL and OPT edges.

with each edge u, it suffices that the algorithm scans only every couple of non-solution edges conflicting with u. If no improving couple can be found, then no improvement of any kind can be made, and the current solution is a local optimum.

5 Approximation analysis

We now prove that, indeed, `LOCAL` improves on the best known 4-approximate algorithm for MAX CONSECUTIVE BIPARTITE MATCHING:

▶ **Theorem 1.** *The algorithm `LOCAL` yields a 3.5 approximation ratio for* MAX CONSECUTIVE BIPARTITE MATCHING *problem.*

Consider that the algorithm `LOCAL` runs on an instance I of MAX CONSECUTIVE BIPARTITE MATCHING and outputs a solution SOL.

The proof is based on counting the conflicts between edges of SOL and edges of an unknown optimal matching OPT. We denote such number of conflicts by C.

On the one hand, a single edge of SOL cannot be conflicting with more than 6 edges of OPT (the worst case is shown in Figure 3). Indeed, on the one hand, any edge u can be in conflict only with edges that share an endpoint with u, or that have an endpoint that is consecutive to an endpoint of u (immediately after or immediately before), which results in no more than 6 possible endpoints for edges conflicting with u (the two endpoints of u, and the four consecutive vertices). On the other hand, any feasible solution including the optimal one can pick at most one edge per vertex of the graph. This gives us the following upper bound on the value of C:

$$C \leq 6|\text{SOL}|\,. \tag{1}$$

We recall that, by definition, there is no solution SOL$'$ in the neighbourhood $\mathcal{N}(\text{SOL})$ of SOL that has more edges than SOL. Hence, given any edge v of SOL the following fact holds:

▶ **Fact 2.** *Let v be an edge of solution* SOL *generated by* `LOCAL`*, and* OPT *be an optimal solution for the problem. There is at most one edge u of* OPT *that conflicts only with v in* SOL*.*

The fact is rather straightforward: suppose that there exist two edges u and t in a solution OPT that both conflict with a single edge v in SOL. The solution $\mathrm{SOL} \setminus \{v\} \cup \{u, t\}$ is an admissible matching in the neighbourhood of SOL and it contains more edges. Hence, `LOCAL` should have picked it instead of SOL.

Let us denote by k_1 the number of edges in OPT that conflict with one edge of SOL only. Fact 2 yields naturally the following bound:

$$k_1 \leq |\mathrm{SOL}|\,. \tag{2}$$

In OPT the remaining $|\mathrm{OPT}| - k_1$ edges conflict with at least 2 edges of SOL, which gives us the following lower bound on the number of conflicts C:

$$C \geq k_1 + 2(|\mathrm{OPT}| - k_1) \geq 2|\mathrm{OPT}| - k_1 \underset{(2)}{\geq} 2|\mathrm{OPT}| - |\mathrm{SOL}|\,. \tag{3}$$

Combining equations (1) and (3), we can easily get the following bound on the approximation ratio of `LOCAL`, which concludes the proof:

$$\frac{\mathrm{OPT}}{\mathrm{SOL}} \leq \frac{7}{2}\,.$$

6 Conclusion and perspectives

We showed that a simple local optimization technique provides a better approximation guarantee than the previously best known algorithm for MPSM. The analysis of more complex local optimums that rely on broader (yet polynomial) definitions of neighbourhood did not lead to immediate further improvements of the approximation guarantee. However, there are strong hints that, in such optimums, the number of edges that conflict with 6 edges of a global optimum is somehow linked to the number of edges of the global optimum conflicting with few edges of the local optimum. Namely, if many edges of the local optimum conflict with 6 edges of the global optimum, then few edges of the global optimum are expected to conflict few edges of the global optimum, resulting in a tighter version of equation 3, bounding for example the value $C(t)$ where t is the number of edges of the local optimum that conflict with 6 edges of the global optimum. Analysing such a bound might eventually lead to further improvements on the approximation ratio.

References

1 S. Beretta, M. Castelli, and R. Dondi. Parameterized Tractability of the Maximum-Duo Preservation String Mapping Problem. *ArXiv e-prints*, December 2015. `arXiv:1512.03220`.

2 N. Boria, A. Kurpisz, S. Leppänen, and M. Mastrolilli. Improved approximation for the maximum duo-preservation string mapping problem. In Dan Brown and Burkhard Morgenstern, editors, *Algorithms in Bioinformatics*, volume 8701 of *Lecture Notes in Computer Science*, pages 14–25. Springer Berlin Heidelberg, 2014. `doi:10.1007/978-3-662-44753-6_2`.

3 L. Bulteau, G. Fertin, C. Komusiewicz, and I. Rusu. A Fixed-Parameter Algorithm for Minimum Common String Partition with Few Duplications. In *WABI*, pages 244–258, 2013. `doi:10.1007/978-3-642-40453-5_19`.

4 L. Bulteau and C. Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In *SODA*, pages 102–121, 2014. doi:10.1137/1.9781611973402.8.

5 J. Chen, I.A. Kanj, and W. Jia. Vertex Cover: Further Observations and Further Improvements. In Peter Widmayer, Gabriele Neyer, and Stephan Eidenbenz, editors, *WG*, volume 1665 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 1999. doi:10.1007/3-540-46784-X_30.

6 W. Chen, Z. Chen, N. F. Samatova, L. Peng, J. Wang, and M. Tang. Solving the maximum duo-preservation string mapping problem with linear programming. *Theoretical Computer Science*, 530(0):1–11, 2014. doi:10.1016/j.tcs.2014.02.017.

7 X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, and T. Jiang. Assignment of Orthologous Genes via Genome Rearrangement. *Transactions on Computational Biology and Bioinformatics*, 2(4):302–315, 2005. doi:10.1145/1100863.1100950.

8 M. Chrobak, P. Kolman, and J. Sgall. The Greedy Algorithm for the Minimum Common String Partition Problem. In Klaus Jansen, Sanjeev Khanna, José D. P. Rolim, and Dana Ron, editors, *APPROX-RANDOM*, volume 3122 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2004. doi:10.1007/978-3-540-27821-4_8.

9 G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1), 2007. doi:10.1145/1219944.1219947.

10 P. Damaschke. Minimum Common String Partition Parameterized. In Keith A. Crandall and Jens Lagergren, editors, *WABI*, volume 5251 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 2008. doi:10.1007/978-3-540-87361-7_8.

11 R.G. Downey and M.R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999. 530 pp.

12 M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, CA, 1979.

13 A. Goldstein, P. Kolman, and J. Zheng. Minimum Common String Partition Problem: Hardness and Approximations. In Rudolf Fleischer and Gerhard Trippen, editors, *ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 484–495. Springer, 2004.

14 H. Jiang, B. Zhu, D. Zhu, and H. Zhu. Minimum common string partition revisited. *Journal of Combinatorial Optimization*, 23(4):519–527, 2012. doi:10.1007/s10878-010-9370-2.

15 P. Kolman and T. Walen. Approximating reversal distance for strings with bounded number of duplicates. *Discrete Applied Mathematics*, 155(3):327–336, 2007. doi:10.1016/j.dam.2006.05.011.

16 P. Kolman and T. Walen. Reversal Distance for Strings with Duplicates: Linear Time Approximation using Hitting Set. *Electronic Journal of Combinatorics*, 14(1), 2007. URL: http://www.combinatorics.org/Volume_14/Abstracts/v14i1r50.html.

17 C. Lund and M. Yannakakis. The Approximation of Maximum Subgraph Problems. In Andrzej Lingas, Rolf G. Karlsson, and Svante Carlsson, editors, *ICALP*, volume 700 of *Lecture Notes in Computer Science*, pages 40–51. Springer, 1993. doi:10.1007/3-540-56939-1_60.

18 K.M. Swenson, M. Marron, J.V. Earnest-DeYoung, and B.M.E. Moret. Approximating the true evolutionary distance between two genomes. *ACM Journal of Experimental Algorithmics*, 12, 2008. doi:10.1145/1227161.1402297.

Fast Compatibility Testing for Rooted Phylogenetic Trees*

Yun Deng[1] and David Fernández-Baca[2]

1 **Department of Computer Science, Iowa State University, Ames, IA 50011, USA**
yundeng@iastate.edu

2 **Department of Computer Science, Iowa State University, Ames, IA 50011, USA**
fernande@iastate.edu

Abstract

We consider the following basic problem in phylogenetic tree construction. Let $\mathcal{P} = \{T_1, \ldots, T_k\}$ be a collection of rooted phylogenetic trees over various subsets of a set of species. The tree compatibility problem asks whether there is a tree T with the following property: for each $i \in \{1, \ldots, k\}$, T_i can be obtained from the restriction of T to the species set of T_i by contracting zero or more edges. If such a tree T exists, we say that $\mathcal{P}$ is compatible.

We give a $\tilde{O}(M_{\mathcal{P}})$ algorithm for the tree compatibility problem, where $M_{\mathcal{P}}$ is the total number of nodes and edges in $\mathcal{P}$. Unlike previous algorithms for this problem, the running time of our method does not depend on the degrees of the nodes in the input trees. Thus, it is equally fast on highly resolved and highly unresolved trees

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory, J.3 Life and Medical Sciences

Keywords and phrases Algorithms, computational biology, phylogenetics

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.12

1 Introduction

Building a phylogenetic tree that encompasses all living species is one of the central challenges of computational biology. Two obstacles to achieving this goal are lack of data and conflict among the data that is available. The data shortage is tied to the vast disparity in the amount of information at our disposal for different families of species and the limited amount of comparable data across families [16]. One approach to overcoming this obstacle begins by identifying subsets of species for which enough data is available, and building phylogenies for each subset. The resulting trees are then synthesized into a single phylogeny – a supertree – for the combined set of species. This approach, proposed in the early 90s [2, 15], has been used successfully to build large-scale phylogenies (see, e.g., [3, 10]).

Any attempt at synthesizing phylogenetic information from multiple input trees must deal with the potential for conflict among these trees. Conflict may arise due to errors, or due to phenomena such as gene duplication and loss, and horizontal gene transfer. A fundamental question is whether conflict exists at all; that is, does there exist a supertree that exhibits the evolutionary relationships implicit in each input tree? We can formalize

* Supported in part by the National Science Foundation under grants CCF-1017189 and CCF-1422134.

this question as follows. Let $\mathcal{P} = \{T_1, \ldots, T_k\}$ be a collection of rooted phylogenetic trees, where, for each $i \in \{1, \ldots, k\}$, T_i is a phylogenetic tree for a set of species $L(T_i)$. The *tree compatibility problem* asks whether there exists a phylogenetic supertree T for the set of species $\bigcup_{i=1}^{k} L(T_i)$ such that, for each $i \in \{1, \ldots, k\}$, T_i can be obtained from $T|L(T_i)$ – the minimal subtree of T spanning $L(T_i)$ – by zero or more contractions of internal edges. If the answer is "yes", then $\mathcal{P}$ is said to be *compatible*; otherwise, $\mathcal{P}$ is *incompatible*.

Here we present an algorithm that solves the compatibility problem for rooted trees in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, where $M_{\mathcal{P}}$ is the total number of vertices and edges in the trees in $\mathcal{P}$. This running time is independent of the degrees of the internal nodes of the input trees.

1.1 Previous Work

Aho et al. [1] gave the first polynomial-time algorithm for the rooted tree compatibility problem. Their motivation was not phylogenetics, but relational databases. Steel [18] was perhaps the first to note the relevance of Aho et al.'s algorithm to supertree construction. His version of the Aho et al. algorithm, which he called the `Build` algorithm, has been a major influence in later work, including the present paper.

Henzinger et al. [9] showed that one can check the compatibility of a collection $\mathcal{R}$ of rooted triples – that is, phylogenetic trees on three species – in $O(|\mathcal{R}| \log^2 |\mathcal{R}|)$ time. (The time bound stated in [9] is higher, but can be improved using a faster dynamic graph connectivity data structure [11].) Any collection of trees $\mathcal{P}$ can be encoded by a collection of rooted triples $\mathcal{R}(\mathcal{P})$, obtained by enumerating the restriction of each input tree to every three-element subset of its species set (see Section 2). If n denotes the total number of distinct species in $\mathcal{P}$, then we get a trivial upper bound of $|\mathcal{R}(P)| = O(n^3 k)$. We can improve on this by finding a *minimal* set $\mathcal{R}^*$ of rooted triples that define the input trees. If the trees are binary – *fully resolved*, in the language of phylogenetics –, then $O(n)$ triples suffice for each tree, giving us $|\mathcal{R}^*| = O(nk)$. If input trees admit non-binary – that is, *unresolved* – nodes, however, the number of triples needed per input tree is roughly proportional to n^2 (the precise bound depends on the sum of the products of the degrees of internal nodes and the degrees of their children [8]), giving us $|\mathcal{R}^*| = O(n^2 k)$. Of course, the extra step of finding $\mathcal{R}^*$ adds to the complexity of the algorithm.

The tree compatibility problem is related to the *incomplete directed perfect phylogeny problem* (IDPP). Indeed, any collection of k phylogenetic trees on n distinct species can be encoded as a problem of testing the compatibility of a collection of $O(M_{\mathcal{P}})$ "directed partial characters" on n species[1]. Intuitively, each such character encodes the species in the subtree rooted at some node in an input tree. There is a $\tilde{O}(nm)$ algorithm to test the compatibility of m incomplete characters [14], which can be adapted to yield a $\tilde{O}(nM_{\mathcal{P}})$ algorithm for tree compatibility.

When the input trees are unrooted, the tree compatibility problem becomes NP-hard [18]. Nevertheless, the decision version is polynomial-time solvable if k is fixed [4]; that is, the problem is fixed-parameter tractable in k. The proof of fixed-parameter tractability in [4] relies on Courcelle's Theorem [6], and thus is an existence proof, rather than a practical algorithm.

Finally, we note that there are linear-time algorithms for testing the compatibility of a collection of trees that all have exactly the same leaf label set. One such algorithm can be obtained using recent results on computing "loose" and "strict" consensus trees [13]. Both

[1] For a precise definition of partial characters and IDPP, we refer the reader to Pe'er et al. [14].

types of consensus trees can be found in $O(nk)$ time, which is $O(M_{\mathcal{P}})$ when all leaf label sets are identical. (We thank J. Jansson for pointing this out.)

1.2 Our Contributions

At a high level, our algorithm resembles `Build` [18, 17]. There are, however, important differences. `Build` relies on the *triplet graph*, whose nodes are the species and where there is an edge between two species if they are involved in a triplet (see Section 2). Our algorithm relies instead on intersection graphs of sets of species associated with certain nodes of the input trees. Our graphs allow a more compact representation of the triplets induced by the trees in $\mathcal{P}$ (see Section 3). The key to the correctness of our approach is the close relationship between the triplet graph and our intersection graph (see Lemma 5 of Section 3). We remark that intersection graphs have a long history of use in testing compatibility, beginning with the work of Buneman [5].

We also take ideas from other sources. From Pe'er et al.'s IDPP algorithm [14], we adapt the idea of a *semi-universal node*. Although the graphs used to solve IDPP and rooted compatibility are different, semi-universal nodes play similar roles in each case: they capture the notion of sets of nodes in the input trees that map to the same node in a supertree, if a supertree exists. The relationship between our algorithm and Pe'er et al.'s goes deeper. Our approach can be viewed as an algorithm for IDPP that takes advantage of the fact that our particular set of incomplete characters arises from a collection of trees.

Intersection graphs are a convenient tool to prove the correctness for our algorithm. They are less convenient for an implementation, because they are hard to maintain dynamically, as our algorithm requires. The difficulty lies in recomputing set intersections whenever the graphs are updated. We avoid this by using *display graphs*, an idea that we borrow from the proof of the fixed-parameter tractability of unrooted compatibility [4]. The display graph of a collection $\mathcal{P}$ is obtained by identifying leaves in the input trees that have the same label. Display graphs provide all the connectivity information we need for our intersection graphs (see Lemma 8 of Section 4), but are easier to maintain.

Through our techniques, we achieve what, to our knowledge, is the first algorithm for rooted compatibility to achieve near-linear time under all input conditions, regardless of the degrees of the nodes in the input trees. This is an essential quality for dealing with large datasets.

1.3 Contents

Section 2 reviews basic concepts in phylogenetics, defines compatibility formally, and introduces triplets and the triplet graph. Section 3 presents our intersection graph approach to testing tree compatibility. Section 4 describes the implementation details needed to achieve the $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time bound. Section 5 contains some final remarks.

2 Preliminaries

For each positive integer r, $[r]$ denotes the set $\{1, \ldots, r\}$.

2.1 Phylogenetic Trees

Let T be a rooted tree. We use $V(T)$, $E(T)$, and $r(T)$ to denote the nodes, edges, and the root of T, respectively. For each $x \in V(T)$, we use $\mathrm{Ch}(x)$ and $T(x)$ to denote the set of

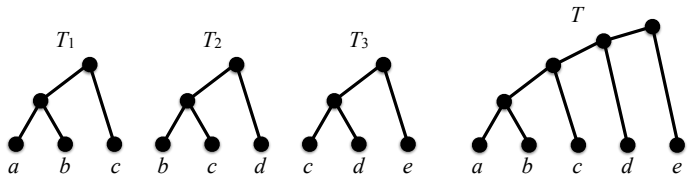


Figure 1 A profile $\mathcal{P} = \{T_1, T_2, T_3\}$ and a tree T that displays $\mathcal{P}$.

children of x and the subtree of T rooted at x, respectively. Suppose $u, v \in V(T)$. Then, u is a *descendant* of v if v lies on the path from u to $r(T)$ in T. Note that v is a descendant of itself. T is *binary*, or *fully resolved*, if each of its internal nodes has two children.

A (rooted) *phylogenetic tree* is a rooted tree T where every internal node has at least two children, along with a bijection λ that maps each leaf of T to an element of a set of *species*, denoted by $L(T)$. For each $x \in V(T)$, $L(x)$ denotes the set of species mapped to the leaves of $T(x)$; that is, $L(x) = \{\lambda(v) : v \text{ is a leaf in } T(x)\}$. $L(x)$ is called the *cluster* at x. Note that $L(r(T)) = L(T)$. The set of all clusters in T is $\mathrm{Cl}(T) = \{L(x) : x \in V(T)\}$.

The following lemma, adapted from [17, p. 52], is part of the folklore of phylogenetics.

▶ **Lemma 1.** *Let $\mathcal{H}$ be a collection of non-empty subsets of a set of species X that includes all singleton subsets of X as well as X itself. If there exists a phylogenetic tree T such that $\mathrm{Cl}(T) = \mathcal{H}$, then, up to isomorphism, T is unique.*

Let T be a phylogenetic tree and A be a set of species. The *restriction* of T to A, denoted $T|A$ is the phylogenetic tree with species set A where $\mathrm{Cl}(T|A) = \{C \cap A : C \in \mathrm{Cl}(T) \text{ and } C \cap A \neq \emptyset\}$. Let T' be a phylogenetic tree. T *displays* T' if $\mathrm{Cl}(T') \subseteq \mathrm{Cl}(T|L(T'))$.

A *rooted triple* is a binary phylogenetic tree on three leaves. A rooted triple with leaves a, b, and c is denoted $ab|c$ if the path from a to b does not intersect the path from c to the root. We treat $ab|c$ and $ba|c$ as equivalent.

When restricted to the three-element subsets of its species set, a phylogenetic tree T induces a set $\mathcal{R}(T)$ of rooted triples, defined as $\mathcal{R}(T) = \{T|X : X \subseteq L(T), |X| = 3 \text{ and } T|X \text{ is binary}\}$.

▶ **Lemma 2** ([17, p. 119])**.** *Let T and T' be two phylogenetic trees. Then T displays T' if and only if $\mathcal{R}(T') \subseteq \mathcal{R}(T)$.*

2.2 Profiles and Compatibility

Throughout the rest of this paper $\mathcal{P} = \{T_1, \ldots, T_k\}$ denotes a set where, for each $i \in [k]$, T_i is a phylogenetic tree. We refer to $\mathcal{P}$ as a *profile*, and write $L(\mathcal{P})$ to denote $\bigcup_{i\in[k]} L(T_i)$, the *species set* of $\mathcal{P}$. We write $V(\mathcal{P})$ for $\bigcup_{i\in[k]} V(T_i)$, $E(\mathcal{P})$ for $\bigcup_{i\in[k]} E(T_i)$, and $\mathcal{R}(\mathcal{P})$ for $\bigcup_{i\in[k]} \mathcal{R}(T_i)$. Given a subset A of $L(P)$, $\mathcal{P}|A$ denotes the profile $\{T_1|A, \ldots, T_k|A\}$. The *size* of $\mathcal{P}$ is $M_{\mathcal{P}} = |V(\mathcal{P})| + |E(\mathcal{P})|$. Note that $M_{\mathcal{P}} = O(nk)$.

Profile $\mathcal{P}$ is *compatible* if there exists a phylogenetic tree T such that, for each $i \in [k]$, T displays T_i. If such a tree T exists, we say that T *displays* $\mathcal{P}$. See Figure 1.

2.3 The Triplet Graph

The *triplet graph* of a profile $\mathcal{P}$, denoted $\Gamma(\mathcal{P})$, is the graph whose vertex set is $L(P)$ and where there is an edge between species a and b if and only if there exists a $c \in L(P)$ such that $ab|c \in \mathcal{R}(\mathcal{P})$. The following observation concerning singleton profiles will be useful.

▶ **Observation 1.** Let T be a phylogenetic tree with $|L(T)| > 2$. Let $u_1, \ldots, u_p$ be the children of $r(T)$. Then, the connected components of $\Gamma(\{T\})$ are $L(u_1), \ldots, L(u_p)$, where $p \geq 2$.

3 Testing Compatibility

Here we describe our compatibility algorithm and prove its correctness. We begin with some definitions.

Let U be a subset of $V(\mathcal{P})$ and let $L(U)$ denote $\bigcup_{u \in U} L(u)$. Then, $G_{\mathcal{P}}(U)$ denotes the graph with vertex set U and where $u, v \in U$ are joined by an edge if and only if $L(u) \cap L(v) \neq \emptyset$. That is, $G_{\mathcal{P}}(U)$ is the intersection graph of the clusters associated with the nodes in U. For each $i \in [k]$, let $U(i) = U \cap V(T_i)$. We say that U is *valid* if, for each $i \in [k]$,

V1 if $|U(i)| \geq 2$, then there exists a node $v \in V(T_i)$ such that $U(i) \subseteq \mathrm{Ch}(v)$ and

V2 $L(U(i)) = L(T_i) \cap L(U)$.

Observe that the set U_{init} defined as follows is valid.

$$U_{\text{init}} = \{r(T_i) : i \in [k]\} \tag{1}$$

Note that $L(U_{\text{init}}) = L(\mathcal{P})$. From this point forward, we assume that $G_{\mathcal{P}}(U_{\text{init}})$ is connected. No generality is lost by doing so. To see why, observe that if $G_{\mathcal{P}}(U_{\text{init}})$ is not connected, then $\mathcal{P}$ can be partitioned into a collection of species-disjoint profiles $\mathcal{P}_1, \ldots, \mathcal{P}_r$ such that $\mathcal{P}$ is compatible if and only if $\mathcal{P}_j$ is compatible for all $j \in [r]$.

The next observation follows from the definition of a valid set.

▶ **Observation 2.** If U is a valid subset of $V(\mathcal{P})$, then, for each $i \in [k]$, $\mathrm{Cl}(T_i|L(U)) = \{L(U(i))\} \cup \{L(v) : v \text{ is a descendant of a node in } U(i)\}$.

Together with Lemma 1, Observation 2 shows that $T_i|L(U)$ is completely determined by the descendants of $U(i)$.

A valid subset U of $V(\mathcal{P})$ is *compatible* if there exists a phylogenetic tree T with $L(T) = L(U)$ that displays $T_i|L(U)$ for every $i \in [k]$. If such a tree T exists, we say that T *displays* U.

▶ **Lemma 3.** *Profile $\mathcal{P}$ is compatible if and only if every valid subset of $V(\mathcal{P})$ is compatible.*

Proof.

(⇐) If every valid subset of $V(\mathcal{P})$ is compatible, then, in particular, so is the set U_{init} of Equation (1). Let T be a tree that displays U_{init}. Then, $L(T) = L(U_{\text{init}}) = L(\mathcal{P})$. Thus, for every $i \in [k]$, $T_i|L(T) = T_i$, and thus T displays T_i. Hence, $\mathcal{P}$ is compatible.

(⇒) Suppose $\mathcal{P}$ is compatible, but there is a valid subset U of $V(\mathcal{P})$ that is not compatible. Let T be a tree that displays $\mathcal{P}$. But then $T|U$ displays U, a contradiction. ◀

`BuildST` (Algorithm 1), which is closely related to Semple and Steel's `Build` algorithm [17], determines whether a valid set $U \subseteq V(\mathcal{P})$ is compatible. The key difference between `BuildST` and `Build` is that the latter uses the triplet graph $\Gamma(\mathcal{P})$, while `BuildST` uses the graph $G_{\mathcal{P}}(U)$, for different subsets U of $V(\mathcal{P})$. As we show in Lemma 5, the two graphs are closely related. Nevertheless, $G_{\mathcal{P}}(U)$ offers some computational advantages over the triplet graph. Intuitively, this is because $G_{\mathcal{P}}(U)$ is a more compact representation of the triplets in $\mathcal{R}(\mathcal{P})$.

`BuildST`(U) attempts to build a tree T_U for U. Step 1 initializes the root of T_U. If $L(U)$ consists of one or two species, then U is trivially compatible; Steps 2–5 handle these cases.

Algorithm 1: `BuildST`(U)

Input: A valid set $U \subseteq V(\mathcal{P})$ such that $G_{\mathcal{P}}(U)$ is connected.
Output: A tree T_U that displays U, if U is compatible; `incompatible` otherwise.

1 Create a node r_U
2 **if** $|L(U)| = 1$ **then**
3 | **return** the tree consisting of node r_U, labeled by the single species in $L(U)$
4 **if** $|L(U)| = 2$ **then**
5 | **return** the tree consisting of node r_U and two children, each labeled by a different species in $L(U)$
6 **foreach** $i \in [k]$ *such that* $|U(i)| = 1$ **do**
7 | Let v be the single element in $U(i)$
8 | $U = (U \setminus \{v\}) \cup \mathrm{Ch}(v)$
9 Let $W_1, W_2, \ldots, W_p$ be the connected components of $G_{\mathcal{P}}(U)$
10 **if** $p = 1$ **then**
11 | **return** `incompatible`
12 **foreach** $j \in [p]$ **do**
13 | Let $t_j =$ `BuildST`(W_j)
14 | **if** t_j *is a tree* **then**
15 | | Add t_j to the set of subtrees of r_U
16 | **else**
17 | | **return** `incompatible`
18 **return** the tree with root r_U

The loop in lines 6–8 identifies the indices $i \in [k]$ such that $U(i)$ is a singleton. For each such i, it removes the single element v in $U(i)$ and replaces v by its children in T_i. Note that if v is a leaf in T_i, then $U(i) = \emptyset$ after this step. As we argue in the proof of Theorem 7, when $\mathcal{P}$ is compatible, all such nodes v – provided they are not leaves – map to the same node w in the tree T that displays $\mathcal{P}$, in the sense that $L(w)$ is the smallest cluster in T such that $L(v) \subseteq L(w)$[2]. In Theorem 7, we also show that, if $G_{\mathcal{P}}(U)$ remains connected after steps 6–8, then U is incompatible. This case is handled in Line 11. Otherwise, Lines 12–17 recursively process each connected component of $G_{\mathcal{P}}(U)$. If the recursive calls succeed in finding trees for all components, these trees are assembled into a phylogeny for U by joining them to the root created in Step 1. If any recursive call determines that a component is incompatible, then U is declared to be incompatible.

The correctness of `BuildST` relies on two lemmas, the first of which can be proved using induction.

▶ **Lemma 4.** *If, given a valid set $U \subseteq V(\mathcal{P})$,* `BuildST`$(U)$ *returns a tree T_U, then T_U is a phylogenetic tree such that $L(T_U) = L(U)$.*

The next lemma is central to the correctness proof of `BuildST`.

▶ **Lemma 5.** *Let $W_1, \ldots, W_p$ be the connected components of $G_{\mathcal{P}}(U)$ at step 9 of* `BuildST`(U)*, for some valid set $U \subseteq V(\mathcal{P})$. Then,*
(i) *for each $j \in [p]$, W_j is a valid set, and*
(ii) *the connected components of $\Gamma(\mathcal{P}|L(U))$ are precisely $L(W_1), \ldots, L(W_p)$.*

[2] Thus, v plays the role of a *semi-universal node*, in the sense of Pe'er et al. [14].

Proof.

(i) Let U_{bef} and U_{aft} denote the values of U before and after executing steps 6–8. Each element of U_{aft} is either an element of U_{bef} or a child of some $v \in U_{\text{bef}}$. In the latter case, every child of v is in U_{aft}. By assumption, U_{bef} is valid, and for every non-leaf node v, $L(v) = \bigcup_{w \in \text{Ch}(v)} L(w)$; therefore, U_{aft} must also be valid. Part (i) follows.

(ii) We can show that the following holds after steps 6–8.

▶ **Claim 6.** *Let a and b be any two species in $L(U)$. Then, (a, b) is an edge in $\Gamma(\mathcal{P}|L(U))$ if and only if there exists a node $v \in U$ such that $a, b \in L(v)$.*

Observe that both $\Pi_1 = \{A : A \text{ is a connected component of } \Gamma(\mathcal{P}|L(U))\}$ and $\Pi_2 = \{L(W) : W \text{ is a connected component of } G_{\mathcal{P}}(U)\}$ are partitions of $L(U)$. We prove that $\Pi_1 = \Pi_2$ by showing that (a) for each connected component A of $\Gamma(\mathcal{P}|L(U))$ there exists a connected component W of $G_{\mathcal{P}}(U)$ such that $A \subseteq L(W)$, and (b) for each connected component W of $G_{\mathcal{P}}(U)$ there exists a connected component A of $\Gamma(\mathcal{P}|L(U))$ such that $L(W) \subseteq A$.

(a) Let A be any connected component of $\Gamma(\mathcal{P}|L(U))$. We argue that any two species a, b in A must be in the same connected component of $G_{\mathcal{P}}(U)$. Let $U_a = \{v \in U : a \in L(v)\}$ and $U_b = \{v \in U : b \in L(v)\}$. Then, each of U_a and U_b is a clique in $G_{\mathcal{P}}(U)$. It thus suffices to show that there is a path between some node in U_a and some node in U_b.

By the definition of A, there exists a path between a and b in $\Gamma(\mathcal{P}|L(U))$. Suppose this path is $\rho = \langle a_1, \ldots, a_m \rangle$, where $a_1 = a$ and $a_m = b$. By Claim 6, for each $l \in [m-1]$, there exists a node $w_l \in U$ such that $\{a_l, a_{l+1}\} \subseteq L(w_l)$. For each $l \in [m-2]$, $L(w_l) \cap L(w_{l+1}) \neq \emptyset$, so, either $w_l = w_{l+1}$ or there is a edge between w_l and w_{l+1} in $G_{\mathcal{P}}(U)$. Let $\pi = \langle w_1, \ldots, w_{m-1} \rangle$. Then, we can extract from π a subsequence that is a path from w_1 to w_{m-1} in $G_{\mathcal{P}}(U)$. By the definition of ρ, $a \in L(w_1)$ and $b \in L(w_{m-1})$, so $w_1 \in U_a$ and $w_l \in U_b$. This completes the proof of part (a).

(b) Let W be any connected component of $G_{\mathcal{P}}(U)$. If $|L(W)| = 1$, the statement holds trivially, so assume that $|L(W)| > 1$. We argue that any two species a, b in $L(W)$ are in the same connected component of $\Gamma(\mathcal{P}|L(U))$. Let v_a and v_b be nodes in W such that $a \in L(v_a)$ and $b \in L(v_b)$. If $v_a = v_b$, then, by Claim 6, (a, b) is an edge of $\Gamma(\mathcal{P}|L(U))$, and we are done. So, suppose instead that $v_a \neq v_b$.

Let us call a path π from v_a to v_b *good* if $|L(w)| > 1$ for every node w in π. We claim that there exists a good path from v_a to v_b. To prove this claim, we first argue that we can choose v_a and v_b such that $|L(v_a)|, |L(v_b)| > 1$. Indeed, consider the case of species a (the case for b is analogous). If $|L(v)| = 1$ for every node $v \in W$ such that $a \in L(v)$, then we would have $|L(W)| = 1$, contradicting our assumption that $|L(W)| > 1$. Now, suppose the path π from v_a to v_b has a node $w \notin \{v_a, v_b\}$ such that $|L(w)| = 1$. Let w' and w'' be the predecessor and successor of w in π. Then, $L(w') \cap L(w'') = L(w) \neq \emptyset$, so there is an edge between w' and w''. Thus, we can delete w from π and the resulting sequence remains a path between v_a and v_b.

Let $\pi = \langle w_1, \ldots, w_l \rangle$, where $w_1 = v_a$ and $w_l = v_b$, be a good path from v_a to v_b in $G_{\mathcal{P}}(U)$. Choose a sequence of species $\rho = \langle c_1, \ldots, c_{l+1} \rangle$, where $c_1 = a$, $c_{l+1} = b$ and, for each $j \in [l]$, $c_j, c_{j+1} \in L(w_j)$ and $c_j \neq c_{j+1}$. Note that such a choice is always possible. Then, by Claim 6, (c_j, c_{j+1}) is an edge of $\Gamma(\mathcal{P}|L(U))$. Hence, ρ is a path from a to b in $\Gamma(\mathcal{P}|L(U))$. ◀

We are now ready to prove the correctness of `BuildST`.

▶ **Theorem 7.** *Let U_{init} be the set defined in Equation* (1)*. Then,* `BuildST`(U_{init}) *either (i) returns a tree T that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible, or (ii) returns* `incompatible` *otherwise.*

Proof. We first argue that if `BuildST`(U_{init}) outputs `incompatible`, $\mathcal{P}$ is indeed incompatible. Assume, on the contrary, that $\mathcal{P}$ is compatible. Then, there must be a call `BuildST`(U) for some valid subset U such that $|L(U)| > 2$, in which the graph $G(U)$ of step 9 has a single connected component, $W_1 = U$. By Lemma 3, U must be compatible, so there exists a phylogeny T_U that displays U. By Observation 1, $\Gamma(\{T_U\})$ has at least two connected components A and B. By Lemma 5(ii), however, $\Gamma(\mathcal{P}|L(U))$ is connected, so there exist species $a \in A$ and $b \in B$ such that $ab|c \in \mathcal{R}(\mathcal{P}|U)$. But $ab|c \notin \mathcal{R}(T)$, and, by Lemma 2, T does not display some tree in $\mathcal{P}|L(U)$, a contradiction. Thus, $G(U)$ has at least two components.

Now, suppose that `BuildST`(U_{init}) returns a tree T. We prove that T displays $\mathcal{P}$ by arguing that for each $i \in [k]$ there is an injective mapping $\phi_i : V(T_i) \to V(T)$ that maps every node $v \in V(T_i)$ to a distinct node $\phi_i(v) \in V(T)$ such that $L(v) \subseteq L(\phi_i(v))$.

By Lemma 4, each recursive call `BuildST`(U) returns a phylogenetic tree T_U for $L(U)$. Let r_U denote the root of T_U. We have two cases.

Case (i): $|L(U)| \leq 2$**.** For each $i \in [k]$, we must have $|U(i)| \in \{0, 1, 2\}$; we only need to consider $|U(i)| \in \{1, 2\}$. Suppose first that $|U(i)| = 1$, and let v be the single node in $U(i)$. Note that $L(v) \subseteq L(r_U)$. Thus, we make $\phi_i(v) = r_U$. If $|L(U(i))| = 1$, we are done. Otherwise, $|L(U(i))| = 2$. Then, v has two children, v_1 and v_2, both leaves, labeled with, say, species s_1 and s_2, respectively. Node r_U also has two children, r_1 and r_2. Assume, without loss of generality, that these children are labeled with species s_1 and s_2, respectively. Then, $L(v_j) = L(r_j)$ for $j \in \{1, 2\}$. Therefore, we make $\phi_i(v_j) = r_j$ for each $j \in \{1, 2\}$. Now, suppose that $|U(i)| = 2$. Then, $|L(U(i))| = 2$, and each node in $U(i)$ is a leaf in T_i. As in the previous case, we map each node of $U(i)$ to the corresponding child of r_U.

Case (ii): $|L(U)| > 2$**.** Let U_{bef} be the value of U before entering the loop of lines 6–8, and let U_{aft} be the value of U at line 9, after the loop of lines 6–8 terminates. Let $U_{\text{rem}} = \{v \in U_{\text{bef}} : v \in U_{\text{bef}}(i)$ for some $i \in [k]$ such that $|U_{\text{bef}}(i)| = 1\}$. Then $U_{\text{aft}} = (U_{\text{bef}} \setminus U_{\text{rem}}) \cup \{u \in \text{Ch}(v) : v \in U_{\text{rem}}\}$. Assume inductively that every descendant of a node in U_{aft} is mapped to an appropriate node in T_U. It therefore suffices to establish mappings for the nodes in U_{rem}. Now, for every $v \in U_{\text{rem}}$, $L(v) \subseteq L(r_U)$. Thus, we make $\phi(v) = r_U$ for every $v \in U_{\text{rem}}$. ◀

4 Implementation

We now explain how to implement `BuildST` in order to solve the tree compatibility problem in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time. Consider a call to `BuildST`(U). Recall that we can assume that $G_{\mathcal{P}}(U)$ is connected. `BuildST`(U) requires the following three pieces of information.

(G1) *The value of* $|L(U)|$. This number is needed in Lines 2 and 4 of `BuildST`.

(G2) *The set* $J(U)$ *of all* $i \in [k]$ *such that* $|U(i)| = 1$. Set $J(U)$ contains the indices i considered in Lines 6–8 of `BuildST`.

(G3) *The set* $U(i) = U \cap V(T_i)$ *for each* $i \in [k]$. For each $i \in J(U)$, $U(i)$ contains precisely the element v used in Lines 7 and 8 of `BuildST`.

It is straightforward to obtain (G1), (G2), and (G3) for the valid set U_{init} of Equation (1): $|L(U_{\text{init}})| = n$, $J(U_{\text{init}}) = [k]$, and, for every $i \in [k]$, $U_{\text{init}}(i) = \{r(T_i)\}$. Now assume that we have (G1), (G2), and (G3) at the beginning of some call to `BuildST`(U). Steps 6–8 modify U and, therefore, $G_{\mathcal{P}}(U)$. Suppose that, at Line 9, $G_{\mathcal{P}}(U)$ has more than one connected component. We need to compute (G1), (G2), and (G3) for each connected component, in order to pass this information to the recursive calls in Line 13. That is, if $p > 1$, for each $j \in [p]$, we need to compute $|L(W_j)|$, $J(W_j)$, and $W_j(i) = W_j \cap V(T_i)$, for each $i \in [k]$.

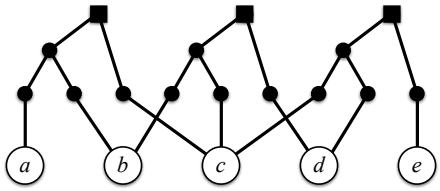


Figure 2 The graph $H_{\mathcal{P}}(U_{\text{init}})$ for the profile $\mathcal{P}$ of Figure 1. Nodes of U_{init} are drawn as squares. Nodes in the set $\{x_s : s \in L(\mathcal{P})\}$ are labeled with the corresponding species. Species labeling the leaves of trees in $\mathcal{P}$ are omitted.

We use the dynamic graph connectivity data structure by Holm et al. [11]. We refer to this data structure as *HDT*. HDT allows us to maintain the list of nodes in each component, as well as the number of these nodes so that, if we start with no edges in a graph with N vertices, the amortized cost of each update is $O(\log^2 N)$. For efficiency, however, we do not use HDT directly on $G_{\mathcal{P}}(U)$. The reason is that the edges of $G_{\mathcal{P}}(U)$ are defined via intersections of sets of species, which could make it costly to determine the new nodes and edges created as a result of Step 8. To avoid this problem, we proceed indirectly, through an auxiliary graph $H_{\mathcal{P}}(U)$, defined below. As we shall see, $H_{\mathcal{P}}(U)$ offers another advantage over $G_{\mathcal{P}}(U)$: maintaining $H_{\mathcal{P}}(U)$ only requires handling deletions, but maintaining $G_{\mathcal{P}}(U)$ additionally requires handling insertions.

We define $H_{\mathcal{P}}(U)$ as a subgraph of the graph $H_{\mathcal{P}}$ constructed as follows. For each species $s \in L(\mathcal{P})$, create a new node $x_s \notin V(\mathcal{P})$, and let $X_{\mathcal{P}} = \{x_s : s \in L(\mathcal{P})\}$. Then, $H_{\mathcal{P}}$ is the graph whose vertex set is $V(\mathcal{P}) \cup X_{\mathcal{P}}$ and whose edge set is $E(\mathcal{P}) \cup \{(u, x_s) : u$ is a leaf in T_i, for some $i \in [k]$, such that $\lambda(u) = s\}$. Note that $H_{\mathcal{P}}$ has $O(M_{\mathcal{P}})$ nodes and edges, and can be constructed from $\mathcal{P}$ in $O(M_{\mathcal{P}})$ time. $H_{\mathcal{P}}$ is essentially the *display graph* for $\mathcal{P}$ [4]. The display graph is the result of glueing together leaves in $\mathcal{P}$ labeled by the same species. Contrast this with $H_{\mathcal{P}}$, which connects leaves with a common label through nodes in $X_{\mathcal{P}}$. This minor difference with respect to the display graph serves to simplify our presentation.

Given a valid subset U of $V(\mathcal{P})$, we define $H_{\mathcal{P}}(U)$ as the subgraph of $H_{\mathcal{P}}$ induced by $\{v : v$ is a descendant of some node $u \in U\} \cup \{x_s \in X_{\mathcal{P}} : s \in L(U)\}$. Note that $H_{\mathcal{P}}(U_{\text{init}}) = H_{\mathcal{P}}$. See Figure 2.

The next result states the basic properties of $H_{\mathcal{P}}(U)$. Due to space limitations, we omit its proof.

▶ **Lemma 8.** *The following statements hold for any valid subset U of $V(\mathcal{P})$.*

(i) *Let v be a node in U. If $U' = (U \setminus \{v\}) \cup \text{Ch}(v)$, then $H_{\mathcal{P}}(U')$ is obtained from $H_{\mathcal{P}}(U)$ by deleting v and every edge (v, u) such that $u \in \text{Ch}(v)$.*

(ii) *Any two nodes in U are in the same connected component in $G_{\mathcal{P}}(U)$ if and only if they are in the same connected component of $H_{\mathcal{P}}(U)$.*

By Lemma 8(ii), the connected components $W_1, \ldots, W_p$ of $G_{\mathcal{P}}(U)$ can be put into a one-to-one correspondence with the connected components $Y_1, \ldots, Y_p$ of $H_{\mathcal{P}}(U)$ so that $W_j = Y_j \cap U$ for each $j \in [p]$.

We represent $H_{\mathcal{P}}(U)$ using the aforementioned HDT data structure. For each connected component Y of $H_{\mathcal{P}}(U)$, we maintain three fields:

(H1) Y.`count`, the cardinality of $Y \cap X_{\mathcal{P}}$,

(H2) Y.`singleton`, a doubly-linked list that contains all indices $i \in [k]$ such that $|U(i)| = 1$, and

(H3) Y.`List`, an array where, for each $i \in [k]$, Y.`List`$[i]$ is a doubly-linked list consisting of the elements of $Y \cap U(i)$.

Recall that we assume that $G_{\mathcal{P}}(U)$ is connected at the beginning of a call to `BuildST`(U). Thus, by Lemma 8, $H_{\mathcal{P}}(U)$ has a single connected component, Y. Then, $|L(U)| = Y.\texttt{count}$, $J(U) = Y.\texttt{singleton}$, and $Y.\texttt{List}[i]$ contains the elements of $U(i)$, for each $i \in [k]$. Thus, the three fields of Y provide `BuildST`(U) with the information that it needs – that is, (G1), (G2), and (G3). In particular, they allow us to easily find each node v considered in Line 7 of `BuildST`(U). Line 8 is then performed as a series of edge deletions, one for each edge (v, u) such that $u \in \mathrm{Ch}(v)$, followed by the deletion of v (we provide further details below). By Lemma 8(i), this correctly updates $H_{\mathcal{P}}(U)$. The deletions break up $H_{\mathcal{P}}(U)$ into a collection of connected components $Y_1, \ldots, Y_p$. For each $j \in [p]$, Y_j corresponds to a connected component W_j of $G_{\mathcal{P}}(U)$ that (if $p > 1$) is processed in a recursive call in Line 13. We need to compute $Y_j.\texttt{count}$, $Y_j.\texttt{singleton}$, $Y_j.\texttt{List}$ for each $j \in [p]$, in order to provide this information to the recursive calls.

The total number of edge and node deletions executed by `BuildST`(U_{init}) – including all deletions conducted by the recursive calls – cannot exceed the total number of edges and nodes in $H_{\mathcal{P}}$, which is $O(M_{\mathcal{P}})$. The HDT data structure allows us to maintain connectivity information throughout the entire algorithm in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$. In the remainder of this section, we show that we can maintain the `count`, `singleton`, and `List` fields throughout the entire algorithm in total time $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$. We also argue that all the required information for $H_{\mathcal{P}}(U_{\text{init}})$ can be initialized in $O(M_{\mathcal{P}})$ time.

Let $Y_{\text{init}} = V(\mathcal{P}) \cup X_{\mathcal{P}}$ be the vertex set of $H_{\mathcal{P}}(U_{\text{init}})$. Then, Y_{init} is the single connected component of $H_{\mathcal{P}}(U_{\text{init}})$. We initialize the data fields of Y_{init} as follows: (1) $Y_{\text{init}}.\texttt{count} = |L(\mathcal{P})|$, (2) $Y_{\text{init}}.\texttt{singleton}$ is the set $[k]$, and (3) for each $i \in [k]$, $Y_{\text{init}}.\texttt{List}[i]$ consists of $r(T_i)$. Thus, we can initialize all data fields in $O(M_{\mathcal{P}})$ time.

We assume that every node v in $H_{\mathcal{P}}(U)$ is either *marked*, if $v \in U$, or *unmarked*, if $v \notin U$. Initially, each node $v \in U_{\text{init}}$ is marked, and every node $v \in Y_{\text{init}} \setminus U_{\text{init}}$ is unmarked. We also assume that for each node v in $H_{\mathcal{P}}(U)$, we maintain sufficient information to determine in $O(1)$ time whether $v \in X_{\mathcal{P}}$ or $v \in V(\mathcal{P})$, and that, in the latter case, we have $O(1)$-time access to the index $i \in [k]$ such that $v \in V(T_i)$. For each i such that $Y.\texttt{List}[i]$ contains exactly one element, we maintain a pointer from $Y.\texttt{List}[i]$ to the entry for i in $Y.\texttt{singleton}$. This allows us to update $Y.\texttt{singleton}$ in $O(1)$ time when $U(i)$ is no longer a singleton. For each marked node $v \in Y$ (so $v \in U$), we maintain a pointer from v to the element in $Y.\texttt{List}[i]$ that contains v. This allows us to update $Y.\texttt{List}[i]$ in $O(1)$ time when v becomes unmarked.

Consider a call to `BuildST`(U) for some valid set U. Step 1 takes $O(1)$ time. Since $H_{\mathcal{P}}(U)$ initially consists of a single connected component, say Y, and we have $Y.\texttt{count}$, Steps 2–5 also take $O(1)$ time. Let $H = H_{\mathcal{P}}(U)$. We implement the loop in lines 6–8 as follows. First, we enumerate the indices in $J = J(U)$ in $O(|J|)$ time by listing the elements of $Y.\texttt{singleton}$. For each $i \in J$, we retrieve and remove the single element v_i of $U(i)$ from $Y.\texttt{List}[i]$, and then delete i from $Y.\texttt{singleton}$. This takes $O(1)$ time. We unmark v_i, and for every node $u \in \mathrm{Ch}(v_i)$ we mark u and add it to $Y.\texttt{List}[i]$. This takes $O(1)$ time per edge. We then successively delete each edge (v_i, u) such that $u \in \mathrm{Ch}(v_i)$, updating (H1)–(H3) for each newly-created component along the way. Once these edges are deleted, we delete v_i itself. By Lemma 8(i), the result is the graph $H_{\mathcal{P}}(U)$ for the new set U. Let us focus on how to handle the deletion of a single edge $e = (v_i, u)$.

Let Y' be the connected component of H that currently contains v_i. We query the HDT data structure to determine, in $O(\log^2 M_{\mathcal{P}})$ amortized time, whether deleting (v_i, u) splits Y' into two components. If Y' remains connected, no updates are needed. Otherwise, Y' is split into two parts Y_1 and Y_2. To fill in the `count`, `singleton`, and `List` fields of Y_1 and Y_2, we use the well-known technique of scanning the smaller component [7]. We query the HDT

data structure to determine, in $O(1)$ time, which of Y_1 and Y_2 has fewer nodes. Suppose without loss of generality that $|Y_1| \leq |Y_2|$. We initialize Y_2.`count` and Y_2.`List` to Y'.`count` and Y'.`List`, respectively. We initialize Y_1.`count` to 0 and Y_1.`List`$[i]$ to null for each $i \in [k]$. We then scan each node v in Y_1, and do the following. If $v \in X_{\mathcal{P}}$, we decrement Y_2.`count` and increment Y_1.`count`. Otherwise $v \in V(\mathcal{P})$; assume that $v \in V(T_i)$. If v is marked, we remove v from Y_2.`List`$[i]$ and add v to Y_1.`List`$[i]$; each such move takes $O(1)$ time. This operation requires at most one update in each of Y_1.`singleton` and Y_2.`singleton`; each update takes $O(1)$ time.

We claim that any node v is scanned $O(\log M_{\mathcal{P}})$ times over the entire execution of `BuildST`(U_{init}). To verify this, let $N(v)$ be the number of nodes in the connected component containing v. Suppose that, initially, $N(v) = N$. Then, the rth time we scan v, $N(v) \leq N/2^r$. Thus, v is scanned $O(\log N)$ times. The claim follows, since $N = O(M_{\mathcal{P}})$. Therefore, the total number of updates over all nodes is $O(M_{\mathcal{P}} \log M_{\mathcal{P}})$, and the work per update is $O(1)$.

To summarize, the work done by `BuildST` consists of three parts: (i) initialization, (ii) maintaining connected components, and (iii) maintaining the `count`, `singleton`, and `List` fields for each connected component. Part (i) takes $O(M_{\mathcal{P}})$ time. Part (ii) involves $O(M_{\mathcal{P}})$ edge and node deletions on the HDT data structure, at an amortized cost of $O(\log^2 M_{\mathcal{P}})$ per deletion. Part (iii) involves scanning the nodes of our graph every time a deletion creates a new component, for a total of $O(M_{\mathcal{P}} \log M_{\mathcal{P}})$ scans, at $O(1)$ cost per scan, over the entire execution of `BuildST`. This yields our main result.

▶ **Theorem 9.** *Let U_{init} be the set defined in Equation (1). Then, there exists and implementation of* `BuildST` *such that* `BuildST`(U_{init}) *runs in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time.*

5 Discussion

A trivial lower bound for the tree compatibility problem is $\Omega(M_{\mathcal{P}})$, the time to read the input. Thus, our result leaves us a polylogarithmic factor away from an optimal algorithm for compatibility. Is it possible to reduce or even eliminate this gap? The bottleneck is the time to maintain the information associated with the various components of $H_{\mathcal{P}}(U)$. It is conceivable that the special structure of this graph and the way the deletions are performed could be used to our advantage. A second question is how well our algorithm performs in practice. To investigate this, it should be possible to leverage existing knowledge on the empirical behavior of dynamic connectivity data structures [12].

References

1. Alfred V. Aho, Yehoshua Sagiv, Thomas G. Szymanski, and Jeffrey D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Comput.*, 10(3):405–421, 1981.
2. Bernard R. Baum. Combining trees as a way of combining data sets for phylogenetic inference, and the desirability of combining gene trees. *Taxon*, 41:3–10, 1992.
3. Olaf R. P. Bininda-Emonds, Marcel Cardillo, Kate E. Jones, Ross D. E. MacPhee, Robin M. D. Beck, Richard Grenyer, Samantha A. Price, Rutger A. Vos, John L. Gittleman, and Andy Purvis. The delayed rise of present-day mammals. *Nature*, 446:507–512, 2007.
4. David Bryant and Jens Lagergren. Compatibility of unrooted phylogenetic trees is FPT. *Theoretical Computer Science*, 351:296–302, 2006.
5. Peter Buneman. A characterisation of rigid circuit graphs. *Discrete Math.*, 9:205–212, 1974.
6. Bruno Courcelle. The monadic second-order logic of graphs I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

7 Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, January 1981. URL: http://doi.acm.org/10.1145/322234.322235, doi:10.1145/322234.322235.

8 Stefan Grünewald, Mike Steel, and M. Shel Swenson. Closure operations in phylogenetics. *Mathematical Biosciences*, 208:521–537, 2007.

9 Monika Rauch Henzinger, Valerie King, and Tandy Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24:1–13, 1999.

10 Cody E. Hinchliff, Stephen A. Smith, James F. Allman, J. Gordon Burleigh, Ruchi Chaudhary, Lyndon M. Coghill, Keith A. Crandall, Jiabin Deng, Bryan T. Drew, Romina Gazis, Karl Gude, David S. Hibbett, Laura A. Katz, H. Dail Laughinghouse IV, Emily Jane McTavish, Peter E. Midford, Christopher L. Owen, Richard H. Reed, Jonathan A. Reesk, Douglas E. Soltis, Tiffani Williams, and Karen A. Cranston. Synthesis of phylogeny and taxonomy into a comprehensive tree of life. *Proceedings of the National Academy of Sciences*, 2015. In press. doi:10.1073/pnas.1423041112.

11 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001. URL: http://doi.acm.org/10.1145/502090.502095, doi:10.1145/502090.502095.

12 Raj Iyer, David Karger, Hariharan Rahul, and Mikkel Thorup. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *J. Exp. Algorithmics*, 6, December 2001. URL: http://doi.acm.org/10.1145/945394.945398, doi:10.1145/945394.945398.

13 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Improved algorithms for constructing consensus trees. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1800–1813, 2013. URL: http://dx.doi.org/10.1137/1.9781611973105.129, doi:10.1137/1.9781611973105.129.

14 Itsik Pe'er, Tal Pupko, Ron Shamir, and Roded Sharan. Incomplete directed perfect phylogeny. *SIAM J. Comput.*, 33(3):590–607, 2004. URL: http://dx.doi.org/10.1137/S0097539702406510, doi:10.1137/S0097539702406510.

15 Mark A. Ragan. Phylogenetic inference based on matrix representation of trees. *Molecular Phylogenetics and Evolution*, 1:53–58, 1992.

16 Michael J. Sanderson. Phylogenetic signal in the eukaryotic tree of life. *Science*, 321(5885):121–123, 2008.

17 Charles Semple and Mike Steel. *Phylogenetics*. Oxford Lecture Series in Mathematics. Oxford University Press, Oxford, 2003.

18 Mike A. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *J. Classification*, 9:91–116, 1992.

Hardness of RNA Folding Problem With Four Symbols*

Yi-Jun Chang

Department of EECS, University of Michigan, Ann Arbor, MI, USA
cyijun@umich.edu

Abstract

An RNA sequence is a string composed of four types of nucleotides, A, C, G, and U. Given an RNA sequence, the goal of the RNA folding problem is to find a maximum cardinality set of crossing-free pairs of the form $\{A, U\}$ or $\{C, G\}$. The problem is central in bioinformatics and has received much attention over the years. Whether the RNA folding problem can be solved in $\mathcal{O}(n^{3-\epsilon})$ time remains an open problem. Recently, Abboud, Backurs, and Williams (FOCS'15) made the first progress by showing a conditional lower bound for a generalized version of the RNA folding problem based on a conjectured hardness of the k-clique problem. However, their proof requires alphabet size ≥ 36 to work, making the result biologically irrelevant. In this paper, by constructing the gadgets using a lemma of Bringmann and Künnemann (FOCS'15) and surrounding them with some carefully designed sequences, we improve upon the framework of Abboud et al. to handle the case of alphabet size 4, yielding a conditional lower bound for the RNA folding problem. We also investigate the Dyck edit distance problem. We demonstrate a reduction from RNA folding problem to Dyck edit distance problem of alphabet size 10, establishing a connection between the two fundamental string problems. This leads to a much simpler proof of the conditional lower bound for Dyck edit distance problem given by Abboud et al. and lowers the required alphabet size for the lower bound to work.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases RNA folding, Dyck edit distance, longest common subsequence, conditional lower bound, clique

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.13

1 Introduction

An *RNA sequence* is a string composed of four types of nucleotides, A, C, G, and U. Given an RNA sequence, the goal of the *RNA folding* problem is to find a maximum cardinality set of crossing-free pairs of nucleotides, where all the pairs are either $\{A, U\}$ or $\{C, G\}$. The problem is central in bioinformatics and has found applications in many areas of molecular biology. For a comprehensive exposition of the topic, the reader is referred to e.g. [18].

It is well-known that the problem can be solved in cubic time by a simple dynamic programming method [9]. Due to the importance of RNA folding in practice, there has been a long line of research on improving the time complexity (See e.g. [3, 11, 12, 13, 18, 21]). Currently the best upper bound is $\mathcal{O}\left(\frac{n^3}{\log^2(n)}\right)$ [13, 18], which can be obtained by four-Russian method or fast min-plus multiplication (based on ideas from Valiant's CFG parser [19]).

* A more detailed version of the paper: http://arxiv.org/abs/1511.04731. Supported by NSF grants CCF-1217338, CNS-1318294, and CCF-1514383.

Whether the RNA folding problem can be solved in $\mathcal{O}(n^{3-\epsilon})$ time for some $\epsilon > 0$ is still a major open problem. Other than attempting to improve the upper bound, we should also approach the problem in the opposite direction, i.e. arguing why the problem is hard.

Conditional lower bounds

A popular way to show hardness of a problem is to demonstrate a lower bound conditioned on some widely accepted hypothesis.

▶ **Conjecture 1** (Strongly Exponential Time Hypothesis (SETH)). *There exists no $\epsilon, k_0 > 0$ such that k-SAT with n variables can be solved in time $\mathcal{O}(2^{(1-\epsilon)n})$ for all $k > k_0$.*

▶ **Conjecture 2.** *There exists no $\epsilon, k_0 > 0$ such that k-clique on graphs with n nodes can be solved in time $\tilde{\mathcal{O}}\left(n^{(\omega-\epsilon)k/3}\right)$ for all $k > k_0$, where $\omega < 2.373$ is the matrix multiplication exponent.*

For instance, assuming that SETH (Conjecture 1) holds, the following bounds are unattainable for any $\epsilon > 0$:

- an $\mathcal{O}(n^{k-\epsilon})$ algorithm for k-dominating set problem [14],
- an $\mathcal{O}(n^{2-\epsilon})$ algorithm for dynamic time warping, longest common subsequence, and edit distance [2, 6, 7],
- an $\mathcal{O}(m^{2-\epsilon})$ algorithm for $(3/2-\epsilon)$-approximating the diameter of a graph with m edges [15].

We note that such negative results allow us to have a better picture of the structure of polynomial time complexity, and identify the main obstacles to obtaining faster algorithms for various fundamental problems.

As remarked in [1], it is easy to reduce the longest common subsequence problem on binary strings to the RNA folding problem as following: Given two binary strings X, Y, we let $\hat{X} \in \{A, C\}^{|X|}$ be the string such that $\hat{X}[i] = A$ if $X[i] = 0$, $\hat{X}[i] = C$ if $X[i] = 1$, and we let $\hat{Y} \in \{G, U\}^{|Y|}$ be the string such that $\hat{Y}[i] = U$ if $Y[i] = 0$, $\hat{Y}[i] = G$ if $Y[i] = 1$. Then we have a 1-1 correspondence between RNA foldings of $\hat{X} \circ \hat{Y}^R$ (i.e. concatenation of $\hat{X}$ and the reversal of $\hat{Y}$) and common subsequences of X and Y. It has been shown in [7] that there is no $\mathcal{O}(n^{2-\epsilon})$ algorithm for longest common subsequence problem on binary strings conditioned on SETH, and we immediately get the same conditional lower bound for RNA folding from the simple reduction!

Very recently, based on a conjectured hardness of k-clique problem (Conjecture 2), a higher conditional lower bound was proved for a generalized version of the RNA folding problem (which coincides with the RNA folding problem when the alphabet size is 4) [1]:

▶ **Theorem 1** ([1]). *If the generalized RNA folding problem on sequences of length n with alphabet size 36 can be solved in $T(n)$ time, then $3k$-clique on graphs with $|V| = n$ can be solved in $\mathcal{O}\left(T\left(n^{k+2}\log(n)\right)\right)$ time.*

Therefore, a $\mathcal{O}(n^{\omega-\epsilon})$ time algorithm for the generalized RNA folding with alphabet size at least 36 will disprove Conjecture 2, yielding a breakthrough to the parameterized complexity of clique problem.

However, the above theorem is irrelevant to the RNA folding problem in real life (which has alphabet size 4). It is unknown whether the generalized RNA folding for alphabet size 4 admits a faster algorithm than the case for alphabet size > 4. In fact, there are examples of string algorithms whose running time scales with alphabet size (e.g. string matching with

mismatched [5] and jumbled indexing [4, 8]). We also note that when the alphabet size is 2, the generalized RNA folding can be trivially solved in linear time.

In this paper, we improve upon Theorem 1 by showing the same conditional lower bound for the RNA folding problem:

▶ **Theorem 2.** *If the RNA folding problem on sequences in* $\{A, C, G, U\}^n$ *can be solved in* $T(n)$ *time, then* $3k$*-clique on graphs with* $|V| = n$ *can be solved in* $\mathcal{O}\left(T\left(n^{k+1}\log(n)\right)\right)$ *time.*

Note that we also get an $\mathcal{O}(n)$ factor improvement inside $T(\cdot)$, though it does not affect the conditional lower bound.

In the proof of Theorem 1 in [1], given a graph $G = (V, E)$, a sequence of length $\mathcal{O}(n^{k+2}\log(n))$ is constructed in such a way that we can decide whether G has a $3k$-clique according to the number of pairs in an optimal generalized RNA folding of S. The construction requires a large alphabet size to build various "walls" which prevent undesired pairings between different parts of the sequence. Extending their approach to handle the case with alphabet size 4 may not be easy without aid from other techniques and ideas.

Overview of our approach

At a high level, our reduction (from $3k$-clique problem to RNA folding problem) follows the approach in [1]: We enumerate all k-cliques, and each of them is encoded as some gadgets. All the gadgets are then put together to form an RNA sequence. The goal is to ensure that an optimal RNA folding corresponds to choosing three k-cliques that form a $3k$-clique, given that the underlying graph admits a $3k$-clique.

To achieve this result using 4 symbols, we implement the above construction using more efficient gadgets based on a key lemma in [7], whose original purpose is to prove that longest common subsequence and other edit distance problems are SETH-hard even on binary strings. We will treat it as a black box and apply it multiple times.

In the final RNA sequence, all clique gadgets are well-separated by some carefully designed sequences whose purpose is to "trap" all the clique gadgets except three of them. We will see that only these three clique gadgets can influence the number of matched pairs in an optimal RNA folding, and the number of matched pairs is maximized when these three clique gadgets correspond to a $3k$-clique. Therefore, we can infer whether the graph has a $3k$-clique from the optimal RNA folding of the RNA sequence.

Dyck Edit Distance

One other way to formulate the RNA folding problem is as follows: deleting the minimum number of letters in a given string to transform the string into a string in the language defined by the grammar $\mathbf{S} \to \mathbf{SS}, A\mathbf{S}U, U\mathbf{S}A, C\mathbf{S}G, G\mathbf{S}C, \epsilon$ (empty string). The *Dyck edit distance problem* [16, 17], which asks for the minimum number of edits to transform a given string to a well-balanced parentheses of s different types, has a similar formulation. Due to the similarity, the same conditional lower bound as Theorem 1 was also shown for the Dyck edit distance problem (with alphabet size ≥ 48) in [1]. In this paper, we improve and simplify their result by demonstrating a simple reduction from RNA folding to Dyck edit distance problem:

▶ **Theorem 3.** *If Dyck edit distance problem on sequences of length* n *with alphabet size 10 can be solved in* $T(n)$ *time, then the RNA folding problem on sequences in* $\{A, C, G, U\}^n$ *can be solved in* $\mathcal{O}(T(n))$ *time.*

▶ **Corollary 4.** *If the Dyck edit distance problem on sequences of length n with alphabet size 10 can be solved in $T(n)$ time, then $3k$-clique on graphs with $|V| = n$ can be solved in $\mathcal{O}\left(T\left(n^{k+1}\log(n)\right)\right)$ time.*

Interpretations of our results

The current state-of-art algorithm for $k-$clique, which takes $\mathcal{O}\left(n^{\omega k/3}\right)$ time, requires the use of fast matrix multiplication [10] which does not perform very efficiently in practice. For combinatorial, non-algebraic algorithm for $k-$clique, the current state-of-art has time complexity $\mathcal{O}\left(\frac{n^k}{\log^k(n)}\right)$ [20], which is only slightly better than the trivial approach.

Therefore, despite the current gap between the n^3 upper bound and the n^ω lower bound (neglecting polylog factors) for RNA folding and Dyck edit distance, it is unlikely to have an $n^{3-\epsilon}$ time "efficient" algorithm for these problems, unless there is a breakthrough in combinatorial algorithms for k-clique. As a result, our reductions (and the ones in [1]) imply that very likely the use of approximation or heuristic is necessary if one needs a faster algorithm.

2 Preliminaries

Given a set of letters Σ, the set Σ' is defined as $\{x'|x \in \Sigma\}$. We require that $\Sigma \cap \Sigma' = \emptyset$, and $\forall x, y \in \Sigma, (x \neq y) \rightarrow (x' \neq y')$. Therefore, we have $|\Sigma'| = |\Sigma|$ and $|\Sigma \cup \Sigma'| = 2|\Sigma|$.

For any $X = (x_1, \ldots, x_k) \in \Sigma^k$, we write $p(X)$ to denote $(x'_1, \ldots, x'_k)$ (the letter p stands for the prime symbol). We denote the reversal of the sequence X as X^R. The concatenation of two sequences X, Y is denoted as $X \circ Y$ (or simply XY). We write *substring* to denote a contiguous subsequence. Two pairs of indices (i_1, j_1), (i_2, j_2), with $i_1 < j_1$ and $i_2 < j_2$, form a *crossing pair* iff $(\{i_1, j_1\} \cap \{i_2, j_2\} \neq \emptyset) \vee (i_1 < i_2 < j_1 < j_2) \vee (i_2 < i_1 < j_2 < j_1)$.

Generalized RNA Folding

Given $S \in (\Sigma \cup \Sigma')^n$, the goal of the generalized RNA folding problem is to find a maximum cardinality set $A \subseteq \{(i, j)|1 \leq i < j \leq n\}$ among all sets meeting the following conditions:

- A does not contain any crossing pair.
- For any $(i, j) \in A$, either (i) $S[i] \in \Sigma$ and $S[j] = S[i]'$ or (ii) $S[j] \in \Sigma$ and $S[i] = S[j]'$ is true.

We write $\mathrm{RNA}(S) = |A|$.

Any set meeting the above conditions is called an *RNA folding* of S. If its cardinality equals $\mathrm{RNA}(S)$, then it is said to be *optimal*.

In the paper we will only focus on the generalized RNA folding problem with four types of letters, i.e. $\Sigma = \{0, 1\}, \Sigma' = \{0', 1'\}$, which coincides with the RNA folding problem for alphabet $\{A, C, G, U\}$.

With a slight abuse of notation, sometimes we will write $(S[i], S[j])$ to denote a pair $(i, j) \in A$. The notation $\{\cdot, \cdot\}$ is used to indicate an unordered pair.

Longest Common Subsequence (LCS)

Given $X \in \Sigma^n$ and $Y \in \Sigma^m$, we define $\delta_{\mathrm{LCS}}(X, Y) = n + m - 2k$, where $k =$ the length of the longest common subsequence of X and Y. It is easy to observe that $\mathrm{RNA}(X \circ p(Y^R))$ equals the length of $\mathrm{LCS} = (n + m - \delta_{\mathrm{LCS}}(X, Y))/2$. In this sense, we can conceive of an LCS problem as an RNA folding problem with some structural constraint on the sequence.

In [7], a conditional lower bound for the LCS problem with $|\Sigma| = 2$ based on SETH was presented. A key technique in their approach is a function that transforms an instance of an alignment problem between two sets of sequences to an instance of the LCS problem, which is described below.

Alignments of two sets of sequences

Let $\mathbf{X} = (X_1, \ldots, X_n)$ and $\mathbf{Y} = (Y_1, \ldots, Y_m)$ be two linearly ordered sets of sequences of alphabet Σ. We assume that $n \geq m$. An *alignment* is a set $A = \{(i_1, j_1), (i_2, j_2), \ldots, (i_{|A|}, j_{|A|})\}$ with $1 \leq i_1 < i_2 < \ldots < i_{|A|} \leq n$ and $1 \leq j_1 < j_2 < \ldots < j_{|A|} \leq m$. An alignment A is called *structural* iff $|A| = m$ and $i_m = i_1 + m - 1$. That is, all sequences in $\mathbf{Y}$ are matched, and the matched positions in $\mathbf{X}$ are contiguous. The set of all alignments is denoted as $\mathcal{A}_{n,m}$, and the set of all structural alignments is denoted as $\mathcal{S}_{n,m}$.

The *cost* of an alignment A (with respect to $\mathbf{X}$ and $\mathbf{Y}$) is defined as:

$$\delta(A) = \sum_{(i,j)\in A} \delta_{\mathrm{LCS}}(X_i, Y_j) + (m - |A|) \max_{i,j} \delta_{\mathrm{LCS}}(X_i, Y_j).$$

That is, unaligned parts of $\mathbf{Y}$ are penalized by $\max_{i,j} \delta_{\mathrm{LCS}}(X_i, Y_j)$.

Given a sequence X, the *type* of X is defined as $(|X|, \sum_i X[i])$, where each letter is assumed to be a number. Note that when the alphabet is simply $\{0, 1\}$, $\sum_i X[i]$ is simply the number of occurrences of 1 in X.

The following key lemma was proved in [7] (Lemma 4.3 of [7]):

▶ **Lemma 5** ([7]). *Let $\mathbf{X} = (X_1, \ldots, X_n)$ and $\mathbf{Y} = (Y_1, \ldots, Y_m)$ be two linearly ordered sets of binary strings such that $n \geq m$, all X_i are of type $\mathcal{T}_X = (\ell_X, s_X)$, and all Y_i are of type $\mathcal{T}_Y = (\ell_Y, s_Y)$. There are two binary strings $S_X = \mathrm{GA}_X^{m,\mathcal{T}_Y}(X_1, \ldots, X_n)$, $S_Y = \mathrm{GA}_Y^{n,\mathcal{T}_X}(Y_1, \ldots, Y_m)$ and an integer C meeting the following requirements:*

- $\min_{A\in\mathcal{A}_{n,m}} \delta(A) \leq \delta_{LCS}(S_X, S_Y) - C \leq \min_{A\in\mathcal{S}_{n,m}} \delta(A)$.
- *The types of S_X, S_Y and the integer C only depend on $n, m, \mathcal{T}_X, \mathcal{T}_Y$.*
- *S_X, S_Y, and C can be calculated in time $\mathcal{O}((n+m)(\ell_X + \ell_Y))$. Hence $|S_X|$ and $|S_Y|$ are both $\mathcal{O}((n+m)(\ell_X + \ell_Y))$.*

Note that the term GA comes from the word gadget.

Intuitively, computing an optimal alignment (or an optimal structural alignment) of two sets of sequences is at least as hard as computing a longest common subsequence. The above lemma gives a reduction from the computation of a number s with $\min_{A\in\mathcal{A}_{n,m}} \delta(A) \leq s \leq \min_{A\in\mathcal{S}_{n,m}} \delta(A)$ (which can be regarded as an approximation of optimal alignments) to a single LCS instance.

In the next section, we will use the above lemma as a black box to devise two encodings, the clique node gadget $\mathrm{CNG}(t)$ and the clique list gadget $\mathrm{CLG}(t)$, for a k-clique t in a graph in such a way that we can decide whether two k-cliques t_1, t_2 form a $2k$-clique according the value of $\delta_{\mathrm{LCS}}(\mathrm{CNG}(t_1), \mathrm{CLG}(t_2))$.

When invoking the lemma, $\mathbf{X}$, $\mathbf{Y}$ are designed in such a way that we can test whether a condition is met (e.g. whether two given k-cliques form a $2k$-clique) by the value of $\min_{A\in\mathcal{A}_{n,m}} \delta(A)$. We will show that $\min_{A\in\mathcal{A}_{n,m}} \delta(A) = \min_{A\in\mathcal{S}_{n,m}} \delta(A)$ for the case we are interested in. Therefore, we can infer whether the condition we are interested in is met from the value of $\delta_{\mathrm{LCS}}(S_X, S_Y)$.

3 From Cliques to RNA Folding

The goal of this section is to prove Theorem 2.

Let $G = (V, E)$ be a graph, and let $n = |V|$. We write $\mathcal{C}_k$ to denote the set of k-cliques in G. We fix $\Sigma = \{0, 1\}$. As in [1], we will construct a sequence $S_G \in (\Sigma \cup \Sigma')^*$ such that we can decide whether G has a $3k$-clique according to the value of $\mathrm{RNA}(S_G)$.

As our framework of the construction of S_G is similar to the one in [1], we will give the building blocks for constructing S_G the same names as their analogues in [1], despite that they have different lower-level implementations.

3.1 Testing $2k$-cliques via LCS

We associate each vertex $v \in V$ a distinct integer in $\{0, 1, \dots\ n-1\}$. Let s_v be the binary encoding of such integer with $|s_v| = \lceil \log(n) \rceil$. We define $\bar{v}$ to be the binary string resulted by replacing each 0 in s_v with 01 and replacing each 1 in s_v with 10. It is clear that for each $v \in V$, $\bar{v}$ is of type $\mathcal{T}_0 = (2\lceil \log(n) \rceil, \lceil \log(n) \rceil)$, and $\delta_{\mathrm{LCS}}(\bar{u}, \bar{v}) = 0$ iff $u = v$.

Our goal is to devise two encodings $\mathrm{CNG}(t), \mathrm{CLG}(t)$ for a k-clique t such that we can infer whether two k-cliques t_1, t_2 form a $2k$-clique from the value of $\delta_{\mathrm{LCS}}(\mathrm{CNG}(t_1), \mathrm{CLG}(t_2))$.

For each $v \in V$, the *list gadget* $\mathrm{LG}(v)$ and the *node gadget* $\mathrm{NG}(v)$ are defined as following:

- $\mathrm{LG}(v) = \mathrm{GA}_X^{1,\mathcal{T}_0}(\bar{u}_1, \bar{u}_2, \dots, \bar{u}_{|N(v)|}, 1^{\lceil \log(n) \rceil}0^{\lceil \log(n) \rceil}, \dots, 1^{\lceil \log(n) \rceil}0^{\lceil \log(n) \rceil})$, where $N(v) = \{u_1, u_2, \dots, u_{|N(v)|}\}$, and the number of occurrences of $1^{\lceil \log(n) \rceil}0^{\lceil \log(n) \rceil}$ is $n - |N(v)|$.
- $\mathrm{NG}(v) = \mathrm{GA}_Y^{n,\mathcal{T}_0}(\bar{v})$.

▶ **Lemma 6.** *There is a constant c_0, depending only on n, such that for any $v_1, v_2 \in V$, we have $\{v_1, v_2\} \in E$ iff $\delta_{LCS}(LG(v_1), NG(v_2)) = c_0 = \min_{v_1', v_2' \in V} \delta_{LCS}(LG(v_1'), NG(v_2'))$.*

Proof. We let $N(v_1) = \{u_1, u_2, \dots, u_{|N(v_1)|}\}$.

Let $\mathbf{X} = (\bar{u}_1, \bar{u}_2, \dots, \bar{u}_{|N(v_1)|}, 1^{\lceil \log(n) \rceil}0^{\lceil \log(n) \rceil}, \dots, 1^{\lceil \log(n) \rceil}0^{\lceil \log(n) \rceil})$, where the number of occurrences of $1^{\lceil \log(n) \rceil}0^{\lceil \log(n) \rceil}$ is $n - |N(v_1)|$, and let $\mathbf{Y} = (\bar{v}_2)$.

In view of Lemma 5, $\min_{A \in \mathcal{A}_{n,1}} \delta(A) \le \delta_{\mathrm{LCS}}(\mathrm{LG}(v_1), \mathrm{NG}(v_2)) - C \le \min_{A \in \mathcal{S}_{n,1}} \delta(A)$, for some C whose value depends on $|\mathbf{X}|, |\mathbf{Y}|$, and $\mathcal{T}_0$. As these parameters depend solely on n, the number C only depends on n.

Since $|\mathbf{Y}| = 1$, any non-empty alignment between $\mathbf{X}$ and $\mathbf{Y}$ is structural. This implies that $\delta_{\mathrm{LCS}}(\mathrm{LG}(v_1), \mathrm{NG}(v_2)) - C = \min_{A \in \mathcal{A}_{n,1}} \delta(A) = \min_{A \in \mathcal{S}_{n,1}} \delta(A)$.

When $\{v_1, v_2\} \in E$, since $\bar{v}_2$ is contained in $\mathbf{X}$, clearly $\min_{A \in \mathcal{S}_{n,m}} \delta(A) = 0$. When $\{v_1, v_2\} \notin E$, $\bar{v}_2$ does not appear in $\mathbf{X}$, so $\min_{A \in \mathcal{S}_{n,m}} \delta(A) > 0$. Note that $1^{\lceil \log(n) \rceil}0^{\lceil \log(n) \rceil} \neq \bar{v}$, for any $v \in V$.

Hence $\{v_1, v_2\} \in E$ iff $\delta_{\mathrm{LCS}}(\mathrm{LG}(v_1), \mathrm{NG}(v_2)) = C = \min_{v_1', v_2' \in V} \delta_{\mathrm{LCS}}(\mathrm{LG}(v_1'), \mathrm{NG}(v_2'))$. Therefore, it suffices to set $c_0 = C$. ◀

We let $\mathcal{T}_X$ be the type of the list gadgets, and we let $\mathcal{T}_Y$ be the type of the node gadgets. For each k-clique $t = \{u_1, u_2, \dots, u_k\}$, we define the *clique node gadget* $\mathrm{CNG}(t)$ and the *clique list gadget* $\mathrm{CLG}(t)$ as following:

- $\mathrm{CLG}(t) = \mathrm{GA}_X^{k^2,\mathcal{T}_Y}(\mathrm{LG}(u_1), \dots, \mathrm{LG}(u_1), \mathrm{LG}(u_2), \dots, \mathrm{LG}(u_2), \dots, \mathrm{LG}(u_k), \dots, \mathrm{LG}(u_k))$, where the number of occurrences of each $\mathrm{LG}(u_i)$ is k.
- $\mathrm{CNG}(t) = \mathrm{GA}_Y^{k^2,\mathcal{T}_X}(\mathrm{NG}(u_1), \mathrm{NG}(u_2), \dots, \mathrm{NG}(u_k), \mathrm{NG}(u_1), \mathrm{NG}(u_2), \dots, \mathrm{NG}(u_k), \dots,$ $\mathrm{NG}(u_1), \mathrm{NG}(u_2), \dots, \mathrm{NG}(u_k))$, where the number of occurrences of each $\mathrm{NG}(u_i)$ is k.

We are ready to prove the main lemma in the subsection:

▶ **Lemma 7.** *There is a constant c_1, depending only on n, k, such that for any $t_1, t_2 \in \mathcal{C}_k$, $t_1 \cup t_2$ is a $2k$-clique iff $\delta_{LCS}(CLG(t_1), CNG(t_2)) = c_1 = \min_{t_1', t_2' \in \mathcal{C}_k} \delta_{LCS}(CLG(t_1'), CNG(t_2'))$.*

Proof. Let $t_1 = \{u_1, u_2, \ldots, u_k\}$, and let $t_2 = \{v_1, v_2, \ldots, v_k\}$.

Let $\mathbf{X} = (\mathrm{LG}(u_1), \ldots, \mathrm{LG}(u_1), \mathrm{LG}(u_2), \ldots, \mathrm{LG}(u_2), \ldots, \mathrm{LG}(u_k), \ldots, \mathrm{LG}(u_k))$, where each $\mathrm{LG}(u_i)$ appears k times, and let $\mathbf{Y} = (\mathrm{NG}(v_1), \mathrm{NG}(v_2), \ldots, \mathrm{NG}(v_k), \mathrm{NG}(v_1), \mathrm{NG}(v_2), \ldots, \mathrm{NG}(v_k), \ldots, \mathrm{NG}(v_1), \mathrm{NG}(v_2), \ldots, \mathrm{NG}(v_k))$, where each $\mathrm{NG}(v_i)$ appears k times.

In view of Lemma 6, we have $\min_{w_1, w_2 \in V} \delta_{\mathrm{LCS}}(\mathrm{LG}(w_1), \mathrm{NG}(w_2)) \geq c_0$, so we can lower bound $\min_{A \in \mathcal{A}_{k^2,k^2}} \delta(A)$ by $k^2 c_0$.

If $\max_{i,j} \delta_{\mathrm{LCS}}(X_i, Y_j) = c_0$, any alignment has cost $k^2 c_0$. When $\max_{i,j} \delta_{\mathrm{LCS}}(X_i, Y_j) > c_0$, it is easy to observe that in order to achieve $\delta(A) = k^2 c_0$, all sequences in $\mathbf{Y}$ must be aligned (as the cost for any unaligned sequence in $\mathbf{Y}$ is now $> c_0$). Therefore, any alignment A with $\delta(A) = k^2 c_0$ must be $A = \{(i,i) | i \in \{1, 2, \ldots, k^2\}\}$ with $\delta_{\mathrm{LCS}}(X_i, Y_i) = c_0$, for all $i \in \{1, 2, \ldots, k^2\}$.

In view of the above, $\min_{A \in \mathcal{A}_{k^2,k^2}} \delta(A) = k^2 c_0$ iff $\delta_{\mathrm{LCS}}(X_i, Y_i) = c_0$ for all $i \in \{1, 2, \ldots, k^2\}$.

Since $A = \{(i,i) | i \in \{1, 2, \ldots, k^2\}\}$ is structural, $\min_{A \in \mathcal{A}_{k^2,k^2}} \delta(A) = k^2 c_0$ iff $\min_{A \in \mathcal{S}_{k^2,k^2}} \delta(A) = k^2 c_0$. Therefore, in view of Lemma 5, there exists a constant C such that:

- If $\min_{A \in \mathcal{A}_{k^2,k^2}} \delta(A) = k^2 c_0$, then $\delta_{\mathrm{LCS}}(\mathrm{CLG}(t_1), \mathrm{CNG}(t_2)) = k^2 c_0 + C$.
- If $\min_{A \in \mathcal{A}_{k^2,k^2}} \delta(A) > k^2 c_0$, then $\delta_{\mathrm{LCS}}(\mathrm{CLG}(t_1), \mathrm{CNG}(t_2)) > k^2 c_0 + C$.

Moreover, the value of C depends only on $|\mathbf{X}|, |\mathbf{Y}|, \mathcal{T}_X, \mathcal{T}_Y$. As these parameters depend solely on n, k, the number C only depends on n, k.

When $t_1 \cup t_2$ is a $2k$-clique, all vertices in t_1 are adjacent to all vertices in t_2. In view of Lemma 6, $\forall_{i,j} \delta_{\mathrm{LCS}}(X_i, Y_j) = c_0$. Hence $\min_{A \in \mathcal{A}_{k^2,k^2}} \delta(A) = k^2 c_0$, implying that $\delta_{\mathrm{LCS}}(\mathrm{CLG}(t_1), \mathrm{CNG}(t_2)) = k^2 c_0 + C$.

When $t_1 \cup t_2$ is not a $2k$-clique, there exist $u_i \in t_1, v_j \in t_2$ such that $\{u_i, v_j\} \notin E$. According to our definition of $\mathbf{X}$ and $\mathbf{Y}$, we have $X_{j+k(i-1)} = \mathrm{LG}(u_i)$, $Y_{j+k(i-1)} = \mathrm{NG}(v_j)$, and hence $\delta_{\mathrm{LCS}}(X_{j+k(i-1)}, Y_{j+k(i-1)}) > c_0$. This implies that $\min_{A \in \mathcal{A}_{k^2,k^2}} \delta(A) > k^2 c_0$, which leads to $\delta_{\mathrm{LCS}}(\mathrm{CLG}(t_1), \mathrm{CNG}(t_2)) > k^2 c_0 + C$.

As a result, $t_1 \cup t_2$ is a $2k$-clique iff $\delta_{\mathrm{LCS}}(\mathrm{CLG}(t_1), \mathrm{CNG}(t_2)) = k^2 c_0 + C = \min_{t_1', t_2' \in \mathcal{C}_k} \delta_{\mathrm{LCS}}(\mathrm{CLG}(t_1'), \mathrm{CNG}(t_2'))$. Setting $c_1 = k^2 c_0 + C$ suffices. ◀

▶ **Lemma 8.** *There exist four integers $\ell_{CNG,0}$, $\ell_{CNG,1}$, $\ell_{CLG,0}$, $\ell_{CLG,1} \in \mathcal{O}(k^2 n \log(n))$, such that for any $t \in \mathcal{C}_k$,*

- *$\ell_{CNG,b}$ = the number of occurrences of b in $CNG(t)$, $b \in \{0, 1\}$.*
- *$\ell_{CLG,b}$ = the number of occurrences of b in $CLG(t)$, $b \in \{0, 1\}$.*

Proof. As a consequence of Lemma 5, all $\mathrm{CNG}(t)$ have the same type, and all $\mathrm{CLG}(t)$ have the same type. Therefore, the existence of these four integers is guaranteed.

In view of Lemma 5, for all $v \in V$, both $\mathrm{LG}(v)$ and $\mathrm{NG}(v)$ have length at most $(n+1) \cdot (2\lceil \log(n) \rceil + 2\lceil \log(n) \rceil) = \mathcal{O}(n \log(n))$. Applying Lemma 5 again, the length of both $\mathrm{CNG}(t)$ and $\mathrm{CLG}(t)$ for all $t \in \mathcal{C}_k$ is $(k^2 + k^2)(\mathcal{O}(n \log(n)) + \mathcal{O}(n \log(n))) = \mathcal{O}(k^2 n \log(n))$.

As a result, the four integers can be bounded by $\mathcal{O}(k^2 n \log(n))$. ◀

3.2 The RNA sequence S_G

In this subsection, we define the RNA sequence S_G and show that we can decide whether G has a $3k$-clique according to $\mathrm{RNA}(S_G)$.

Based on the parameters in Lemma 8, we define $\ell_0 = \ell_{\mathrm{CNG},0} + \ell_{\mathrm{CNG},1} + \ell_{\mathrm{CLG},0} + \ell_{\mathrm{CLG},1} = \mathcal{O}(k^2 n \log(n))$; for $i \in \{1, 2, 3\}$, we set $\ell_i = 100\ell_{i-1}$; and $\ell_4 = 100|\mathcal{C}_k|\ell_3 = \mathcal{O}(k^2 n^{k+1} \log(n))$.

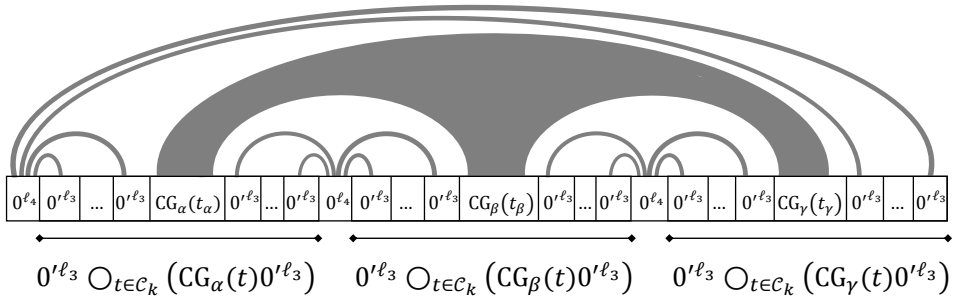


Figure 1 The three selected clique gadgets and the matchings between $0'^{\ell_3}$ and 0^{ℓ_4}.

The RNA sequence S_G is then defined as below:

$$S_G = 0^{\ell_4}\left[0'^{\ell_3} \bigcirc_{t\in\mathcal{C}_k} \left(\mathrm{CG}_\alpha(t)0'^{\ell_3}\right)\right] 0^{\ell_4}\left[0'^{\ell_3} \bigcirc_{t\in\mathcal{C}_k} \left(\mathrm{CG}_\beta(t)0'^{\ell_3}\right)\right] 0^{\ell_4}\left[0'^{\ell_3} \bigcirc_{t\in\mathcal{C}_k} \left(\mathrm{CG}_\gamma(t)0'^{\ell_3}\right)\right],$$

where

$$\mathrm{CG}_\alpha(t) = 1'^{2\ell_2} p(\mathrm{CLG}(t)^R) 0'^{\ell_1} 1^{\ell_2} 0^{\ell_1} \mathrm{CNG}(t) 1^{\ell_2},$$
$$\mathrm{CG}_\beta(t) = 1'^{\ell_2} p(\mathrm{CLG}(t)^R) 0'^{\ell_1} 1'^{2\ell_2} 0'^{\ell_1} p(\mathrm{CNG}(t)) 1'^{\ell_2},$$
$$\mathrm{CG}_\gamma(t) = 1^{\ell_2} \mathrm{CLG}(t)^R 0^{\ell_1} 1^{\ell_2} 0^{\ell_1} \mathrm{CNG}(t) 1^{2\ell_2}.$$

For any $t \in \mathcal{C}_k$, $x \in \{\alpha, \beta, \gamma\}$, the string $\mathrm{CG}_x(t)$ is called a *clique gadget*.
Note that $\mathrm{CG}_\alpha(t) \in (\Sigma \cup \Sigma')^*$, $\mathrm{CG}_\beta(t) \in \Sigma'^*$, and $\mathrm{CG}_\gamma(t) \in \Sigma^*$.
It is obvious that $|S_G| = \mathcal{O}(|\mathcal{C}_k|\ell_0) = \mathcal{O}(k^2 n^{k+1} \log(n))$.

▶ **Lemma 9.** *$RNA(S_G) = f(n,k) - \frac{Q}{2}$, for $Q = \min_{t_\alpha,t_\beta,t_\gamma\in\mathcal{C}_k}(\delta_{LCS}(CLG(t_\alpha), CNG(t_\beta)) + \delta_{LCS}(CLG(t_\alpha), CNG(t_\gamma)) + \delta_{LCS}(CLG(t_\beta), CNG(t_\gamma)))$, and $f(n,k) = 6\ell_2 + 3\ell_1 + \frac{3}{2}\ell_0 + 3(|\mathcal{C}_k|+1)\ell_3 + (|\mathcal{C}_k|-1)(2\ell_1 + 2\ell_2 + \min(\ell_{CLG,1}, \ell_{CNG,1}) + \ell_{CLG,0} + \ell_{CNG,0})$.*

Proof (Sketch). Due to the page limit, we only demonstrate an example of an RNA folding matching this bound, omitting the proof of optimality:

- We link all $0'$ in all $0'^{\ell_3}$ to some 0 in some 0^{ℓ_4} in such a way that all clique gadgets are "blocked" (a clique gadget is blocked if its letters can only link to letters in the same clique gadget or some 0 in some 0^{ℓ_4}) except $\mathrm{CG}_\alpha(t_\alpha)$, $\mathrm{CG}_\beta(t_\beta)$, and $\mathrm{CG}_\gamma(t_\gamma)$. This gives us $3(|\mathcal{C}_k|+1)\ell_3$ amount of pairs. See Fig. 1.
- For a clique gadget that is "blocked", our design of S_G ensures that the optimal number of pairs involving letters in the clique gadget is irrelevant to its corresponding k-clique:
  - For a blocked $\mathrm{CG}_\alpha(t)$, since ℓ_2 is significantly larger than ℓ_1, ℓ_0, an optimal way to pair up the letters is to match as many $\{1', 1\}$ as possible. This gives us $2\ell_2 + \min(\ell_{\mathrm{CLG},1}, \ell_{\mathrm{CNG},1})$ pairs.
  - For a blocked $\mathrm{CG}_\beta(t)$, since we do not have any 1 here, the best we can do is to match all $0'$ to some 0^{ℓ_4}. This gives us $2\ell_1 + \ell_{\mathrm{CLG},0} + \ell_{\mathrm{CNG},0}$ pairs.
  - For a blocked $\mathrm{CG}_\gamma(t)$, no matching can be made.

  The total amount of pairs involving blocked clique gadgets is $(|\mathcal{C}_k|-1)(2\ell_1 + 2\ell_2 + \min(\ell_{\mathrm{CLG},1}, \ell_{\mathrm{CNG},1}) + \ell_{\mathrm{CLG},0} + \ell_{\mathrm{CNG},0})$. See Fig. 2 for an illustration.
- For the three clique gadgets that are not blocked, the matching described in Fig. 3 has cardinality $6\ell_2 + 3\ell_1 + \frac{1}{2}(\ell_0 - \delta_{\mathrm{LCS}}(\mathrm{CLG}(t_\alpha), \mathrm{CNG}(t_\beta))) + \frac{1}{2}(\ell_0 - \delta_{\mathrm{LCS}}(\mathrm{CLG}(t_\alpha), \mathrm{CNG}(t_\gamma))) + \frac{1}{2}(\ell_0 - \delta_{\mathrm{LCS}}(\mathrm{CLG}(t_\beta), \mathrm{CNG}(t_\gamma)))$. Recall that $\frac{1}{2}(\ell_0 - \delta_{\mathrm{LCS}}(\mathrm{CLG}(t_x), \mathrm{CNG}(t_y)))$ is the length of the LCS between $\mathrm{CLG}(t_x)$ and $\mathrm{CNG}(t_y)$. ◀

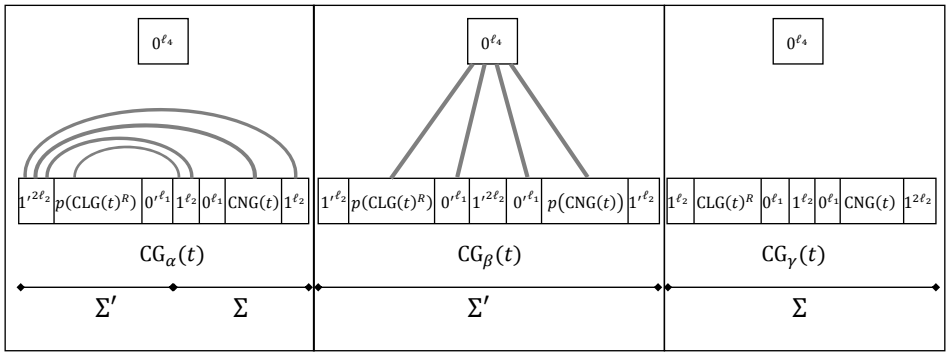


Figure 2 The matchings between a blocked clique gadget and 0^{ℓ_4}.

By Lemma 7, there exists a number c_1 such that:

- the number c_1 depends only on n, k, and $Q \geq 3c_1$.
- If $Q = 3c_1$, then there exist $t_\alpha, t_\beta, t_\gamma \in \mathcal{C}_k$ such that $t_\alpha \cup t_\beta$, $t_\alpha \cup t_\gamma$, $t_\beta \cup t_\gamma$ are three $2k$-cliques. This implies that $t_\alpha \cup t_\beta \cup t_\gamma$ is a $3k$-clique.
- If $Q > 3c_1$, then the graph has no $3k$-clique.

Hence we can decide whether G has a $3k$-clique according to $\mathrm{RNA}(S_G)$, which can be calculated in time $T\left(\mathcal{O}\left(k^2 n^{k+1}\log(n)\right)\right) = \mathcal{O}\left(T\left(n^{k+1}\log(n)\right)\right)$ (k is a constant, and $T(\cdot)$ is the time complexity of computing optimal RNA folding). Theorem 2 is concluded.

4 Hardness of Dyck Edit Distance Problem

In this section, we shift our focus to the Dyck edit distance problem. We will present a simple reduction from RNA folding problem (with alphabet size 4) to Dyck edit distance problem (with alphabet size 10). This leads to a much simplified and improved proof for a conditional lower bound of Dyck edit distance based on the conjectured hardness k-clique. Recall that the previous proof in [1] requires 48 symbols.

Given $S \in (\Sigma \cup \Sigma')^n$, the goal of the Dyck edit distance problem is to find a minimum number of edit operations (insertion, deletion, and substitution) that transform S into a string in the Dyck context free language defined by the grammar: $\mathbf{S} \to \mathbf{SS}$, $\forall x \in \Sigma, \mathbf{S} \to x\mathbf{S}x'$, and $\mathbf{S} \to \epsilon$ (empty string).

An alternative definition of the Dyck edit distance problem is described as follows: Given a sequence $S \in (\Sigma \cup \Sigma')^n$, find a minimum cost set $A \subseteq \{(i,j) | 1 \leq i < j \leq n\}$ satisfying the following conditions:

- $A = A_M \uplus A_S$ has no crossing pair.
- A_M contains only pairs of the form (x, x'), $x \in \Sigma$ (i.e. for all $(i,j) \in A_M$, we have $S[i] = x$, $S[j] = x'$, for some $x \in \Sigma$). A_M corresponds to the set of matched pairs.
- A_S does not contain any pair of the form (y', x), $x, y \in \Sigma$ (i.e. for all $(i,j) \in A_S$ we have either $S[i] \in \Sigma$ or $S[j] \in \Sigma'$). A_S corresponds to the set of pairs that can be fixed by one substitution operation per each pair.
- Let D be the set of letters in S that do not belong to any pair in A. Each letter in D requires one deletion/insertion operation to fix.

The cost of A is then defined as $|A_S| + |D|$, and the Dyck edit distance of the string S is the cost of a minimum cost set meeting the above conditions.

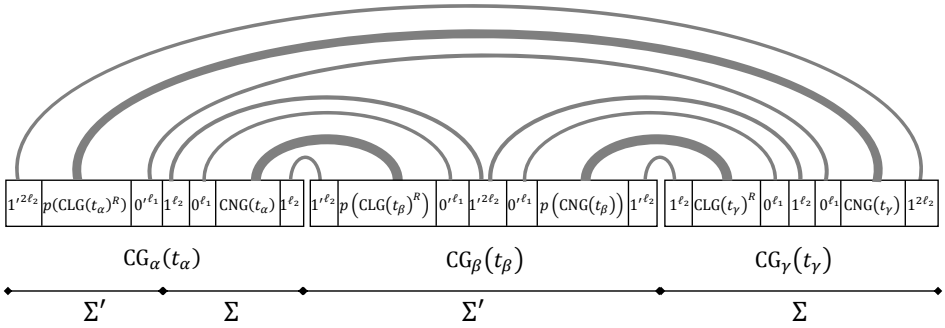


Figure 3 The matchings within the three selected clique gadgets.

We can view Dyck edit distance problem as an asymmetric version of RNA folding (both (x, x') and (x', x) are legit aligned pairs in RNA folding) that also handles substitution (in addition to deletion and insertion). Intuitively, Dyck edit distance is more complicated than RNA folding. Indeed, the same conditional lower bound as Theorem 1 for Dyck edit distance problem shown in [1] requires a bigger alphabet size (48 instead of 36) and a longer proof. In the next, we prove Theorem 3 by showing a simple reduction from RNA folding to Dyck edit distance with alphabet size 10. This improves upon the hardness result in [1], and justifies the intuition that Dyck edit distance is a harder problem than RNA folding.

Proof of Theorem 3. For notational simplicity, we let the alphabet for the RNA folding problem be $\Sigma \cup \Sigma' = \{0, 0', 1, 1'\}$ (instead of $\{A, C, G, U\}$). Let S be any string in $(\Sigma \cup \Sigma')^n$. We define the string S_{Dyck} as the result of applying the following operations on S:

- Replace each letter 0 with the sequence $S_0 = aeb'aeb'$.
- Replace each letter $0'$ with the sequence $S_{0'} = bba'a'$.
- Replace each letter 1 with the sequence $S_1 = ced'ced'$.
- Replace each letter $1'$ with the sequence $S_{1'} = ddc'c'$.

It is clear that S_{Dyck} is a sequence of length at most $6n$ on the alphabet $\{a, b, c, d, e\} \cup \{a', b', c', d', e'\}$, though the letter e' is not used. We claim that the Dyck edit distance of S_{Dyck} is $\frac{|S_{\text{Dyck}}|}{2} - 2\text{RNA}(S)$.

First, we show that the Dyck edit distance of S_{Dyck} is at most $\frac{|S_{\text{Dyck}}|}{2} - 2\text{RNA}(S)$. Given an optimal RNA folding of S, we construct a crossing-free matching A with cost $\frac{|S_{\text{Dyck}}|}{2} - 2\text{RNA}(S)$ as follows:

- For matched pairs in the RNA folding of S:
  - For each matched pair $(0, 0')$ in the RNA folding of S, we add two pairs $(a, a'), (a, a')$ to A_M, and add three pairs $(e, b'), (e, b'), (b, b)$ to A_S in its corresponding pair of substrings $(S_0 = \mathbf{a}(eb')\mathbf{a}(eb'), S_{0'} = (bb)\mathbf{a'a'})$ in S_{Dyck}.
  - For each matched pair $(0', 0)$ in the RNA folding of S, we add two pairs $(b, b'), (b, b')$ to A_M, and add three pairs $(a', a'), (a, e), (a, e)$ to A_S in its corresponding pair of substrings $(S_{0'} = \mathbf{bb}(a'a'), S_0 = (ae)\mathbf{b'}(ae)\mathbf{b'})$ in S_{Dyck}.
  - Similarly, for each matched pair $(1, 1'), (1', 1)$ in the RNA folding of S, we can add two pairs to A_M and three pairs to A_S.
- For unmatched letters in S:
  - For each unmatched letter 0 in S, we add three pairs $(a, b'), (e, b'), (a, e)$ to A_S in its corresponding substring $S_0 = (a(eb')(ae)b')$. Similarly, for each unmatched letter 1, we can add three pairs to A_S.

- For each unmatched letter $0'$ in S, we add two pairs $(b,b),(a',a')$ to A_S in its corresponding substring $S_0 = (bb)(a'a')$. Similarly, for each unmatched letter $1'$, we can add two pairs to A_S.

The set A_M has size $2\text{RNA}(S)$, the set A_S has size $\frac{|S_{\text{Dyck}}|-4\text{RNA}(S)}{2}$, and D is an empty set. Therefore, the cost of A is $\frac{|S_{\text{Dyck}}|-4\text{RNA}(S)}{2} = \frac{|S_{\text{Dyck}}|}{2} - 2\text{RNA}(S)$.

Second, we show that the Dyck edit distance of S_{Dyck} is at least $\frac{|S_{\text{Dyck}}|}{2} - 2\text{RNA}(S)$. Given a crossing-free matching A (on the string S_{Dyck}) of cost C, we recover an RNA folding of S that has $\geq \frac{|S_{\text{Dyck}}|}{4} - \frac{C}{2}$ number of matched pairs.

We build a multi-graph $G = (V, E)$ such that V is the set of all substrings $S_0, S_{0'}, S_1, S_{1'}$ that constitute S_{Dyck}, and the number of edges between two substrings in V is the number of pairs in A_M linking letters between these two substrings. Note that $|V| = n, |E| = A_M$. It is clear that $C \geq \frac{|S_{\text{Dyck}}|-2|E|}{2}$, since $|A_S| + |D| \geq \frac{|S_{\text{Dyck}}|-2|A_M|}{2} = \frac{|S_{\text{Dyck}}|-2|E|}{2}$. Therefore, we are done if we can recover an RNA folding of size $\geq \frac{|E|}{2}$, since $\frac{|E|}{2} \geq \frac{|S_{\text{Dyck}}|}{4} - \frac{C}{2}$.

We observe the following:

- G has degree at most 2 (due to our definition of $S_0, S_{0'}, S_1, S_{1'}$, at most two letters in such a substring can participate in pairings of the form (x, x'), $x \in \{a, b, c, d\}$, without crossing).
- In the graph G, any edge must either links an S_0 with an $S_{0'}$ or links an S_1 with an $S_{1'}$ (due to our definition of $S_0, S_{0'}, S_1, S_{1'}$, any pairing of the form (x, x'), $x \in \{a, b, c, d\}$, must be made between $S_0, S_{0'}$ or between $S_1, S_{1'}$).
- G does not contain any cycle of odd length (due to the above observation).

In view of the above second observation, a (graph-theoretic) matching $M \subseteq E$ of G naturally corresponds to a (size $|M|$) RNA folding of S: for each edge (a pair of substrings in S_{Dyck}) in M, we add its corresponding pair of letters in S to the RNA folding. Since a maximum matching has size $\geq \frac{|E|}{2}$ in a graph of maximum degree 2 without odd cycles, we conclude the proof. ◀

We note that for the case substitution is not allowed, the letter e in the above proof is not needed, and this lowers the required alphabet size to 8.

Acknowledgements. The author thanks Seth Pettie and anonymous reviewers for helpful comments.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is Valiant's parser. In *Proceedings of 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 98–117, 2015.

2 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *Proceedings of 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 59–78, 2015.

3 Tatsuya Akutsu. Approximation and exact algorithms for RNA secondary structure prediction and recognition of stochastic context-free languages. *Journal of Combinatorial Optimization*, 3(2):321–336, 1999.

4 Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In *Proceedings of 41st International Colloquium Automata, Languages, and Programming (ICALP)*, pages 114–125, 2014.

5 Amihood Amir and Gad M. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81(1):97–115, 1991.

6 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of 47th Annual ACM Symposium on Theory of Computing (STOC)*, pages 51–58, 2015.

7 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 79–97, 2015.

8 Timothy M. Chan and Moshe Lewenstein. Clustered integer 3SUM via additive combinatorics. In *Proceedings of 47th Annual ACM Symposium on Theory of Computing (STOC)*, pages 31–40, 2015.

9 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.

10 Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, 326(1):57–67, 2004.

11 Yelena Frid and Dan Gusfield. A simple, practical and complete $\mathcal{O}(\frac{n^3}{\log n})$-time algorithm for RNA folding using the Four-Russians speedup. *Algorithms for Molecular Biology*, 5(1):1–8, 2010.

12 Tamar Pinhas, Dekel Tsur, Shay Zakov, and Michal Ziv-Ukelson. Edit distance with duplications and contractions revisited. In *Proceedings of 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 441–454. Springer Berlin Heidelberg, 2011.

13 Tamar Pinhas, Shay Zakov, Dekel Tsur, and Michal Ziv-Ukelson. Efficient edit distance with duplications and contractions. *Algorithms for Molecular Biology*, 8(1):1–28, 2013.

14 Mihai Pătraşcu and Ryan Williams. On the possibility of faster SAT algorithms. In *Proceedings of 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1065–1075, 2010.

15 Liam Roditty and Virginia Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of 45th ACM Symposium on Theory of Computing (STOC)*, pages 515–524, 2013.

16 Barna Saha. The Dyck language edit distance problem in near-linear time. In *Proceedings of 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 611–620, 2014.

17 Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In *Proceedings of 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 118–135, 2015.

18 Yinglei Song. Time and space efficient algorithms for RNA folding with the Four-Russians technique. Technical Report arXiv:1503.05670, 2015.

19 Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308–315, 1975.

20 Virginia Vassilevska. Efficient algorithms for clique problems. *Information Processing Letters*, 109(4):254–257, 2009.

21 Balaji Venkatachalam, Dan Gusfield, and Yelena Frid. Faster algorithms for RNA-folding using the Four-Russians method. *Algorithms for Molecular Biology*, 9(1):1–12, 2014.

Efficient Non-Binary Gene Tree Resolution with Weighted Reconciliation Cost

Manuel Lafond[1], Emmanuel Noutahi[2], and Nadia El-Mabrouk[3]

1 **DIRO, Université de Montréal, H3C 3J7, Canada**
`lafonman@iro.umontreal.ca`
2 **DIRO, Université de Montréal, H3C 3J7, Canada**
`noutahie@iro.umontreal.ca`
3 **DIRO, Université de Montréal, H3C 3J7, Canada**
`mabrouk@iro.umontreal.ca`

Abstract

Polytomies in gene trees are multifurcated nodes corresponding to unresolved parts of the tree, usually due to insufficient differentiation between sequences. Resolving a multifurcated tree has been considered by many authors, the objective function often being the number of duplications and losses reflected by the reconciliation of the resolved gene tree with a given species tree. Here, we present PolytomySolver, an algorithm accounting for a more general model allowing for costs that can vary depending on the operation, but also on the considered genome. The time complexity of PolytomySolver is linear for the unit cost and is quadratic for the general cost, which outperforms the best known solutions so far by a linear factor. We show, on simulated trees, that the gain in theoretical complexity has a real practical impact on running times.

1998 ACM Subject Classification Biology and genetics

Keywords and phrases gene tree, polytomy, reconciliation, resolution, weighted cost, phylogeny

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.14

1 Introduction

Reconstructing gene trees is a fundamental task in bioinformatics and a prerequisite for most biological studies on gene function. Consequently, a plethora of phylogenetic methods have been developed, most of them integrating measures of statistical support (e.g. by bootstrapping or jackknifing), reflecting the confidence we have on the prediction. Some of them, such as bayesian methods [9, 11] lead to non-binary trees. Moreover, weakly supported branches are often contracted and also lead to non-binary trees. Thus, although unresolved nodes in a tree may reflect a true (or *hard* [12]) simultaneous speciation or duplication event leading to more than two gene copies, they are usually artifacts (called *soft*), due to methodological reasons or to a lack of resolution between sequences.

Information for the full resolution of a gene tree may rely on the weakly exploited link between gene and species evolution. The question of resolving a non-binary gene tree by minimizing the number of duplications and losses resulting from the reconciliation of the gene tree with the species tree has first been considered in NOTUNG [2] and later by Chang and Eulenstein [1]. In 2012 [8], we developed the first linear-time algorithm for resolving a polytomy (a single unresolved node), leading to a quadratic-time algorithm for a whole tree. Recently, algorithmic results extending linearity to a whole gene tree have been obtained by Zheng and Zhang [15]. These linearity results are however restricted to the case of a unit cost for duplications and losses. On the other hand, an algorithm allowing different costs for

Table 1 Time-complexity results for reporting a single optimal resolution of a whole gene tree G of size $|G|$ with a species tree S of size $|S|$, where Δ is the largest degree of a node in G, δ is the cost of a duplication and λ the cost of a loss. The last column refers to the case in which each species s has its own duplication cost δ_s and loss cost λ_s.

	$\delta = \lambda = 1$	$(\delta, \lambda) \in \mathbb{R}_{>0} \times \mathbb{R}_{>0}$	$\{(\delta_s, \lambda_s)\}_{s \in V(S)}$
NOTUNG[6]	$O(\|S\|\|G\|\Delta^2)$	$O(\|S\|\|G\|\Delta^2)$	
Lafond[8]	$O(\|S\|\|G\|)$	$O(\|S\|\|G\|\Delta)$	
Zheng & Zhang[15]	$O(\|G\|)$	$O(\|G\|\Delta^2)$	
PolytomySolver	$O(\|G\|)$	$O(\|G\|\Delta)$	$O(\|G\|\|S\|\Delta)$

duplications and losses has been considered in NOTUNG [6], and further improved by Zheng and Zhang [15], using a compressed species tree idea.

In this paper, we present a new algorithm called PolytomySolver, which handles unit costs in linear time and improves the best complexity to date for more general duplication and loss cost model by a linear factor (complexity results are given in Table 1). Additionally, PolytomySolver is the first algorithm enabling to account for various evolutionary rates across the branches of a species tree, as it allows assigning each taxa its specific duplication and loss cost. This functionality may be used to reduce the effect of missing data by assigning a lower loss cost to species that are more likely to be concerned by such loss of information. It is also of practical use when biological evidence supports some particularly low or high gene duplication or loss rates in some species of interest [10]. In particular, fractionation following whole genome duplication (WGD) results in an excess of gene losses. In Section 6, we give an example showing that assigning appropriate costs to post-WGD genomes is important for an accurate inference.

The paper is subdivided as follows. First, in Section 3, we show how the linear-time algorithm developed previously by our group [8] for resolving a polytomy with unit duplication and loss cost can be extended to arbitrary costs, depending on the operation and on the genome affected by the operation. This extension is however not linear anymore but rather leads to a cubic-time algorithm. We then, in Section 4, show how using the ideas introduced by Zheng and Zhang [15] allows to reduce this time complexity to quadratic, which is the best obtained to date for the same problem. We also show how unit costs can be handled in linear time, and how PolytomySolver can be used to output all optimal resolutions, which is an advantage compared to Zheng and Zhang's algorithms. In Section 5, comparing our new algorithm with NOTUNG and Zheng and Zhang's algorithm, we show that the obtained gain in theoretical complexity actually leads to a significant gain in running times. For space reason, all proofs are given in Appendix, which is available online at `http://www-ens.iro.umontreal.ca/~lafonman/en/publications.php`.

2 Preliminary

All trees are considered to be rooted. Given a set X, a *tree T for X* has its leafset $\mathcal{L}(T)$ in bijection with X. Denote by $V(T)$ its set of nodes, $r(T)$ its root, and write $|T| = |V(T)|$. Given two nodes x and y of T, x is a *descendant* of y, and y is an *ancestor* of x, if y is on the (inclusive) path between x and $r(T)$. The *degree* $deg(x)$ of a node x is the number of edges incident to x. The maximum degree of T is $\Delta(T) = \max_{v \in V(T)} deg(v)$ (or just Δ when T is clear from the context). Given a set L of leaves, the *lowest common ancestor* of L in

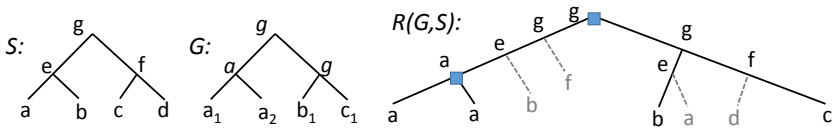


Figure 1 S is a species tree over $\Sigma = \{a, b, c, d\}$; G is a gene tree on the gene family Γ with two copies in genome a, one in genome b and one in genome c; $R(G, S)$ is a reconciliation of G with S with two duplications and four losses. Each node x of G and $R(G, S)$ is labeled by $s(x)$.

T, denoted $lca_T(L)$, is the common ancestor of L in T that is farthest from the root. A *polytomy* (or star tree) over a set L is a tree with a single internal node, which is of degree $|L|$, adjacent to each leaf of L. Finally, if x is a node of T, denote by T_x the subtree of T rooted at x, and by $T(x)$ the polytomy obtained by keeping only x and its children in T_x.

2.1 Gene Tree, Species Tree and Reconciliation

A species tree S for a set $\Sigma = \{\sigma_1, \cdots, \sigma_t\}$ of species represents an ordered set of speciation events that have led to Σ. Inside the species' genomes, genes undergo speciations when the species to which they belong do, but also duplications and losses (other events such as transfers can happen, but we ignore them here). A *gene family* is a set Γ of genes where each gene x belongs to a given species $s(x)$ of Σ. The evolutionary history of Γ can be represented as a *gene tree* G where $\mathcal{L}(G)$ is in bijection with Γ, and each internal node refers to an ancestral gene at the moment of an event (either speciation or duplication) belonging to the species $s(x) = lca_S(\{s(y) : y \in \mathcal{L}(G_x)\})$. We denote $\mathcal{S}(G) = \{s(y) : y \in \mathcal{L}(G)\}$ the set of species *represented by* G.

In this paper, we make no distinction between paralogous gene copies. In other words, a gene x is simply identified by the genome $s(x)$ it belongs to. A gene tree is therefore a tree where each leaf is labeled by an element of Σ, with possibly repeated leaf labels (Figure 1).

A *reconciliation* is an extension of the gene tree, obtained by adding lost branches, reflecting a history of duplications and losses in agreement with the species tree. Formally, an *extension* of G is a tree obtained from G by a sequence of graftings, where a *grafting* consists in subdividing an edge uv of G, thereby creating a new node w between u and v, then adding a leaf x with parent w. The new leaf x is mapped to a species $s(x)$ which is a node of S (internal or leaf). A formal definition follows (see Figure 1 for an example).

▶ **Definition 1** (Reconciled gene tree)**.** Let G be a binary gene tree and S be a binary species tree. A *reconciliation* $R(G, S)$ of G with S is an extension of G verifying: for each internal node x of $R(G, S)$ with two children x_l and x_r, either $s(x_l) = s(x_r) = s(x)$, or $s(x_l)$ and $s(x_r)$ are the two children of $s(x)$. The node x is a duplication in $s(x)$ in the former case, and a speciation node in $s(x)$ in the latter case. A grafted leaf x corresponds to a loss in $s(x)$.

Define δ_s as the duplication cost and λ_s as the loss cost assigned to a given species s. Then, the *reconciliation cost* of $R(G, S)$ is the sum of costs of the induced duplications and losses.

2.2 Problem statement

We consider a binary species tree S and a non-binary gene tree G. The goal is to find a *binary refinement* of G, as defined below.

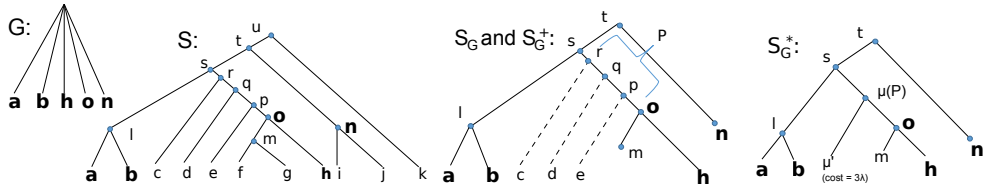


Figure 2 From left to right: a gene tree G; a species tree S; the species tree S_G linked to G is the tree illustrated by plain lines, and the augmented species tree S_G^+ linked to G is illustrated by plain and dotted lines; the compressed tree S_G^* linked to G as defined in Section 4. The leaf μ' of S_G^* has a special loss cost $\lambda_{\mu'} = 3$, as it results from the contraction of a path of length 3.

▶ **Definition 2** (binary refinement)**.** A *binary refinement* $B = B(G)$ of G is a binary tree such that $V(G) \subseteq V(B)$ and for every $x \in V(G)$, $\mathcal{L}(G_x) = \mathcal{L}(B_x)$.

The objective function taken for choosing among all possible binary refinements is the reconciliation cost.

▶ **Definition 3** (Resolution)**.** A *resolution* of G with respect to S is a reconciliation $R(B, S)$ between a binary refinement B of G and S. The set of all possible resolutions of a tree G is denoted $\mathcal{R}(G)$.

We are now ready to state our optimization problem.

Minimum Resolution Problem
Input: A binary species tree S and a non-binary gene tree G.
Output: A *Minimum Resolution* of G with respect to S (or simply *Minimum Resolution of* G), e.g. a resolution of G of minimum reconciliation cost with respect to S.

It has been previously shown [1] that each polytomy of G can be considered independently. In particular, a minimum resolution of G can be obtained by a depth-first procedure that solves each polytomy $G(x)$ iteratively, for each internal node x of G. Thus, in the following, we focus on a single polytomy $G = G(x)$.

Some parts of the species tree can be ignored in the process of refining G. Define the *species tree linked to* G, denoted by S_G, as the tree obtained from the subtree of S rooted at the lowest common ancestor of $\mathcal{S}(G)$, by removing all nodes that have no descendant in $\mathcal{S}(G)$ (Figure 2). The algorithms with the best known complexity results (Table 1) are obtained by using a compressed version S_G^* of this tree, which is defined in Section 4. We first begin, in Section 3, by describing the refinement strategy by using an *augmented species tree linked to* G, denoted S_G^+, obtained from S_G by adding to every node of degree two its missing child in S. It is known (c.f. [8, 15]) that resolving G with either S or S_G^+ leads to the same reconciliation cost. Intuitively, S_G^+ contains every node of S that may appear in a resolution of G, whether as a loss, a duplication or a speciation.

3 A dynamic programming approach

We present a dynamic programming approach for the Minimum Resolution Problem for a single polytomy G. It is a generalization of that presented in [8]. While the previous algorithm was developed for a unit cost of duplications and losses, the one we present here

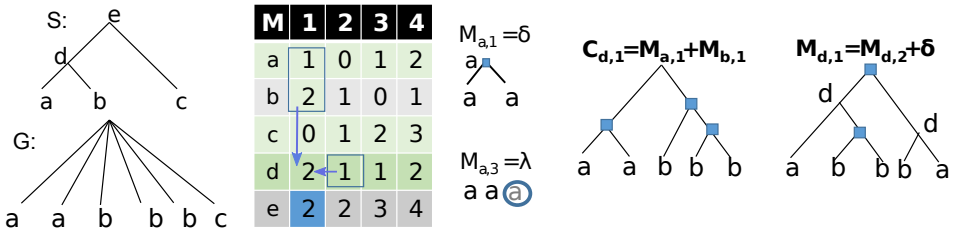


Figure 3 A polytomy G and a species tree S. The corresponding table M is obtained for $\delta_s = \lambda_s = 1$ for all species. Squares on trees illustrate duplications. To the right of table M, the forests corresponding to an $(a, 1)$ and $(a, 3)$-resolution are given, where the circled a illustrates a singleton loss. We illustrate the $(d, 1)$-resolution, rooted at a speciation node, corresponding to $C_{d,1} = 3$ (obtained from the vertical arrow in table M), and an optimal $(d, 1)$-resolution, obtained from a $(d, 2)$-resolution (horizontal arrow in M).

holds for a more general reconciliation cost, where each $s \in \Sigma$ has its own duplication cost δ_s and loss cost λ_s. In this section, we assume that $S = S_G^+$.

The recursion is made on the subtrees of S. Define the multiplicity $m(s)$ of $s \in V(S)$ in G as the number of times it appears in G, i.e. $m(s) = |\{x \in \mathcal{L}(G) : s(x) = s\}|$. An (s, k)-*resolution* of G is a forest of k reconciled gene trees $\mathcal{T} = \{T_1, \ldots, T_k\}$ such that, for each $1 \leq i \leq k$, $s(r(T_i)) = s$, and each leaf x of G with $s(x)$ being a descendant of s is present as a leaf of some tree of $\mathcal{T}$ (see Figure 3 for an example). All leaves of trees in $\mathcal{T}$ that are not in $\mathcal{L}(G)$ represent losses. Also, some trees of $\mathcal{T}$ may be restricted to a single node which is either a child x of $r(G)$ with $s(x) = s$, or a singleton loss in s. The cost of $\mathcal{T}$, denoted $c(\mathcal{T})$, is the sum of reconciliation costs of all T_is. Notice that since $S = S_G^+$, a resolution of G is an $(r(S), 1)$-*resolution*.

Denote by $M_{s,k}$ the minimum cost of an (s, k)-resolution for a given node s of S and a given integer $k \geq 1$ (and $M_{s,k} = \infty$ for $k < 1$). The final cost of a minimum resolution of G is given by $M_{r(S),1}$. The table M is computed, line by line, for all nodes of S, in a bottom-up traversal. For now, k is unlimited, but we show in the complexity section that there is no need to consider more than $|G| - 1$ columns.

The following lemma gives the base case for the leaves of S. It follows from the fact that, if k is larger than the number of available leaves, then additional leaves have to be added (called *singleton losses*); otherwise leaves have to be joined under duplication nodes. As an illustration, in Figure 3, this lemma is used to compute the three first lines of M.

▶ **Lemma 4** (Base case). *For a leaf node s of S, if $k > m(s)$ then $M_{s,k} = \lambda_s \cdot (k - m(s))$; otherwise $M_{s,k} = \delta_s \cdot (m(s) - k)$.*

The rest of this section focuses on the computation of a line M_s of M for an internal node s of S, from the lines M_{s_l} and M_{s_r}, where s_l and s_r are the two children of s in S. We require an intermediate cost table $C_{s,k}$, defined for internal nodes of S, accounting only for speciation events. That is, $C_{s,k}$ represents the minimum cost of an (s, k)-resolution in which every tree is rooted at a speciation node with two children (these two children may both be losses), or consists of a singleton node that is a child of $r(G)$ already mapped to s. For $k > m(s)$, such an (s, k)-resolution of cost $C_{s,k}$ can only be obtained from an $(s_l, k - m(s))$-resolution and an $(s_r, k - m(s))$-resolution by creating $k - m(s)$ speciation nodes, each joining a pair of (s_l, s_r) trees, then adding the $m(s)$ singleton trees mapped to s. No other scenarios are possible, since (s, k)-resolutions are reconciled trees, and each non-singleton root is a speciation in s

that must have genes mapped to s_l and s_r as children. See for example the $(d, 1)$-resolution corresponding to $C_{d,1}$ in Figure 3. Note that if instead $k \leq m(s)$, such an (s, k)-resolution cannot exist, since $m(s)$ trees are required for the children of $r(G)$ mapped to s, plus at least another tree containing the genes in a descendant of s. Thus we define:

$$C_{s,k} = M_{s_l,k-m(s)} + M_{s_r,k-m(s)} \text{ if } k > m(s) \text{ and } C_{s,k} = +\infty \text{ otherwise} \quad \textbf{(1)}$$

It is readily seen that $M_{s,k} \leq C_{s,k}$. A recurrence for computing $M_{s,k}$ follows.

▶ **Lemma 5.** *For an internal node s of S, $M_{s,k} = \min(M_{s,k-1} + \lambda_s, M_{s,k+1} + \delta_s, C_{s,k})$.*

This recurrence cannot be used as such to compute C and M, as it induces both a left and right dependency. That is, $M_{s,k}$ depends on $M_{s,k+1}$ and vice-versa, leading to a chicken-and-egg problem as to which value should be computed first. In the case of a unit cost $\delta_s = \lambda_s = 1$ for all s, we have shown in [8] that this dependency can be avoided by considering a strong property on lines of M. Indeed, each line M_s is characterized by two values k_1 and k_2 such that, for any $k_1 \leq k \leq k_2$, $M_{s,k}$ is minimum, for any $k \leq k_1$, $M_{s,k-1} = M_{s,k} + 1$, and for any $k \geq k_2$, $M_{s,k+1} = M_{s,k} + 1$. In other words, M_s has a slope of -1 until k_1, a slope of 0 until k_2, then a slope of 1. In particular, M_s can be treated as a convex function fully determined by k_1, k_2 and its minimum value γ. We then say M_s has a *minimum plateau* between k_1 and k_2. For example, line M_d in Figure 3 is fully determined by $k_1 = 2$ and $k_2 = 3$.

Here, we extend these results by first showing, in Lemma 7, that both C and M are still convex, albeit having less predictable changes in the slopes. Nevertheless, this allows to first compute the bounds k_1 and k_2 of the functions' minimum plateau, and then extend to the left and to the right from this plateau.

We first recall the formal definition of a discrete convex function, then state the convexity result for C and M and finally give the recurrences of the dynamic programming algorithm in Theorem 8.

▶ **Definition 6** (Convex function)**.** A discrete function f is convex if and only if, for any integer $n > 1$, the two following statements, which are equivalent, are true.

- $f(n+1) + f(n-1) - 2f(n) \geq 0$;
- for any integers $\epsilon_1, \epsilon_2 > 0$ and any integer $n > \epsilon_1$, $f(n - \epsilon_1) + f(n + \epsilon_2) - 2f(n) \geq 0$.

▶ **Lemma 7.** *Both M_s and C_s are convex.*

▶ **Theorem 8** (Recurrence 2)**.** *Let k_1 and k_2 be the smallest and largest values, respectively, such that $C_{s,k_1} = C_{s,k_2} = \min_k C_{s,k}$. Then,*

$$M_{s,k} = \begin{cases} C_{s,k} & \text{if } k_1 \leq k \leq k_2 \\ \min(C_{s,k}, M_{s,k+1} + \delta_s) & \text{if } k < k_1 \\ \min(C_{s,k}, M_{s,k-1} + \lambda_s) & \text{if } k > k_2 \end{cases}$$

Theorem 8 provides the way for computing a row M_s for an internal node s of S: for each k, compute $C_{s,k}$ using recurrence **(1)** and keep the two columns k_1 and k_2 setting the bounds of the convex function's plateau. Extend to the left of k_1 using $M_{s,k} = \min(C_{s,k}, M_{s,k+1} + \delta_s)$, and to the right of k_2 using $M_{s,k} = \min(C_{s,k}, M_{s,k-1} + \lambda_s)$. These recurrences, with the base case for S leaves given in Lemma 4, describe the dynamic programming algorithm, that we call PolytomySolver, for computing the cost $M_{r(S),1}$ of a minimum resolution of the polytomy G with respect to S. We refer the reader to [8] for the reconstruction of a solution from M in linear time, which is accomplished using a standard backtracking procedure.

Complexity

The following lemma states that there is no reason to explore more gene copies of a given species than the size of the polytomy, in other words, the size of a line of M can be bounded by $|G|$. This fact may seem obvious to the accustomed, but in [6] it was equally "obvious" that only $m^* = \max_{s \in V(S)} m(s)$ columns needed to be considered, which turns out to be wrong [1]. In fact, this Lemma requires a surprising amount of care in the details (see Appendix).

▶ **Lemma 9.** *Only the values of M and C for columns k between 1 and $|G| - 1$ need to be computed.*

It follows from Lemma 4, Theorem 8 and Lemma 9 that each row of C and M can be computed in time $O(|G|)$, and the whole table in time $O(|S||G|)$.

Now suppose that H is a general tree with p polytomies, where Δ is the largest degree of a polytomy. According to the depth-first procedure described at the end of Section 2, G can be resolved in time $O(p|S|\Delta)$, which is less than $O(|H||S|\Delta)$. In the next section, we improve this to $O(|H|\Delta)$ in the case of distinct costs δ and λ that are shared across all species, and $O(|H|)$ in the case of equal costs $\delta = \lambda$.

4 A faster algorithm using species tree compression

Assume that all species have the same duplication cost δ and the same loss cost λ. We call it *unit cost* if $\delta = \lambda$, and *general cost* otherwise. Again we assume that G is a polytomy.

In the previous section, results have been obtained using the augmented linked species tree S_G^+. As observed by Zheng and Zhang [15], S_G^+ contains many "useless" nodes that do not provide any meaningful information with regards to the resolution of G. This idea allowed them to optimize their refinement algorithm for the unit cost, leading to a linear-time algorithm. However, their algorithm does not apply to the general cost. For such a cost, their optimisation idea was rather applied to the NOTUNG's algorithm, which is less efficient. Here, we use a similar idea to optimize PolytomySolver. More precisely, we show how a compressed version of the linked species tree S_G can be used to reduce the complexity for refining a general tree G to $O(|G|\Delta)$ for the general cost, and to $O(|G|)$ for the unit cost.

We first need some definitions. Let T be a tree. Call P a *path in* T if P is a sequence of non-root adjacent vertices of degree two in T. *Contracting* P in T consists in replacing P by a single node $\mu = \mu(P)$. Now, let U be the set of non-root vertices of degree two of S_G that are not in $\mathcal{S}(G)$. We call U the set of "useless nodes" of S_G. Notice that $S_G[U]$, the graph obtained from S_G by keeping only nodes of U and edges with both endpoints in U, corresponds to a set of disjoint paths in S_G. The *compressed tree* S_G^* is the tree obtained from S_G by contracting every path P of $S_G[U]$ to $\mu = \mu(P)$, then adding a leaf child μ' to every such μ (see Figure 2 for an example). Moreover, we set a special loss cost $\lambda_{\mu'} = \lambda|P|$ to μ' (and duplication cost δ as every other node). This special loss cost ensures that a loss in μ' is counted as a loss in every node in P. Notice that some internal nodes of S_G that are included in $\mathcal{S}(G)$ may still have only one child. Thus S_G^* is finally obtained by adding to each remaining node having only one child a new leaf child (duplication of cost δ and loss cost λ). The following Theorem ensures that S_G^* does not change the solution space.

[1] The complexity reported in Table 1 is not the one reported by NOTUNG, as dependency is not given on Δ but instead on m^*. However, it can be shown that considering m^* columns is not enough on some examples.

▶ **Theorem 10.** *Let T be a binary refinement of G. Then the reconciliation cost of T is the same whether we reconcile it with S_G^+ or S_G^* and their corresponding duplication/loss costs.*

Thus, using S_G^+ or S_G^* leads to the same minimum resolution for G. We show that using S_G^* leads to reduction in time complexity of the algorithm.

▶ **Theorem 11.** *Given a gene tree H,* PolytomySolver *can run in time $O(\Delta|H|)$.*

4.1 The case of a unit cost

In [8], we showed how, in the case of a unit cost $\delta = \lambda$, each line M_s of M can be computed in constant time. However, in order to take advantage of the compressed species tree $S = S_G^*$, we need to account for special leaves μ' with loss cost $\lambda_{\mu'} > 1$, since they make the cost not unitary anymore. The following theorem allows us to extend the result to this specific case. It leads to the computation of M in time $O(|S_G^*|) = O(|G|)$ for a polytomy G. The complexity for a gene tree H is thus reduced to $O(|H|)$, which results in a reduction of the previous complexity by a factor of Δ.

▶ **Theorem 12.** *Suppose $S = S_G^*$. Then for $s \in V(S)$,*
1. *if s is a leaf with loss cost $\lambda = 1$, then $M_{s,k} = |k - m(s)|$;*
2. *if s is a leaf with loss cost $\lambda_s > 1$, then $M_{s,k} = k \cdot \lambda_s$;*
3. *if s is an internal node, there exist 3 integers k_1, k_2 and γ_s such that*

$$M_{s,k} = \begin{cases} \gamma_s & \text{if } k_1 \leq k \leq k_2 \\ \gamma_s + k_1 - k & \text{if } k < k_1 \\ \gamma_s + k - k_2 & \text{if } k > k_2 \end{cases}$$

Moreover, k_1, k_2 and γ_s can be computed in constant time.

4.2 Constructing all minimum resolutions

After computing table M, it remains to compute $(r(S), 1)$-resolutions, i.e. all resolutions of minimum cost. Without any increase in the theoretical time complexity of the algorithm, a simple pass through table M leads to one minimum resolution (see [8] for the details). Here we rather show how to recover all minimum resolution.

Denote by $\mathcal{P}_{s,k}$ the set of all minimum (s,k)-resolutions of a polytomy G. By setting $s = r(S)$ and $k = 1$, we exhibit the following recursive algorithm that finds $\mathcal{P}_{r(S),1}$. To do so, we define three intermediate solution sets $\mathcal{P}_{s,k}^{dup}, \mathcal{P}_{s,k}^{loss}$ and $\mathcal{P}_{s,k}^{spec}$, which respectively correspond to (s,k)-resolutions containing a duplication root, a singleton loss and only speciation roots (it turns out that these three cases are disjoint).

We show in the Appendix that this algorithm eventually terminates, and does find every solution. The essential reason that this algorithm finishes is because of the convexity of M_s, which allows avoiding circular dependencies between say $\mathcal{P}_{s,k}$ and $\mathcal{P}_{s',k'}$.

It can be shown that this algorithm takes time $O(|S| \cdot |\mathcal{P}_{r(S),1}|)$, which may be exponential. Methods for outputting solutions iteratively, each in polynomial time, seem possible, but are not immediately obvious. Notice that Zheng and Zhang's algorithms [15] can only output a subset of $\mathcal{P}_{r(S),1}$. As for NOTUNG, it takes time $O(|S|\Delta \cdot (|\mathcal{P}_{r(S),1}| + \Delta))$ to construct every optimal solution [2].

procedure COMPUTE $\mathcal{P}_{s,k}$
  if s is a leaf and $m(s) = k$ **then**
    return k singleton trees mapped to s
  Let $\mathcal{P}^{dup}_{s,k} = \emptyset, \mathcal{P}^{loss}_{s,k} = \emptyset, \mathcal{P}^{spec}_{s,k} = \emptyset$
  if $M_{s,k} = M_{s,k+1} + \delta_s$ **then**
    Compute $\mathcal{P}_{s,k+1}$
    for every forest $\mathcal{T}$ in $\mathcal{P}_{s,k+1}$, and for every pair of distinct trees $T_1, T_2 \in \mathcal{T}$ **do**
      Add to $\mathcal{P}^{dup}_{s,k}$ the (s,k)-resolution obtained by joining $r(T_1)$ and $r(T_2)$
  if $M_{s,k} = M_{s,k-1} + \lambda_s$ **then**
    Compute $\mathcal{P}_{s,k-1}$
    for every forest $\mathcal{T}$ in $\mathcal{P}_{s,k-1}$ **do**
      Add to $\mathcal{P}^{loss}_{s,k}$ the (s,k)-resolution obtained adding a singleton loss in s in $\mathcal{T}$
  if s is an internal node with children s_1, s_2 and $M_{s,k} = M_{s_1,k-m(s)} + M_{s_2,k-m(s)}$ **then**
    Compute $\mathcal{P}_{s_1,k-m(s)}$ and $\mathcal{P}_{s_2,k-m(s)}$
    for each pair $(\mathcal{T}_1, \mathcal{T}_2)$ in $\mathcal{P}_{s_1,k-m(s)} \times \mathcal{P}_{s_2,k-m(s)}$, and for every bijection $f : \mathcal{T}_1 \longrightarrow \mathcal{T}_2$ **do**
      Add to $\mathcal{P}^{spec}_{s,k}$ the (s,k)-resolution $\mathcal{T}$ obtained by joining $r(T_1)$ with $r(f(T_1))$ for every
      $T_1 \in \mathcal{T}_1$, then adding the $m(s)$ children of G mapped to s as singleton trees
  Let $\mathcal{P}_{s,k} = \mathcal{P}^{dup}_{s,k} \cup \mathcal{P}^{loss}_{s,k} \cup \mathcal{P}^{spec}_{s,k}$, and **return** $\mathcal{P}_{s,k}$
end procedure

5 Results on simulated data

We compare the running time of our algorithm to Zheng and Zhang's algorithms [15] and NOTUNG, on simulated datasets for both cases of unit and general costs. We implemented PolytomySolver and Zheng and Zhang's algorithms in python and used the latest stable version (v2.6)[2] of NOTUNG. Our implementations are available at `https://github.com/UdeM-LBIT/profileNJ`. Run times are reported for single outputs of the algorithms.

We first simulated species trees with n leaves using a birth-death process. For each species tree, gene trees of fixed size $(1.5 \times n)$ and branch support picked from a standard uniform distribution, were simulated using a simple Yule process [13]. In order to mimic a gene family history with a high number of events (duplications and losses), we labeled each leaf of the gene tree with a uniformly chosen species from the set of leaves of the species tree. Non-Binary gene trees were then obtained by contracting edges of the gene trees with support lower than a fixed threshold r (0.2, 0.4, 0.6 and 0.8).

For each species tree and each algorithm, we measured the average running time on 40 non-binary trees (10 simulated gene trees for each contraction rate). All software were run on the same computer and with the same costs for duplications and losses.

We first considered the unit cost ($\lambda = \delta = 1$), for which both PolytomySolver and Zheng and Zhang's algorithm (LZZ) are linear. Figure 4a shows the results for values of n ranging from 500 to 10000, and Figure 4b shows results for n between 10000 and 100000. As expected, the two linear algorithms exhibit very similar run time in all cases, and are significantly

[2] Notice that an improved version of NOTUNG v2.8 became available after these tests were performed.

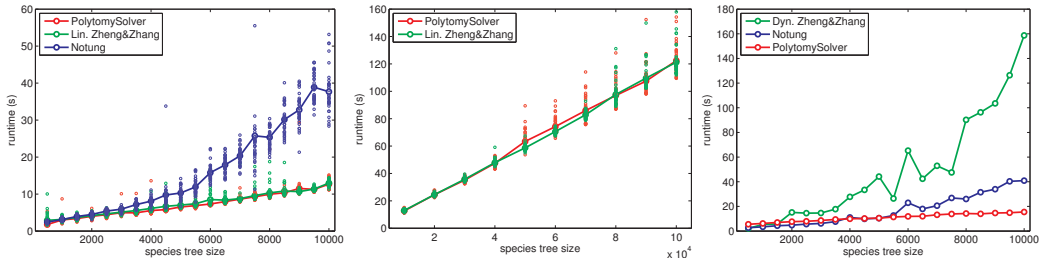


Figure 4 Running times comparisons between all algorithms for species trees of increasing size n and gene trees of size $1.5 \times n$. a Running times of PolytomySolver, LZZ (linear Zheng and Zhang's algorithm) and NOTUNG, using unit cost, for species trees of increasing size ranging from 500 to 10000. b Running times of PolytomySolver and LZZ for unit cost on larger species trees (n in the range of 10000 to 100000). c Running times of PolytomySolver, DZZ (Dynamic Zheng and Zhang's algorithm) and NOTUNG using $\delta = 3$ and $\lambda = 2$.

faster than NOTUNG, which could not be included in Figure 4b. Indeed, on those trees, NOTUNG took a considerable amount of time, and in some cases we could not get a result after many hours.

We then considered a non-unit cost, using $\delta = 3$ and $\lambda = 2$. Recall that PolytomySolver is quadratic in this case. As for the algorithm proposed by Zheng and Zhang for these costs, that we refer to by DZZ (for Dynamic Zheng and Zhang's algorithm), it is (essentially) cubic (see Table 1). Figure 4c gives the results for species trees of size ranging between 500 and 10000. As expected, PolytomySolver is faster than DZZ and NOTUNG. Surprisingly, NOTUNG turns out to be faster than DZZ, which rather expected to improve over NOTUNG as it uses the species tree compression idea. This could be due to the fact that NOTUNG is a well optimized program. Moreover, the error in NOTUNG of using m^* instead of Δ (see footnote in this Section 3), may accelerate the process, as m^* is usually much smaller than Δ.

6 A practical use of PolytomySolver

As handling species specific costs is one of the major contribution of this paper, we conclude our presentation by providing a biological example for which taking advantage of this flexibility of PolytomySolver leads to better accuracy.

We first downloaded the orthogroup of the yeast gene REG1, a regulatory subunit of type 1 protein phosphatase Glc7p, involved in negative regulation of glucose-repressible genes, from the Fungal Orthogroups Repository (http://www.broadinstitute.org/regev/orthogroups/). We then reconstructed the gene tree with PolytomySolver, using the same species tree as [14] and a unit cost for both λ and δ. Two equally parsimonious solutions with a reconciliation cost of 2 were obtained (Figures 5B, 5C).

It has been shown that the yeast *Saccharomyces cerevisiae* arose from an ancient whole-genome duplication (WGD) [4, 5, 7]. This WGD was immediately followed by a massive gene loss period, during which most of the duplicated gene copies were lost [7]. There is also evidence of lineage-specific loss of paralogous genes. In particular, *C. glabrata* and *S. castellii* appear to have lost several hundred paralogs [3, 5]. This is reflected in their total gene count, which are the lowest among the post-WGD genomes [14].

Whereas the solution shown in Figure 5C is in agreement with this WGD event, the alternative gene family history in Figure 5B places the duplication much lower in the tree, with

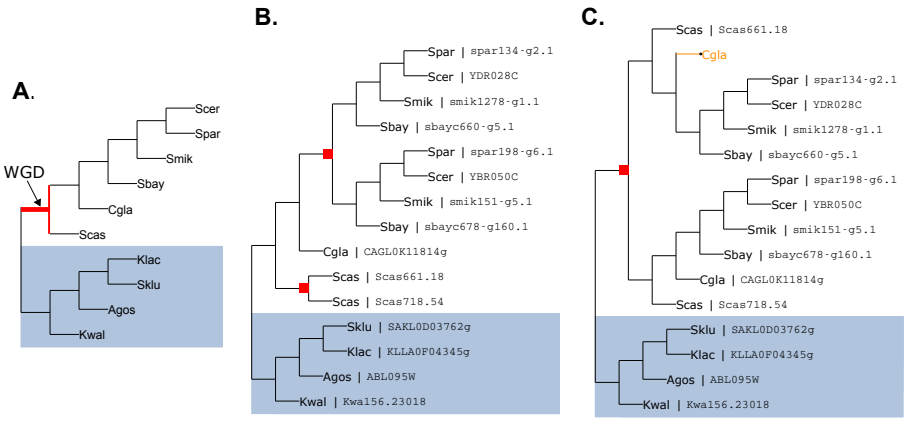


Figure 5 A. Phylogeny of ten Hemiascomycota fungi, including *S. cerevisiae (Scer), S. paradoxus (Spar), S. mikatae (Smik), S. bayanus (Sbay), C. glabrata (Cgla), S. castellii (Scas), K. waltii (Kwal), K. lactis (Klac), S. kluyveri (Sklu) and A. gossypii (Agos).* The whole-genome duplication (WGD) event in yeast is indicated. The species that did not went through the WGD are shadowed in light blue. **B.** and **C.** Two minimally resolved gene trees of the phosphatase Glc7p gene family. Duplication nodes are depicted by a red square and lost genes are shown in orange.

and additional duplication in *S. castellii* instead. By assigning to *C. glabrata* and *S. castellii* a loss cost lower than for all other species, the only solution returned by PolytomySolver is the one shown in Figure 5C. Using appropriate species dependant costs might therefore allow to filter the solution space with additional relevant information.

7 Conclusion

PolytomySolver is the most efficient algorithm to date for refining an unresolved gene tree. In contrast to previous methods, this algorithm is flexible enough to handle general reconciliation costs, allowing for instance to account for different costs over the branches of a species tree. Moreover, all topologies of optimal trees can be output by PolytomySolver. Notice that here we made no distinction between paralogous genes, which are simply referred to by their genome of origin. If we rather consider the specificity of each gene copy then, for a given topology obtained by PolytomySolver, an appropriate method shall be considered to distribute gene copies on leaves. We are presently investigating the possibility of introducing a Neighbor-Joining principle in the resolution process.

The gain in running time attained with PolytomySolver allows to perform exhaustive corrections of all trees contained in a large gene tree dataset such as Ensembl. Moreover, compared with NOTUNG, running time is independent upon the largest degree of a node, which makes the algorithm efficient enough to resolve highly unresolved trees. The next step will be to perform such a large scale gene tree dataset correction.

References

1 W.C. Chang and O. Eulenstein. Reconciling gene trees with apparent polytomies. In D.Z. Chen and D. T. Lee, editors, *Proceedings of the 12th Conference on Computing and Combinatorics (COCOON)*, volume 4112 of *Lecture Notes in Computer Science*, pages 235–244, 2006.

2 K. Chen, D. Durand, and M. Farach-Colton. Notung: Dating gene duplications using gene family trees. *Journal of Computational Biology*, 7:429–447, 2000.

3 Paul F Cliften, Robert S Fulton, Richard K Wilson, and Mark Johnston. After the duplication: gene loss and adaptation in saccharomyces genomes. *Genetics*, 172(2):863–872, 2006.

4 Fred S Dietrich, Sylvia Voegeli, Sophie Brachat, Anita Lerch, Krista Gates, Sabine Steiner, Christine Mohr, Rainer Pöhlmann, Philippe Luedi, Sangdun Choi, et al. The ashbya gossypii genome as a tool for mapping the ancient saccharomyces cerevisiae genome. *Science*, 304(5668):304–307, 2004.

5 Bernard Dujon, David Sherman, Gilles Fischer, Pascal Durrens, Serge Casaregola, Ingrid Lafontaine, Jacky De Montigny, Christian Marck, Cécile Neuvéglise, Emmanuel Talla, et al. Genome evolution in yeasts. *Nature*, 430(6995):35–44, 2004.

6 D. Durand, B.V. Haldórsson, and B. Vernot. A hybrid micro-macroevolutionary approach to gene tree reconstruction. *Journal of Computational Biology*, 13:320–335, 2006.

7 Manolis Kellis, Bruce W Birren, and Eric S Lander. Proof and evolutionary analysis of ancient genome duplication in the yeast saccharomyces cerevisiae. *Nature*, 428(6983):617–624, 2004.

8 M. Lafond, K.M. Swenson, and N. El-Mabrouk. An optimal reconciliation algorithm for gene trees with polytomies. In *LNCS*, volume 7534 of *WABI*, pages 106-122, 2012.

9 Nicolas Lartillot and Hervé Philippe. A bayesian mixture model for across-site heterogeneities in the amino-acid replacement process. *Molecular Biology and Evolution*, 21(6):1095–1109, Jun 2004. doi:10.1093/molbev/msh112.

10 Michael Lynch and John S Conery. The evolutionary demography of duplicate genes. *Journal of structural and functional genomics*, 3(1-4):35–44, 2003.

11 F. Ronquist and J.P. Huelsenbeck. MrBayes3: Bayesian phylogenetic inference under mixed models. *Bioinformatics*, 19:1572- 1574, 2003.

12 J.B. Slowinski. Molecular polytomies. *Molecular Phylogenetics and Evolution*, 19(1):114-120, 2001.

13 Mike Steel and Andy McKenzie. Properties of phylogenetic trees generated by yule-type speciation models. *Mathematical biosciences*, 170(1):91–112, 2001.

14 Ilan Wapinski, Avi Pfeffer, Nir Friedman, and Aviv Regev. Natural history and evolutionary principles of gene duplication in fungi. *Nature*, 449(7158):54–61, 2007.

15 Y. Zheng and L. Zhang. Reconciliation with non-binary gene trees revisited. In *Lecture Notes in Computer Science*, volume 8394, pages 418-432, 2014. Proceedings of RECOMB.

Genomic Scaffold Filling Revisited

Haitao Jiang[1], Chenglin Fan[2], Boting Yang[3], Farong Zhong[4], Daming Zhu[5], and Binhai Zhu[6]

1 **School of Computer Science and Technology, Shandong University, Jinan, Shandong, China**
htjiang@sdu.edu.cn

2 **Department of Computer Science, Montana State University, Bozeman, MT 59717, USA**
chenglin.fan@msu.montana.edu

3 **Department of Computer Science, University of Regina, Regina, Saskatchewan S4S 0A2, Canada**
boting.yang@uregina.ca

4 **College of Math, Physics and Information Technology, Zhejiang Normal University, Jinhua, Zhejiang, China**
zfr@zjnu.cn

5 **School of Computer Science and Technology, Shandong University, Jinan, Shandong, China**
dmzhu@sdu.edu.cn

6 **Department of Computer Science, Montana State University, Bozeman, MT 59717, USA**
bhz@montana.edu

Abstract

The genomic scaffold filling problem has attracted a lot of attention recently. The problem is on filling an incomplete sequence (scaffold) I into I', with respect to a complete reference genome G, such that the number of adjacencies between G and I' is maximized. The problem is NP-complete and APX-hard, and admits a 1.2-approximation. However, the sequence input I is not quite practical and does not fit most of the real datasets (where a scaffold is more often given as a list of contigs). In this paper, we revisit the genomic scaffold filling problem by considering this important case when, (1) a scaffold S is given, the missing genes $X = c(G) - c(S)$ can only be inserted in between the contigs, and the objective is to maximize the number of adjacencies between G and the filled S', and (2) a scaffold S is given, a subset of the missing genes $X' \subset X = c(G) - c(S)$ can only be inserted in between the contigs, and the objective is still to maximize the number of adjacencies between G and the filled S''. For problem (1), we present a simple NP-completeness proof, we then present a factor-2 greedy approximation algorithm, and finally we show that the problem is FPT when each gene appears at most d times in G. For problem (2), we prove that the problem is W[1]-hard and then we present a factor-2 FPT-approximation for the case when each gene appears at most d times in G.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases Computational biology, Approximation algorithms, FPT algorithms, NP-completeness

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.15

1 Introduction

The cost of sequencing a genome has been reduced significantly in the last decade, with the current cost being around $1k. This results in a lot of genomes being sequenced, usually not completely finished (we call them *draft* genomes). On the other hand, the cost to finish these genomes completely has not been decreased as much compared with a decade ago [9]. The result is that we are having more and more draft genomes. On the other hand, for many tools to analyze the genomic data we do need complete genomes. For instance, to compute the reversal distance between two genomes we do need two complete genomes. Hence, there is a need to turn a draft genome into a complete one.

To make the result biologically interesting, Munoz *et al.* first proposed the following *scaffold filling* problem (on multichromosomal genomes with no gene repetition) as follows [28]. Given a complete (permutation) genome R and an incomplete scaffold S, fill the missing genes in $R - S$ into S to have S' such that the genomic distance (or DCJ distance [30]) between R and S' is minimized. It was shown that this problem can be solved in polynomial time. In [22], Jiang *et al.* considered the case for singleton genomes without gene repetition (i.e., permutations), using the simplest *breakpoint* distance as the similarity measure. It was shown that this problem is solvable in polynomial time; in fact, even for the two-sided case when both the input scaffolds, being a reference to each other, are incomplete permutations.

When the genomes and scaffolds contain gene repetitions, the problem becomes harder. (That should not be considered as a surprise as even computing certain similarity measure between two complete genomes is NP-complete, for instance, with the exemplar breakpoint distance [11, 13, 2, 5, 24], exemplar adjacency number [12, 14], or the minimum common string partition [15].) The similarity measure adopted for the scaffold filling problem is the *number of common (string) adjacencies*, which can be computed in polynomial time [2, 21, 22]. In [21, 22], it was shown by Jiang *et al.* that scaffold filling to maximize the number of common string adjacencies (SF-MNSA) is NP-hard. (Formally, the problem is to fill an incomplete sequence scaffold I into I', with respect to a complete reference genome G, such that the missing letters in $G - I$ are inserted back to I and the number of common adjacencies between G and I' is maximized.) A factor-1.33 approximation was designed in [21, 22], and this bound has been improved to 1.25 [25], and to 1.20 [23]. For the corresponding two-sided case, i.e., when two scaffolds are references to each other, the problem admits a factor-1.5 approximation with the number of common adjacencies between the filled scaffolds being maximized [26]. Using the number of common adjacencies as a parameter, it was shown that this problem is also fixed-parameter tractable (FPT) – this only handles that case when G and I' are not very similar so it is only of a theoretical meaning [7].

The motivation of this paper is two-fold. Firstly, the 'scaffold' used in most of these papers is an incomplete sequence, i.e., a missing gene can be inserted anywhere in such a 'scaffold'. In practice, most of the real datasets are not in this format; in fact, a scaffold in a real dataset is usually composed of a sequence of contigs, where a contig is usually computed with mature tools like BLAST [1], hence should not be arbitrarily altered. This case was considered briefly in [28, 22], all other research on scaffold filling used an incomplete sequence as a scaffold. Secondly, take a complete reference genome G and a scaffold S, there is no guarantee that the filled scaffold S' is of the same length as that of G; in fact, sometimes we could know roughly the length of the target genome S^* (S' should be as close to S^* as possible). Then, we might only need to insert a subset of letters in $G - S$ into S (to obtain S').

The main contribution of this paper is to present some research results along these two lines. We formally call the two problems as One-sided Scaffold Filling (One-sided-SF-max), and One-sided Subset Scaffold Filling (One-sided-SF-max($\subset$)). (For the important practical case when a gene can only appear at most d times in G, we call the corresponding problems One-sided-SF-max(d) and One-sided-SF-max($\subset, d$) respectively.) The objective function in both cases are to maximize the number of common adjacencies between the reference and the filled scaffold. For One-sided-SF-max, we present a simple reduction from the Hamiltonian Path problem hence showing it to be NP-hard, we then present a factor-2 approximation. Then we show that One-sided-SF-max(d) is FPT. For One-sided-SF-max($\subset$), we prove a stronger negative result by showing that, parameterized by the number of missing genes inserted, the problem is W[1]-hard. We then present a factor-2 FPT-approximation for the special case One-sided-SF-max($\subset, d$). As far as we know, this is the first W[1]-hardness result on the research of scaffold filling.

The paper is organized as follows. In Section 2, we give the preliminaries. In Section 3, we present the approximation results for One-sided-SF-max. In Section 4, we present the FPT algorithm for One-sided-SF-max(d). In Section 5, we present the results for One-sided-SF-max($\subset$). We conclude the paper in Section 6.

2 Preliminaries

Throughout this paper we focus only on singleton genomes (i.e., each is a sequence). But the results can be easily generalized to multichromosomal or circular genomes, with minor changes.

At first, we review some necessary definitions, which are also defined in [22, 31]. We assume that all genes and genomes are unsigned, and it is straightforward to generalize the result to signed genomes. Given a gene set Σ, a string P is called *permutation* if each element in Σ appears exactly once in P. We use *c(P)* to denote the set of elements in permutation P. A string A is called *sequence* if some genes appear more than once in A, and $c(A)$ denotes genes of A, which is a multi-set of elements in Σ. For example, $\Sigma = \{a, b, c, d\}$, $A = abcdacd$, $c(A) = \{a, a, b, c, c, d, d\}$. A *sequence scaffold* is an incomplete sequence, typically obtained by some sequencing and assembling process. A substring with m genes (in a sequence) is called an *m-substring*, and a 2-substring is also called a *pair*; as the genes are unsigned, the relative order of the two genes of a pair does not matter, i.e., the pair xy is equal to the pair yx. Given an incomplete sequence (or sequence scaffold) $A{=}a_1a_2a_3\cdots a_n$, let $P_A = \{a_1a_2, a_2a_3, \ldots, a_{n-1}a_n\}$ be the set of pairs in A.

▶ **Definition 1.** Given two sequence scaffolds $A{=}a_1a_2\cdots a_n$ and $B{=}b_1b_2\cdots b_m$, if $a_ia_{i+1} = b_jb_{j+1}$ (or $a_ia_{i+1}{=}b_{j+1}b_j$), where $a_ia_{i+1} \in P_A$ and $b_jb_{j+1} \in P_B$, we say that a_ia_{i+1} and b_jb_{j+1} are matched to each other. In a maximum matching of pairs in P_A and P_B, a matched pair is called an **adjacency**, and an unmatched pair is called a **breakpoint** in A and B respectively.

It follows from the definition that sequence scaffolds A and B contain the same set of adjacencies but distinct breakpoints. The maximum matched pairs in B (or equally, in A) form the *(common) adjacency set* between A and B, denoted as $a(A, B)$. We use $b_A(A, B)$ and $b_B(A, B)$ to denote the set of breakpoints in A and B respectively. We illustrate the above definitions in Fig. 1.

For a sequence A and a multi-set of elements X, let $A + X$ be the set of all possible resulting sequences after filling all the elements in X into A. We define a contig as a string

$$
\begin{aligned}
sequence\ scaffold\ A &= \langle c\ b\ c\ e\ d\ a\ b\ a\ \rangle \\
sequence\ scaffold\ B &= \langle a\ b\ a\ b\ d\ c\rangle \\
P_A &= \{cb, bc, ce, ed, da, ab, ba\} \\
P_B &= \{ab, ba, ab, bd, dc\} \\
matched\ \ pairs &: (ab \leftrightarrow ba), (ba \leftrightarrow ab) \\
a(A,B) &= \{ab, ba\} \\
b_A(A,B) &= \{cb, bc, ce, ed, da\} \\
b_B(A,B) &= \{ab, bd, dc\}
\end{aligned}
$$

Figure 1 An example for adjacency and breakpoint definitions.

over a gene set Σ whose contents should not be altered. A *scaffold* S is simply a sequence of contigs $\langle C_1, ..., C_m\rangle$. We define $c(S) = c(C_1) \cup \cdots \cup c(C_m)$. Now, we define the problems on scaffolds formally.

▶ **Definition 2.** One-Sided-SF-max.
Input: a complete genome G and a scaffold $S = \langle C_1, C_2, ..., C_m\rangle$ where G and the contig C_i's are over a gene set Σ, a multiset $X = c(G) - c(S) \neq \emptyset$.
Question: Find $S^* \in S + X$ such that $|a(S^*, G)|$ is maximized.

One-Sided-SF-max($\subset$) is exactly the same as One-Sided-SF-max except that only a subset $X' \subset X$ need to be inserted into S. When a gene can appear at most d times in G, the two versions of problems are abbreviated as One-Sided-SF-max(d) and One-Sided-SF-max($\subset, d$) respectively.

We first present a simple reduction from Hamiltonian Path to One-Sided-SF-max.

▶ **Theorem 3.** *The decision version of One-Sided-SF-*max *is NP-complete.*

Proof. It is obvious that the decision version of One-Sided-SF-max is in NP, so we just focus on the reduction from Hamiltonian Path. Given a connected graph $H = (V, E)$, with $V = \{v_1, v_2, \cdots, v_n\}$ and $e_i = (v_{i,1}, v_{i,2})$, for $e_i \in E$, let $e'_i = v_{i,1}v_{i,2}$, for $i = 1..m$. Let $deg(v)$ be the degree of vertex v (assuming $deg(v) > 1$ for all v). G and S are constructed as follows.

$$G = \#e'_1\#e'_2\#\cdots\#e'_m\# \circ \#_2\#_3\#_1^n,$$

and

$$S = \langle C_1, C_2\rangle,$$

with $C_1 = \langle \#_2 v_1^{deg(v_1)-1}\#_1 \cdots v_n^{deg(v_n)-1}\#_1\#\rangle$ and $C_2 = \langle \#^m\#_3\rangle$. Here $\circ$ is a connector and $X = c(G) - c(S) = V$. As there are only three places to insert elements in X back to S, moreover, the only possible adjacencies are between two vertices forming an edge in H and between a vertex and a $\#$, it is obvious that to maximize the number of adjacencies we need to insert the sequence of vertices forming a Hamiltonian Path in between C_1, C_2.

We make the following claim: H has a Hamiltonian path iff n missing genes can be inserted into S to obtain $n+1$ adjacencies. We only show the "only if" part here as the other direction is trivial. If n missing genes can be inserted into S to obtain $n+1$ adjacencies, say they are inserted between C_1 and C_2 as $v'_1v'_2\cdots v'_n$ (where $v'_j = v_i$), then $n-1$ adjacencies must be $v'_jv'_{j+1}$ and the other two are $\#v'_1$ and $v'_n\#$. Then each $v'_jv'_{j+1}$ corresponds to an

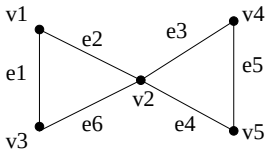


Figure 2 A simple graph H for the reduction.

edge in H and $v'_1v'_2\cdots v'_n$ corresponds to a Hamiltonian path in H. It is obvious that this reduction take $O(n^2)$ time. ◀

We show a simple example for the reduction. The graph H is given in Fig. 2. We have

$$G = \#v_1v_3\#v_1v_2\#v_2v_4\#v_2v_5\#v_4v_5\#v_2v_3\#\#_2\#_3\#_1\#_1\#_1\#_1\#_1,$$

$$S = \langle \boxed{\#_2v_1\#_1v_2v_2v_2\#_1v_3\#_1v_4\#_1v_5\#_1\#} \rangle, \langle \boxed{\#\#\#\#\#\#\#_3} \rangle.$$

After inserting genes in V into S, we obtain

$$S^* = \boxed{\#_2v_1\#_1v_2v_2v_2\#_1v_3\#_1v_4\#_1v_5\#_1\#} v_1v_3v_2v_4v_5 \boxed{\#\#\#\#\#\#\#_3}.$$

It is easy to verify that we have $n+1 = 6$ common adjacencies between G and S^*: $\#v_1$, v_1v_3, v_3v_2, v_2v_4, v_4v_5 and $v_5\#$.

We note that the reduction for the unbounded case SF-MNSA (from X3C in [21, 22]) in fact also works for One-Sided-SF-max – just making each letter in I a contig. (Of course, this would make the contigs too artificial.) But it is obvious that the above proof is simpler and more straightforward. We next present an approximation algorithm for One-sided-SF-max.

3 An Approximation Algorithm for One-Sided-SF-max

Before presenting our algorithm, we make the following definitions.

Let α_i, β_i be the first and last letter of $C_i, i = 1..m$, respectively. Then $\langle \beta_i, \alpha_{i+1} \rangle$ constitutes a region where missing genes can inserted between β_i and α_{i+1}, for $i = 1..m$. Here, we also have two open regions on the two ends of S. We denote them as $\langle -\infty, \alpha_1 \rangle$ and $\langle \beta_m, +\infty \rangle$ respectively.

We define a type-1 substring s of length $\ell \geq 1$, over X, as one which can be inserted in $\langle \beta_i, \alpha_{i+1} \rangle$, for $1 \leq i \leq m-1$, to generate $\ell + 1$ new common adjacencies. We call $\langle \beta_i, \alpha_{i+1} \rangle$ a type-1 *slot* for s. (Throughout this paper, once a type-1 slot is inserted with a corresponding substring we do not allow the insertion of any other letter.) It is easy to see that we could have at most $m-1$ type-1 slots.

Then, we define a type-2 substring s of length $\ell \geq 1$, over X, as one which can be inserted in $\langle \beta_i, \alpha_{i+1} \rangle$, for $0 \leq i \leq m$, to generate ℓ common adjacencies. (We write $\beta_0 = -\infty$ and $\alpha_{m+1} = +\infty$. Clearly the two open slots can be type-2 or type-3.) Note that in this case, in $\langle \beta_i, \alpha_{i+1} \rangle$, we could have two type-2 slots, i.e., right after β_i (written as $\beta_i\circ$) or right before α_{i+1} (written as $\circ\alpha_{i+1}$). By definition, for a fixed $\langle \beta_i, \alpha_{i+1} \rangle$, it cannot be type-1 and type-2 at the same time. It is easy to see that we could have at most $2(m-1)+2 = 2m$ type-2 slots.

Note that if $\beta_i\alpha_{i+1}$ is already a common adjacency with respect to G, then it is possible that s is inserted in the slot to generate $|s|+1$ common adjacencies (while destroying the common adjacency $\beta_i\alpha_{i+1}$). In this case, s really increases the total number of common adjacencies by $|s|$. Hence, s is considered as type-2. For convenience, we simply say that

in this case s generates $|s|$ new common adjacencies. In fact, with a simple example we could show that such an existing adjacency in a slot must be destroyed to obtain an optimal solution. Example: $G = \langle 1, 1, 5, 4, 3, 5, 3, 7, 7\rangle$, $S = \langle \boxed{1,7,3,5}, \boxed{3,1,5,7}\rangle$, the missing gene 4 must be inserted between $\boxed{1,7,3,5}, \boxed{3,1,5,7}$ to obtain the optimal solution.

Finally, we define a type-3 substring s of length $\ell \geq 1$, over X, as one which can be inserted in the slot $\langle \beta_i, \alpha_{i+1}\rangle$, for some i, to generate $\ell - 1$ common adjacencies. Note that a type-3 substring can only form adjacencies internally, hence it does not matter where we insert s – provided that it does not destroy any existing adjacencies.

We show an example as follows:

$G = \langle 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6\rangle$,

$S = \langle \boxed{1,5}, \boxed{3,6}, \boxed{2,4}\rangle$.

We have $\alpha_1 = 1, \beta_1 = 5$, $\alpha_2 = 3, \beta_2 = 6$, $\alpha_3 = 2, \beta_3 = 4$. Then, $X = \{1, 2, 3, 4, 5, 6\}$ are missing from S. One of the optimal solution is

$S' = \langle 1, 2, \boxed{1,5}, 6, \boxed{3,6}, 5, 4, 3, \boxed{2,4}\rangle$.

In this case, $\langle 5, 4, 3\rangle$ is type-1, 6 and $\langle 1, 2\rangle$ are type-2.

We comment that in general a type-j substring, $j = 1, 2, 3$, does not have to be a substring of G. If a type-j substring is composed of i letters, we call it an i-type-j substring.

Let the number of common adjacencies between G and S be k_0, and the number of newly increased common adjacencies be k_1 (after all genes in X have been inserted into S). To approximate $k_0 + k_1$, it suffices to approximate k_1. This is because if we have an approximation solution A_1 for k_1, i.e., $|A_1| \geq k_1/\rho$, then $k_0 + |A_1| \geq (k_0 + k_1)/\rho$ (for $\rho > 1$). From now on, we will only discuss the approximation for the newly increased common adjacencies.

Our **Algorithm 1** is a simple greedy one:

1. Scan through all slots, if an 1-string (i.e., a letter) x or a 2-string xy in X can be inserted in such a slot t to obtain two adjacencies or three adjacencies, insert x or xy into t, lock t. Update $X \leftarrow X - \{x\}$ or $X \leftarrow X - \{x, y\}$ accordingly.
2. For all the remaining (type-2) slots, if a letter $x \in X$ could be inserted to obtain one adjacency, then insert x into the slot and update the the slot as follows. If x is inserted at the slot $y\circ$ (resp. $\circ y$) then update the slot as $x\circ$ (resp. $\circ x$).
3. For all the letters in X after Step 1 (including those already inserted at Step 2), compute a multigraph Q with the vertices being these letters in X (after Step 1), and if xy is a potential adjacency in G (ignoring those already matched with the ones computed at Step 1 and 2), then there is an edge between all $x \in X$ and all $y \in X$. Compute a maximum matching M in Q. For all the pairs xy in M with one end x being a letter inserted at Step 3, insert y before or after x accordingly. For the remaining pairs in M, insert them arbitrarily in any unlocked slot in S, provided no existing adjacency is destroyed.
4. Insert the remaining letters in X arbitrarily in any unlocked slot in S, provided no existing adjacency is destroyed.

Let b_{ij} denote the number of j-type-i substrings in some optimal solution. Then the optimal solution value

$$Opt = \sum_{j=1..p} (j+1)b_{1j} + \sum_{j=1..q} jb_{2j} + \sum_{j=2..r} (j-1)b_{3j},$$

for some p, q, r. Let b'_{ij} denote the number of j-type-i substrings in the approximation solution. We show the properties of the greedy algorithm as follows.

▶ **Lemma 4.** *After Step 1,* $2b'_{11} + 3b'_{12} \geq \frac{1}{2}(2b_{11} + 3b_{12})$.

Proof. By the greedy choice, we have $b'_{11} + b'_{12} \geq b_{11} + b_{12}$. Then,

$$\begin{aligned} 2b'_{11} + 3b'_{12} &\geq 2b'_{11} + 2b'_{12} \\ &\geq 2b_{11} + 2b_{12} \\ &= \frac{1}{2}(4b_{11} + 4b_{12}) \\ &\geq \frac{1}{2}(2b_{11} + 3b_{12}). \end{aligned}$$

◀

▶ **Lemma 5.** *After Step 2,* $b'_{21} \geq b_{21}$.

Proof. If a slot t could be either inserted with an i-type-1 substring s_i for $i = 1, 2$, then a 1-type-2 substring (letter) x could not be inserted at the slot t in an optimal solution. The reason is as follows. (1) Suppose that t can be inserted with an 1-type-1 substring s_1. If t in the optimal solution is inserted with x to generate one adjacency, then we could swap x with s_1 to generate at least two adjacencies. This contradicts with the optimality of the assumed optimal solution. (2) Suppose that t can be inserted with an 2-type-1 substring s_2. If t in the optimal solution is inserted with x to generate one adjacency, then, again, we could swap x with s_2 to generate at least three adjacencies. This implies that there is an optimal solution where all 2-type-1 substrings are always inserted before any 1-type-2 substring is processed.

Then following the greedy choice at Step 2, we have $b'_{21} \geq b_{21}$. ◀

Hence, we could have the following theorem.

▶ **Theorem 6.** *One-Sided-SF-*max *can be approximated within a factor of 2.*

Proof. By definition, the optimal solution value OPT satisfies

$$Opt = \sum_{j=1..p} (j+1)b_{1j} + \sum_{j=1..q} jb_{2j} + \sum_{j=2..r} (j-1)b_{3j},$$

for some p, q, r. At Step 3, the size of the maximum matching, $|M|$, satisfies

$$|M| \geq \frac{1}{2}\left(\sum_{j=3..p} (j+1)b_{1j} + \sum_{j=2..q} jb_{2j} + \sum_{j=2..r} (j-1)b_{3j}\right).$$

The right-hand side of the above inequality represents the optimal internal adjacencies among the corresponding type-1, type-2, and type-3 substrings in the optimal solution. The approximation solution value, *App*, satisfies

$$\begin{aligned} App &= (2b'_{11} + 3b'_{12}) + b'_{21} + |M| \\ &\geq \frac{1}{2}(2b_{11} + 3b_{12}) + b'_{21} + |M| \quad \text{(by Lemma 4)} \\ &\geq \frac{1}{2}(2b_{11} + 3b_{12}) + b_{21} + |M| \quad \text{(by Lemma 5)} \\ &\geq \frac{1}{2}Opt. \end{aligned}$$

◀

4 An FPT Algorithm for One-Sided-SF-max(d)

In this section, we present an FPT algorithm for One-Sided-SF-max(d), parameterized by the optimal number of common adjacencies k. Whether One-Sided-SF-max is FPT is still open, but One-Sided-SF-max(d) represents the important practical version where each gene appears in a genome at most d times. We first review *Fixed-Parameter Tractable* (FPT) algorithms.

4.1 Definition of FPT Algorithms

Let Σ be the alphabet, and $Q \subseteq \Sigma^*$ be a classic decision problem. A parameterized problem is a pair (Q, κ) where $\kappa : \Sigma^* \to \mathbb{N}$ is a polynomial computable function. An instance of (Q, κ) is a pair $(x, \kappa(x))$ consisting of a string $x \in \Sigma^*$ and an integer $\kappa(x)$.

▶ **Definition 7.** Let (Q, κ) be a parameterized problem. We say (Q, κ) is Fixed-Parameter Tractable (FPT) if for each instance $(x, \kappa(x))$, there is an algorithm A which decides whether $x \in Q$ in $f(\kappa(x)) \cdot |x|^c$ time, where f is an arbitrary computable function and c is a constant.

As a convention now, we write $O(f(\kappa(x))n^c) = O^*(f(k))$. FPT algorithms are efficient tools for handling some NP-complete problems, especially when $k = \kappa(x)$ is small in some practical datasets [16, 18, 29].

4.2 The FPT Algorithm

We now present an FPT algorithm for One-Sided-SF-max(d), parameterized by the optimal number of common adjacencies k. (Here k includes the existing number of common adjacencies between S and G, though it is obvious that our algorithm also works by looking at newly created common adjacencies.) As the running time of the algorithm is high and the result is mostly for theoretical purpose.

Our idea is as follows. We use the color-coding method to find a potential ℓ-type-i substring for $i = 1, 2$. Then we use the property that each gene appears at most d times to search for a slot to put this string in a right slot. After this process are repeated for all potential type-1 and type-2 substrings, type-3 substrings can then be inserted arbitrarily, as long as they do not destroy the existing adjacencies.

Note that a 1-type-3 substring cannot contribute any common adjacency with respect to G, so it is *useless*. All other inserted letters are *useful*. We first show the following lemma regarding the number of useful letters in an optimal solution.

▶ **Lemma 8.** *Let $X^* \subseteq X$ be the set of genes in X that contribute in generating some new common adjacencies. If the optimal number of common adjacencies between G and S^* is k, then $|X^*| \leq 2k$.*

Proof. From the previous discussions, a ℓ-type-1 substring creates $\ell + 1$ common adjacencies, a ℓ-type-2 substring creates ℓ common adjacencies, and a ℓ-type-3 substring creates $\ell - 1$ common adjacencies. Hence, in the worst case, the k common adjacencies are created by $2k$ type-3 substrings, each of length 2 (creating one common adjacency). In this case, these genes form the set of optimal active genes X^*, with $|X^*| \leq 2k$. ◀

We then make use of the color-coding method [3, 4], summarized as the following lemma. For a positive integer n, let $[n] = \{1, 2, ..., n\}$.

▶ **Lemma 9** ([3, 4]). *Let $1 \leq \ell \leq k$. For every n, ℓ there is a family $\Delta_{n,\ell}$ of polynomial time computable functions from $[n]$ to $[k]$ such that for every ℓ-element subset Y of $[n]$, there is an $h \in \Delta_{n,\ell}$ such that h is injective on Y. Moreover, $\Delta_{n,\ell}$ can be computed in time $2^{O(k)} \cdot n^{O(1)}$.*

The following lemma is similar to that for solving the k-path problem using color-coding [3, 4].

▶ **Lemma 10.** *Given a fixed slot, a p-type-j substring, $j = 1, 2$, can be computed in FPT time.*

Proof. A p-type-j substring is formed by the 2-substrings (or, at most $n-1$ possible adjacencies) in G. We use the color-coding technique. For the ease of description, we focus on $j = 1$. We give each 2-substring in G one of the $p+1$ random colors. A p-type-1 substring for a given slot is determined by $p+1$ 2-substrings in G. The probability that we could find such a colorful p-type-1 substring is at least

$$\frac{(p+1)!}{(p+1)^{p+1}} = \frac{\sqrt{2\pi(p+1)}}{e^{p+1}} > (\frac{1}{e})^{p+1},$$

where $p! \sim \sqrt{2\pi p}(\frac{p}{e})^p$, following Stirling's formula. To guarantee that we could obtain a valid solution, we simply run this algorithm e^{p+1} times. This process can be derandomized with standard techniques [16, 18, 4]. The total running time of this algorithm is then bounded by $O^*(e^{p+1})$. For constructing the corresponding p-type-2 and p-type-3 substrings (over the unused/unmatched 2-substrings in G), the running times are $O^*(e^p)$ and $O^*(e^{p-1})$ respectively. Note that type-3 substrings are not relevant to any specific slot. ◀

▶ **Theorem 11.** *One-Sided-SF-*max*(d) is FPT.*

Proof. The general idea is a combination of bounded-degree search and color-coding. Following Lemma 9 and 10, the algorithm generates a proper p-type-j substring s, where $p \leq k-1, j = 1$ or $p \leq k, j = 2$, for a potential slot $\langle \beta_i, \alpha_{i+1} \rangle$. As β_i and α_{i+1} can each appear d times, we could have $2d$ possible slots to put s. We then delete the letters in s from X and repeat the process until no type-1 or type-2 substring can be inserted in S. If the number of common adjacencies is at least k, we stop and insert the remaining letters in X arbitrarily, not to destroy any existing adjacency. If the number of common adjacencies is still less than k, we use Lemma 10 to generate some p-type-3 substring and insert it arbitrarily into S (not to destroy any existing adjacency). By Lemma 8, the search stops when a total of at most $2k$ useful letters have been inserted. (The remaining letters can be inserted arbitrarily, provided that they do not destroy any existing common adjacency). We can then check and report a solution with at least k common adjacencies, or report that such a solution does not exist.

The total running time of this algorithm is

$$O^*((2d \cdot e^k)^k) = O^*(2^k d^k e^{2k}).$$

Hence we have the theorem. ◀

In the next section, we discuss the One-sided Subset Scaffold Filling (One-sided-SF-max($\subset$)) problem.

5 Results for One-Sided-SF-max($\subset$)

In this section, we present some results for One-Sided-SF-max($\subset$). We prove that if the parameter is the number of genes inserted, then the problem is W[1]-hard. This implies that the problem cannot be solved with an FPT algorithm, unless FPT=W[1] [16, 18, 29]. We then present a simple FPT-approximation for the problem, with a factor of 2, for One-Sided-SF-max($\subset, d$).

5.1 W[1]-Hardness Result

The main theorem is stated as follows.

▶ **Theorem 12.** *One-Sided-SF-*max($\subset$) *parameterized by the number of genes inserted is W[1]-hard.*

Proof. Throughout this proof, assume that $k \leq (n-1)/2$. We show that Independent Set can be reduced to One-Sided-SF-max($\subset$) via a linear FPT reduction. Given a graph $Q = (V, E)$, if the maximum vertex degree is Δ, then for each vertex $u_i \in V$ with degree $deg(u_i) < \Delta$, we create $\Delta - deg(u_i)$ new nodes and connect them only to u_i. In the resulting graph $Q' = (V', E')$, all the original vertices in V have degree Δ. It can be easily seen that Q has an independent set of size k iff k vertices in Q' can be selected to cover exactly $k\Delta$ edges. This part of the proof is adapted from [20].

Now we arrange the graph $Q' = (V', E')$ as a genome G as follows. WLOG, still assume that $|V'| = n, |E'| = m$ throughout this proof. For each $v_i \in V'$, construct E_i as the list of edges incident to v_i (ordered by their indices). Then we use separators #'s and $\#_j$, for $j = 1..5$. The set of genes are $\{e_i | i = 1..m\} \cup \{\#_j | j = 1..5\} \cup \{\#\}$. Finally we arrange G as follows.

$$G = \#^{m+1} \circ \#_1\#_2\#_3\#_4 \circ \#_4\#_3\#_2\#_1 \circ \#_5 E_1 \#_5 E_2 \#_5 \cdots \#_5 E_n \#_5.$$

Note that $\circ$ is used as a connector, each e_i $(i = 1..m)$, $\#_j$ $(j = 1..4)$ appears twice in G, # appears $m+1$ times and $\#_5$ appears $n+1$ times in G. S is constructed such that it is composed of exactly $k+1$ contigs $C_1, ..., C_{k+1}$, each C_i starts and ends with $\#_5$. For C_1, between the two $\#_5$'s, we arrange all the genes #'s and e_i's such that $C_1 = \#_5\#e_1\#e_2\#\cdots\#e_m\#\#_5$. We construct $C_2 = \#_5\#_3\#_1\#_4\#_2 \circ \#_5^{n-2k-1} \circ \#_2\#_4\#_1\#_3\#_5$. The remaining contigs are constructed as $C_i = \#_5\#_5$ for $i = 3, ..., k+1$.

It is clear that in S we have missed a copy of e_i for each $i = 1..m$. Due to the construction of G, e_i cannot form any common adjacency with # or $\#_j$ for $j = 1..4$, the only possible common adjacencies are from e_i and e_ℓ's (i.e., in some sequences of E_p's, each of length Δ) and between e_i and $\#_5$'s. To maximize the common adjacencies obtained, these missing genes can only be inserted in k slots, after C_i and before C_{i+1} for $i = 1..k$. Then, it is safe for us to claim, with some easy details omitted, that Q has an independent set of size k iff $k\Delta$ missing genes can be inserted into the k slots in S to obtain a maximum of $k(\Delta+1)$ adjacencies with respect to the reference genome G. This is obviously an FPT-reduction. ◀

With the above W[1]-hardness result, it is easy to obtain the following corollary (part of it is similar to the corollary in [27]).

▶ **Corollary 13.** *The optimization version of One-Sided-SF-*max($\subset$) *does not admit an EPTAS (resp. FPTAS) unless FPT=W[1].*

Proof. Assume that there is an EPTAS (resp. FPTAS) which runs in time $O((\frac{1}{\epsilon})^{O(\frac{1}{\epsilon})}n^c)$ (resp. $O((\frac{1}{\epsilon})^{c_1}n^{c_2})$), for some constant c (resp. c_1 and c_2); moreover, it achieves an approximation factor of $1+\epsilon$, for any $\epsilon > 0$. Then, if k^* is the optimal solution value and APP is the approximation solution value, we have

$$APP \geq \frac{k^*}{1+\epsilon}.$$

Setting $\epsilon = \frac{1}{2k^*-1}$, we have $APP \geq \frac{k^*}{1+\epsilon} = k^* - \frac{1}{2}$, which further implies $APP = k^*$. In this case, the running time of the algorithm becomes $O((k^*)^{O(k^*)}n^c)$ (resp. $O((k^*)^{O(c_1)}n^{c_2})$); i.e., the problem would admit an FPT algorithm. A contradiction to Theorem 12, unless FPT=W[1]. ◀

5.2 FPT-Approximation for One-Sided-SF-max($\subset$,d)

For W[1]-hard problems, a natural way to handle them is to use FPT-approximations. Here we briefly review the Fixed-Parameter Tractable Approximation Algorithm (FPT-approximation for short), which was first proposed in 2006 [10, 17, 8] (but the development has been slow.)

▶ **Definition 14.** A Fixed-Parameter Tractable ρ-approximation for a minimization (resp. maximization) parameterized problem (Q, κ) is an FPT algorithm which, given any instance $(x, k) \in (Q, \kappa)$, returns a solution of cost at most $\rho(k) \cdot k$ (resp. at least $k/\rho(k)$) if a solution of cost at most (resp. at least) k exists.

Our FPT-approximation algorithm for One-Sided-SF-max$(\subset, d)$, parameterized by the number of inserted genes, is as follows.

1. As in Theorem 11, use bounded-degree search and color-coding to insert ℓ $(0 \leq \ell \leq k)$ type-1 and type-2 substrings into the ℓ slots, which can be done in FPT time.
2. If these ℓ substrings have a total length at least k, then the problem can be solved optimally in FPT time.
3. If these ℓ substrings have a total length k_1 with $k_1 < k$, then we insert enough type-3 substrings (of a total length $k - k_1$) as follows.
4. We use a maximum matching method to insert $k - k_1$ letters. For all the remaining genes to be inserted into G, form a graph D such that there is an edge connecting two such genes if they could potentially form a common adjacency with respect to G. Then simply compute a maximum matching in D and insert all the pairs in the matching arbitrarily into D (provided that they do not destroy any existing common adjacency).

▶ **Theorem 15.** *One-Sided-SF-*max$(\subset, d)$ *parameterized by the number of genes inserted admits a factor-2 FPT-approximation.*

Proof. The analysis of the first two steps of the FPT algorithm is the same as in Theorem 11, hence omitted.

Let k_1 letters inserted at Step 1 generate k_1^* common adjacencies. The $k - k_1$ genes forming type-3 substrings could generate at most $k_2 \leq k - k_1 - 1$ common adjacencies. By the maximum matching algorithm at step 4, we could generate at least $k_2/2$ common adjacencies. (For any connected component in D, if it contains a path of length $k_3 \leq k_2$ then the maximum matching algorithm could return at least $k_3/2$ common adjacencies.) Then

$$OPT = k_1^* + k_2,$$

and

$$APP \geq k_1^* + k_2/2 \geq \frac{OPT}{2}.$$

The whole algorithm obviously takes FPT time. ◀

6 Concluding Remarks

In this paper, we revisit the genomic scaffold filling problem by considering each scaffold as a sequence of contigs (instead of as an incomplete sequence as in most of the previous research). We obtain a list of algorithmic results, some of which could eventually lead to the practical processing of genomic datasets. However, as in [7], the parameter k (i.e., number of common adjacencies) in reality should be relatively large, so the FPT algorithms we obtained here are only theoretically meaningful. Further research is needed along this line. On the other hand, theoretically, it is interesting to decide whether One-Sided-SF-max is FPT and whether One-Sided-SF-max($\subset$) admits an FPT-approximation.

Acknowledgments This research is partially supported by the Open Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University. We also thank anonymous reviewers for several useful comments.

References

1 S. Altschul, W. Gish, W. Miller, E. Myers and D. Lipman. Basic local alignment search tool. *J. Molecular Biology*, 215(3):403-410, 1990.

2 S. Angibaud, G. Fertin, I. Rusu, A. Thevenin and S. Vialette. On the approximability of comparing genomes with duplicates. *J. Graph Algorithms and Applications*, 13(1):19-53, 2009.

3 N. Alon, R. Yuster and U. Zwick. Color-coding. *J. ACM*, 42(4):844-856, 1995.

4 N. Alon, R. Yuster and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209-218, 1997.

5 G. Blin, G. Fertin, F. Sikora and S. Vialette. The exemplar breakpoint distance for non-trivial genomes cannot be approximated. *Proc. 3nd Workshop on Algorithm and Computation (WALCOM'2009)*, LNCS 5431, pp. 357-368, 2009.

6 H. Bodlaender, R. Downey, M. Fellows and D. Hermelin. On problems without polynomial kernels. *J. Comput. Syst. Sci.*, 75(8):423-434, 2009.

7 L. Bulteau, A.P. Carrieri and R. Dondi. Fixed-parameter algorithms for scaffold filling. *Theoretical Computer Science*, 568: 72–83, 2015.

8 L. Cai and X. Huang. Fixed-parameter approximation: conceptual framework and approximability results. *Algorithmica*, 57(2):398-412, 2010.

9 P.S. Chain, D.V. Grafham, R.S. Fulton, *et al.* Genome project standards in a new era of sequencing. *Science*, 326:236-237, 2009.

10 Y. Chen, M. Grohe and M. Grueber. On parameterized approximability. *Proc. 2nd Intl. Workshop on Parameterized and Exact Computation (IWPEC'06)*, LNCS 4169, pp. 109-120, 2006.

11 Z. Chen, B. Fu and B. Zhu. The approximability of the exemplar breakpoint distance problem. *Proc. 2nd Intl. Conf. on Algorithmic Aspects in Information and Management (AAIM'06)*, LNCS 4041, pp. 291-302, 2006.

12 Z. Chen, B. Fu, B. Yang, J. Xu, Z. Zhao, and B. Zhu. Non-breaking similarity of genomes with gene repetitions. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM'07)*, LNCS 4580, pp. 119–130, 2007.

13 Z. Chen, B. Fu, R. Fowler and B. Zhu. On the inapproximability of the exemplar conserved interval distance problem of genomes. *J. Combinatorial Optimization*, **15**(2):201-221, 2008.

14 Z. Chen, B. Fu, R. Goebel, G. Lin, W. Tong, J. Xu, B. Yang, Z. Zhao and B. Zhu. On the approximability of the exemplar adjacency number problem of genomes with gene repetitions. *Theoretical Computer Science*, **550**:59-65, 2014.

15 G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA'02)*, pp. 667-676, 2002.

16 R. Downey and M. Fellows. *Parameterized Complexity*, Springer-Verlag. 1999.

17 R. Downey, M. Fellows, C. McCartin and F. Rosamond. Parameterized approximation of dominating set problems. *Info. Process. Lett.*, 109(1): 68–70, 2008.

18 J. Flum and M. Grohe. *Parameterized Complexity Theory*, Springer-Verlag. 2006.

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

20 J. Guo, R. Niedermeier and S. Wernicke. Parameterized complexity of vertex cover variants. *Theory Comput. Syst.*, 41(3):501-520. 2007.

21 H. Jiang, F. Zhong and B. Zhu. Filling scaffolds with gene repetitions: maximizing the number of adjacencies. *Proc. 22nd Annual Combinatorial Pattern Matching Symposium (CPM'11)*, LNCS 6661, pp. 55-64, Palermo, Italy, June 27-29, 2011.

22 H. Jiang, C. Zheng, D. Sankoff, and B. Zhu. Scaffold filling under the breakpoint and related distances. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(4):1220-1229, July/August, 2012.

23 H. Jiang, J. Ma, J. Luan and D. Zhu. *Approximation and nonapproximability for the one-sided scaffold filling problem. Proc. 21st Intl. Ann. Comput. and Combinatorics (COCOON'15),* LNCS 9198, pp. 251-263, 2015,

24 M. Jiang. The zero exemplar distance problem. *Proc. of the 2010 International RECOMB-CG Workshop (RECOMB-CG'10)*, LNBI 6398, pp. 74-82, 2010.

25 N. Liu, H. Jiang, D. Zhu, and B. Zhu. An improved approximation algorithm for scaffold filling to maximize the common adjacencies. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 10(4):905-913, July/August, 2013.

26 N. Liu, D. Zhu, H. Jiang and B. Zhu. A 1.5-approximation algorithm for two-sided scaffold filling. *Algorithmica*, 74(1):91-116, 2016.

27 D. Marx. Parameterized complexity and approximation algorithms. *Computer Journal*, 51(1):60-78, 2008.

28 A. Muñoz, C. Zheng, Q. Zhu, V. Albert, S. Rounsley and D. Sankoff. Scaffold filling, contig fusion and gene order comparison. *BMC Bioinformatics*, 11:304, 2010.

29 R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*, Oxford Univ. Press. 2006.

30 S. Yancopoulos, O. Attie and R. Friedberg. *Efficient sorting of genomic permutations by translocation, inversion and block interchange. Bioinformatics*, **21**:3340-3346, 2005.

31 B. Zhu. A retrospective on genomic preprocessing for comparative genomics. In Chauve et al., eds., *Models and Algorithms for Genome Evolution*, pages 183-206. Springer, 2013.

A Linear-Time Algorithm for the Copy Number Transformation Problem

Ron Shamir[1], Meirav Zehavi[1], and Ron Zeira[1]

1 **School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel**
`rshamir@post.tau.ac.il`
2 **School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel**
`meizeh@post.tau.ac.il`
3 **School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel**
`ronzeira@post.tau.ac.il`

Abstract

Problems of genome rearrangement are central in both evolution and cancer. Most evolutionary scenarios have been studied under the assumption that the genome contains a single copy of each gene. In contrast, tumor genomes undergo deletions and duplications, and thus the number of copies of genes varies. The number of copies of each gene along a chromosome is called its *copy number profile*. Understanding copy number profile changes can assist in predicting disease progression and treatment. To date, questions related to distances between copy number profiles gained little scientific attention. Here we focus on the following fundamental problem, introduced by Schwarz *et al.* (PLOS Comp. Biol., 2014): given two copy number profiles, u and v, compute the edit distance from u to v, where the edit operations are segmental deletions and amplifications. We establish the computational complexity of this problem, showing that it is solvable in linear time and constant space.

1998 ACM Subject Classification F.2.2 Pattern Matching

Keywords and phrases Genome Rearrangement, Copy Number

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.16

1 Introduction

The genome of a species evolves by undergoing small and large mutations over generations. Large mutations modify genome organization by rearrangement of genomic segments. Computational analysis of the process of *genome rearrangement* has been subject of extensive research over the last two decades [5]. The majority of these studies to date were restricted to a single copy of each gene, and were concerned with the reordering of segments. Extant models that do not make this assumption often result in NP-hard problems [12, 14, 15].

While most work on genome rearrangements to date was done in the context of species evolution, there is today great opportunity in analysis of cancer genome evolution. Cancer is a dynamic process characterized by the rapid accumulation of somatic mutations, which produce complex tumor genomes. Species evolution happens over eons and changes are carried over from one generation to the next. In contrast, cancer evolution happens within a single individual over a few decades. In many tumor genomes, a lot of the changes are segmental deletions and amplifications [16]. As a result, the number of copies of each gene along a chromosome, known as its *copy number profile*, changes during cancer development, compared to the normal genome that has two copies (or *alleles*) for each gene. Understanding these changes can assist in predicting disease progression and the outcome of medical interventions.

However, computational questions related to distances between copy number profiles received little scientific attention to date. Such questions are the topic of this paper.

Over the years, a variety of methods were used to determine the copy number profile of a cancer genome, at different resolutions. G-banding allows viewing the chromosomes bands [11]. FISH measures the copy numbers of tens to hundreds of targeted genes [4]. Array comparative genomic hybridization gives a higher resolution of copy number estimation for a cell population [17]. Most recently, deep sequencing techniques yield copy number profiles by using read depth data [10]. While it would have been preferable to analyze the genome (karyotype) itself and not its copy number profile, detection of structural variations from sequencing data is still problematic [7, 1]. Today it is a routine procedure to obtain detailed copy number profiles of cancer genomes, but utilizing them to understand cancer evolution is still an open problem.

Given two copy number profiles, the healthy tissue's and the tumor's, evaluating the distance between them can help in understanding cancer progression. A naïve measure of distance is the Euclidean distance between the two profiles [13]. Chowdhury *et al.* defined edit distance between copy number profiles obtained from FISH, where the edit operations are amplification or deletion of single genes, single chromosomes or the whole genome [3, 4, 2]. However, calculating these distances requires exponential time in the number of genes and therefore is limited to low resolution FISH data. The *TuMult* algorithm uses the number of breakpoints (loci where the copy numbers change) between two profiles as a simple distance measure [6].

Schwartz *et al.* introduced a model that admits amplification and deletion of contiguous segments [13]. The edit distance between two copy number profiles was defined as the minimum number of segmental deletions and duplications over all separations of the profiles into two alleles (a procedure known as *phasing*). Their algorithm *MEDICC* for computing the edit distance uses finite-state transducers (FSTs) [9] in order to model the profiles and efficiently compute the distance. However, the complexity of this method was not analyzed. Even without the phasing computation, the method needs to compose a 3-state transducer with itself N times, resulting in a transducer with 3^N states [13, 8]. The running time of FST procedures relies on the number of states and transitions, and in some cases may be exponential [9, 8].

Copy Number Transformation. We investigate the following problem, which underlies the model of [13]: Given two copy number profiles (*CNPs*), u and v, compute the minimum number of segmental duplications and deletions needed to transform u into v. We call this problem the Copy Number Transformation Problem (CNTP). A CNP is represented by a vector of nonnegative integers (the number of copies of each gene). A segmental deletion (amplification) decreases (resp. increases) by 1 the values of a contiguous segment of the vector, where zero values are not affected. Formal definitions are given in Section 2.

Our Contribution. We show that CNTP is solvable in linear time and constant space. The algorithm relies on several properties of the problem that we establish in Section 3.1, which may also be relevant to the analysis of other problems involving CNPs. Exploiting these properties results in a pseudo-polynomial dynamic programming algorithm for CNTP, presented in Section 3.2. In Section 3.3, by establishing that a certain function in the dynamic programming recursion is piecewise linear, we improve its performance and obtain our main result. For lack of space, some proofs are omitted.

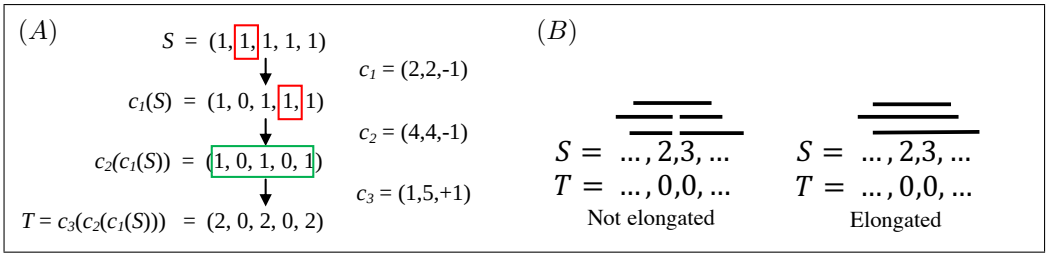


Figure 1 Copy number transformations. (A) The CNT $C = (c_1, c_2, c_3)$ transforms S into T. The size of C is 3. Red and green blocks indicate deletions and amplifications, respectively. (B) Elongated and non-elongated CNTs. Bold lines indicate the range of deletions.

2 Preliminaries

In this section, we give definitions and notation that are used throughout the paper. Let $n \in \mathbb{N}$. A *CN profile (CNP)* is a vector $V = (v_1, v_2, \ldots, v_n)$, where $v_i \in \mathbb{N} \cup \{0\}$. A *CN operation (CNO)* is a triple $c = (\ell, h, w)$, where $1 \leq \ell \leq h \leq n$ and $w \in \{-1, 1\}$. We say that a CNO $c = (\ell, h, -1)$ is a *deletion* and $c = (\ell, h, 1)$ is an *amplification*. Given a CNP $V = (v_1, v_2, \ldots, v_n)$ and a CNO $c = (\ell, h, w)$, we define the operation $c(V) = (c(v_1), c(v_2), \ldots, c(v_n))$ as follows. For each $i \in \{1, 2, \ldots, n\}$, if $\ell \leq i \leq h$ and $v_i \geq 1$, then $c(v_i) = v_i + w$, otherwise (i.e., if $i < \ell$ or $i > h$ or $v_i = 0$) $c(v_i) = v_i$. A triple $c = (\ell, h, w)$ with $h < \ell$ has no effect on the CNP, i.e., $c(V) = V$. Given two CNPs, $S = (s_1, s_2, \ldots, s_n)$ (source) and $T = (t_1, t_2, \ldots, t_n)$ (target), a *CN transformation (CNT)* is a vector $C = (c_1, c_2, \ldots, c_m)$, where $m \in \mathbb{N}$ and each $c_i = (\ell_i, h_i, w_i)$ is a CNO, such that $C(S) = c_m(c_{m-1}(\cdots(c_1(S)))) = T$. The *size* of C, denoted $|C|$, is m. An example is given in Fig. 1(A). Finally, we denote the number of operations of weight $w \in \{-1, 1\}$ affecting s_i by $op(C, w, i) = |\{(\ell, h, w) \in C : \ell \leq i \leq h\}|$. For example, in Fig. 1(A) $op(C, -1, 2) = 1$.

The *CN distance (CND)* from S to T, $\mathrm{dist}(S, T)$, is the smallest size of a CNT C that satisfies $C(S) = T$, where if no such CNT exists, $\mathrm{dist}(S, T) = \infty$. Note that dist is not symmetric. For example, for $S = (1)$ and $T = (0)$, $\mathrm{dist}(S, T) = 1$ but $\mathrm{dist}(T, S) = \infty$. Given two CNPs, $S = (s_1, s_2, \ldots, s_n)$ and $T = (t_1, t_2, \ldots, t_n)$, the Copy Number Transformation problem, CNTP, seeks $\mathrm{dist}(S, T)$ (if one exists). We say that a CNT C is *optimal* if it realizes $\mathrm{dist}(S, T)$, i.e., $|C| = \mathrm{dist}(S, T)$ (there may exist several optimal CNTs). We let $N = \max\{\max_{i=1}^{n}\{s_i\}, \max_{i=1}^{n}\{t_i\}\}$ denote the maximum copy number in the input. Finally, for all $1 \leq i \leq n$, we define $u_i = s_i - t_i$.

3 An Algorithm for CNTP

We first present an $O(nN^2)$-time, $O(N)$-space algorithm for CNTP that is based on dynamic programming (Sections 3.1 and 3.2). Recall that N is the maximal integer in the input, so that the algorithm is pseudo-polynomial. Then, we modify this algorithm to run in linear time (Section 3.3). On a high level, the modification is based on the observation that the table used by the algorithm to store values of partial solutions can be described by $O(n)$ piecewise linear functions, where each function encapsulates $O(N)$ entries of the table. We show that each function has only three linear segments and so the computation of an entry can be performed in time $O(1)$ rather than $O(N)$. Furthermore, since each function can be represented in a compact manner, the size of table shrinks from $O(nN)$ to $O(n)$. The precise definitions of the table and the functions are given in Sections 3.2 and 3.3. Our proof of the

correctness of the use of these functions requires a somewhat extensive case analysis that is presented separately in Section 3.4.

3.1 Key Propositions

We start by developing Alg1, an $O(nN^2)$-time dynamic programming algorithm for CNTP. Let $(S = (s_1, s_2, \ldots, s_n), T = (t_1, t_2, \ldots, t_n))$ be the input. Observe that there exists a CNT C such that $C(S) = T$ if and only if there does not exist an index $1 \leq i \leq n$ such that $s_i = 0$ and $t_i > 0$. Since the existence of such an index can be determined in linear time (where, if such an index is found, we return ∞), we will assume that $\text{dist}(S, T) < \infty$. To simplify the presentation, we further assume w.l.o.g. that $t_1, t_n \neq 0$. Indeed, if $t_1 = 0$ or $t_n = 0$, we can solve the input $(S' = (1, s_1, s_2, \ldots, s_n, 1), T' = (1, t_1, t_2, \ldots, t_n, 1))$ instead, since it holds that $\text{dist}(S, T) = \text{dist}(S', T')$. Finally, we assume w.l.o.g. that for all $1 \leq i \leq n$, $s_i > 0$. Indeed, if there exists $1 \leq i \leq n$ such that $s_i = 0$, then also $t_i = 0$, and we can solve the input $(S' = (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n), T' = (t_1 \ldots, t_{i-1}, t_{i+1}, \ldots, t_n))$ since $\text{dist}(S, T) = \text{dist}(S', T')$.

Alg1 exploits four key observations about the nature of the problem at hand, summarized as follows: (1) it is sufficient to examine CNTs where all of the deletions precede all of the amplifications; (2) it is sufficient to examine CNTs that do not contain both a deletion that affects s_i but not s_{i+1} and a deletion that affects s_{i+1} but not s_i, and the same is true for amplifications; (3) when seeking an optimal solution, it is not necessary to store information indicating how many deletions/amplifications affect s_i if $t_i = 0$; (4) the maximum number of deletions/amplifications that affect each s_i can be bounded by N.

To formally state the first observation, we need the following definition.

▶ **Definition 1.** A CNT $C = (c_1, c_2, \ldots, c_m)$ is *ordered* if for all $1 \leq i < j \leq m$, if c_j is a deletion, then c_i is also a deletion.

▶ **Proposition 2.** *There exists an optimal ordered CNT.*

We note that the "opposite" proposition, stating that there exists an optimal CNT where all of the amplifications precede all of the deletions, does not hold: consider, e.g., $S = (1, 1, 1, 1, 1)$ and $T = (2, 0, 2, 0, 2)$. To prove this proposition, we will need the following claim.

▶ **Claim 3.** *Let $C = (c_1, c_2, \ldots, c_m)$ be an optimal CNT and let i be an index such that $c_i = (\ell_i, h_i, 1)$ and $c_{i+1} = (\ell_{i+1}, h_{i+1}, -1)$. Then, there exists an optimal CNT $C' = (c_1, \ldots, c_{i-1}, c'_i, c'_{i+1}, c_{i+2}, \ldots, c_m)$, where $c'_i = (\ell'_i, h'_i, w'_i)$ and $c'_{i+1} = (\ell'_{i+1}, h'_{i+1}, w'_{i+1})$, such that one of the following conditions holds.*

1. $(h'_i - \ell'_i) + (h'_{i+1} - \ell'_{i+1}) < (h_i - \ell_i) + (h_{i+1} - \ell_{i+1})$.
2. $(h'_i - \ell'_i) + (h'_{i+1} - \ell'_{i+1}) = (h_i - \ell_i) + (h_{i+1} - \ell_{i+1})$ *and* $w'_i = -1$.

Proof. Consider the following exhaustive case-analysis.

1. $h_i < \ell_{i+1}$ or $h_{i+1} < \ell_i$: In this case, the segments corresponding to c_i and c_{i+1} are disjoint. Thus, we can simply define $c'_i = c_{i+1}$ and $c'_{i+1} = c_i$. Then, Condition 2 is satisfied.
2. $\ell_i \leq \ell_{i+1} \leq h_i \leq h_{i+1}$: Define $c'_i = (h_i + 1, h_{i+1}, -1)$ and $c'_{i+1} = (\ell_i, \ell_{i+1} - 1, 1)$. For any CNP $V = (v_1, v_2, \ldots, v_n)$, $c'_{i+1}(c'_i(V)) = c_{i+1}(c_i(V))$. This argument holds because an application of c_i which is followed by an application of c_{i+1} does not change any entry v_k such that $\ell_{i+1} \leq k \leq h_i$. We have that $C'(S) = T$. Since $|C'| = |C|$, C' is an optimal CNT. Now, Condition 1 is satisfied.

3. $\ell_{i+1} \leq \ell_i \leq h_{i+1} \leq h_i$: Define $c'_i = (\ell_{i+1}, \ell_i - 1, -1)$ and $c'_{i+1} = (h_{i+1} + 1, h_i, 1)$. As in the second case, we obtain an optimal CNT that satisfies Condition 1.
4. $\ell_i \leq \ell_{i+1} \leq h_{i+1} \leq h_i$: Define $c'_i = (\ell_i, \ell_{i+1} - 1, 1)$ and $c'_{i+1} = (h_{i+1} + 1, h_i, 1)$. As in the second case, we obtain an optimal CNT that satisfies Condition 1.
5. $\ell_{i+1} \leq \ell_i \leq h_i \leq h_{i+1}$: Define $c'_i = (\ell_{i+1}, \ell_i - 1, -1)$ and $c'_{i+1} = (h_i + 1, h_{i+1}, -1)$. As in the second case, we obtain an optimal CNT that satisfies Condition 1. ◀

As we show below, Claim 3 implies the existance of an ordered optimal CNT. In each of the cases in Claim 3, a local change is made in the CNT. Note however that just performing enough local operations does not guarantee reaching an ordered optimal CNT. For example, in a CNT with three consecutive CNOs, $c_i = (\ell_i, h_i, 1), c_{i+1} = (\ell_{i+1}, h_{i+1}, 1), c_{i+2} = (\ell_{i+2}, h_{i+2}, -1)$, one may loop between changing c_{i+1} into a deletion and then into an amplification.

Proof of Proposition 2. Let $\mathcal{C}$ be the set of optimal CNTs, and suppose, by way of contradiction, that it does not contain an ordered CNT. The three following phases sieve some solutions out of $\mathcal{C}$. Informally, we initially consider only optimal CNTs that minimize the sum of the sizes of the segments corresponding to their CNOs ($\mathcal{C}^1$); then, we further consider only the CNTs whose first amplification is as late as possible ($\mathcal{C}^2$); finally, we only take the CNTs whose first deletion after their first amplification is as early as possible ($\mathcal{C}^3$).

- Given $C = (c_1, c_2, \ldots, c_m) \in \mathcal{C}$, define $x(C) = \sum_{i=1}^{m}(h_i - \ell_i)$. Let $\mathcal{C}^1$ be the set of every $C \in \mathcal{C}$ for which there does not exist $C' \in \mathcal{C}$ such that $x(C) > x(C')$.
- Given $C = (c_1, c_2, \ldots, c_m) \in \mathcal{C}^1$, let $y(C)$ be the largest index $0 \leq i \leq m$ such that for all $1 \leq j \leq i$, c_j is a deletion. Note that $y(C) = 0$ if and only if c_1 is an amplification. Let $\mathcal{C}^2$ be the set of every $C \in \mathcal{C}^1$ for which there does not exist $C' \in \mathcal{C}^1$ such that $y(C) < y(C')$.
- Given $C = (c_1, c_2, \ldots, c_m) \in \mathcal{C}^2$, let $z(C)$ be the smallest index $i \in \{y(C) + 1, \ldots, m\}$ such that c_i is a deletion. By the definition of $y(C)$ and since C is not ordered, we have that $z(C)$ is well-defined and $z(C) \geq y(C) + 2$. Let $\mathcal{C}^3$ be the set of every $C \in \mathcal{C}^2$ for which there does not exist $C' \in \mathcal{C}^2$ such that $z(C) > z(C')$.

Since $\mathcal{C} \neq \emptyset$, we have that $\mathcal{C}^3 \neq \emptyset$. Thus, we can let $C = (c_1, c_2, \ldots, c_m)$ be a solution in $\mathcal{C}^3$. Let i be the smallest index such that c_i is an amplification and c_{i+1} is a deletion. Now, consider the conditions in Claim 3: if Condition 1 holds, we have a contradiction to the fact that $C \in \mathcal{C}^1$, while if Condition 2 holds, we have a contradiction either to the fact that $C \in \mathcal{C}^2$ (if $i = 1$ or c_{i-1} is a deletion) or to the fact that $C \in \mathcal{C}^3$ (otherwise). Thus, we conclude that $\mathcal{C}$ contains an ordered CNT. ◀

The other three propositions are stated without proof.

▶ **Definition 4.** A CNT C is *elongated* if for all $1 \leq i < n$ and $w \in \{-1, 1\}$,

$$\min\{op(C, w, i), op(C, w, i + 1)\} = |\{(\ell, h, w) \in C : \ell \leq i, i + 1 \leq h\}|.$$

Equivalently, C is elongated if no two amplifications (or deletions) "dovetail", i.e., one ending at i and the other starting at $i+1$. It is clear that for any CNT C, the inequality $\geq$ holds above (since $\{(\ell, h, w) \in C : \ell \leq i, i + 1 \leq h\}$ is a subset of both $\{(\ell, h, w) \in C : \ell \leq i \leq h\}$ and $\{(\ell, h, w) \in C : \ell \leq i + 1 \leq h\}$). Our second key proposition implies the inequality $\leq$ holds as well. An example for an elongated CNT is given in Fig. 1(B).

▶ **Proposition 5.** *Every ordered optimal CNT is elongated.*

To formalize our third key proposition, we need the following definition.

▶ **Definition 6.** A CNT C *skips zeros* if for every $1 \leq i < j \leq n$ such that for all $i < r \leq j, t_r = 0$ we have

$$op(C, -1, j) = \max\{\max_{r=i+1}^{j}\{s_r\}, op(C, -1, i)\}, \text{ and } op(C, 1, j) = op(C, 1, i).$$

In words, for a block of consecutive zeros in the target profile, all deletions that span the block also include its flanking positions. An example of a CNT that skips zeros is given in Fig. 2(A).

▶ **Proposition 7.** *There exists an optimal ordered CNT that skips zeros.*

For a position with positive target value, knowing the number of deletions that affected it uniquely determines the number of amplifications that affected it. This simple fact will help the efficiency of our procedures. Formally:

▶ **Observation 8.** *Let $1 \leq i \leq n$ be an index such that $t_i > 0$, and let $C = (c_1, c_2, \ldots, c_m)$ be a CNT such that $C(S) = T$. Then, $op(C, 1, i) = -u_i + op(C, -1, i)$.*

Finally, we formalize our fourth key proposition.

▶ **Definition 9.** A CNT C is *bounded* if for all $1 \leq i \leq n$ and every $w \in \{-1, 1\}$, we have $op(C, w, i) \leq N$.

▶ **Proposition 10.** *Every optimal ordered CNT that skips zeros is bounded.*

3.2 An $O(nN^2)$-Time Algorithm for CNTP

On a high-level, the dynamic programming algorithm works as follows. It considers increasing prefixes $S^i = (s_1, s_2, \ldots, s_i)$ and $T^i = (t_1, t_2, \ldots, t_i)$ of the input. It computes a table M having $n(N+1)$ entries where $\mathrm{M}[i, d]$ is the best value of a solution on (S^i, T^i) that uses exactly d deletions that affect the i^{th} position. The parameter d ranges between zero and N, and the values for each i are computed based on values $\mathrm{M}[j, \cdot]$ for a single specific $j < i$. In particular, at each point of time, only two rows of the table M are stored. By Propositions 2–10, the algorithm considers only ordered, elongated, zero-skipping and bounded solutions. We call such solutions *good*.

More formally, given $1 \leq i \leq n$ and $0 \leq d \leq N$, we say that a CNT C is an (i, d)-*CNT* if $C(S^i) = T^i$, $d = op(C, -1, i)$, and C is good. We say that an (i, d)-CNT C is *optimal* if there is no (i, d)-CNT C' such that $|C'| < |C|$. Our goal will be to ensure that each entry $\mathrm{M}[i, d]$ stores the size of an optimal (i, d)-CNT, where if no such CNT exists, it stores ∞. We do not compute entries $\mathrm{M}[i, d]$ such that $t_i = 0$; indeed, by relying on Property 7, we are able to skip such entries (though our recursive formula does consider CNs s_i referring to indices i such that $t_i = 0$). In this context, observe that any ordered CNT C such that $C(S) = T$ consists of at least u_i deletions that affect s_i, and if $t_i > 0$, it cannot consist of more than $s_i - 1$ such deletions (since after decreasing s_i to 0, it remains 0). Moreover, if $u_i \leq d < s_i$, there exists an (i, d)-CNT – by independently adjusting the value of each position $< i$ to its target position and the value at position i with d deletions, using operations of span 1.

▶ **Observation 11.** *Given $1 \leq i \leq n$ such that $t_i > 0$ and $0 \leq d \leq N$, there exists an (i, d)-CNT if and only if $u_i \leq d < s_i$.*

In case $s_i < t_i$, Observation 11 states that there exists an (i, d)-CNT if and only if $d < s_i$. In light of this observation, we will use the following assumption.

▶ **Assumption 12.** *In the computation below, we assume that* $\max\{u_i, 0\} \leq d < s_i$. *Entries* $M[i, d]$ *for which it is not true that* $\max\{u_i, 0\} \leq d < s_i$ *store* ∞.

By Observation 8, if a solution involved d deletions at position i with $t_i > 0$, then it involved $-u_i + d$ amplifications at that position. For convenience denote that number by $a(i, d) = -u_i + d$ for all $1 \leq i \leq n$ satisfying $t_i > 0$ and $\max\{u_i, 0\} \leq d < s_i$, and $a(i, d) = \infty$ otherwise.

For input profiles S, T, the algorithm precomputes two vectors .Given an index $1 < i \leq n$ such that $t_i > 0$, let $\text{prev}(i)$ denote the largest index $j < i$ such that $t_j > 0$. Moreover, if $\text{prev}(i) = i - 1$, let $Q_i = 0$, and otherwise let $Q_i = \max_{\text{prev}(i)<j<i}\{s_j\}$. A skipping zero solution will skip the positions between i and $prev(i)$ in the computation, but will make sure to perform at least Q_i deletions spanning the skipped positions.

Initialization. The initialization step sets all entries $\text{M}[1, d]$ as follows.

$$\text{M}[1, d] \leftarrow d + a(1, d).$$

Recursion. If $t_i = 0$ position i is skipped. Suppose that $i > 1$, $t_i > 0$ and $\max\{u_i, 0\} \leq d < s_i$. The order of the computation is determined by the first argument. The computation is summarized in the following formula.

$$\text{M}[i, d] \leftarrow \min_{0 \leq d' \leq N}\{\text{M}[\text{prev}(i), d'] + \max\{d - d', 0\} + \max\{a(i, d) - a(\text{prev}(i), d'), 0\}$$

$$+ \max\{Q_i - \max\{d, d'\}, 0\}\}.$$

Roughly speaking, to compute $M[i, d]$ we look back to the previous non zero position in T, and for each value d' in that position add the difference from d if needed, the number of amplifications to be added if needed, and the number of additional deletions if such are needed to take care of the skipped zero positions. After filling the table M, Alg1 returns $\min_{0 \leq d \leq N} \text{M}[n, d]$. An example if a filled table is given in Fig. 2(B).

Correctness. First, we claim that the entries of the table M are computed properly.

▶ **Lemma 13.** *For all* $1 \leq i \leq n$ *such that* $t_i > 0$ *and for all* $0 \leq d \leq N$, $M[i, d]$ *stores the size of an optimal* (i, d)*-CNT, where if no such CNT exists, it stores* ∞.

Proof. We prove the lemma by induction on the order of the computation.

The correctness of the initialization step follows from the definition of an (i, d)-CNT and Observation 8.

Now, fix $1 < i \leq n$ such that $t_i > 0$, and fix $\max\{u_i, 0\} \leq d < s_i$. Let m be the size of an optimal (i, d)-CNT. Suppose that the lemma is correct for all $i' < i$ and $0 \leq d' \leq N$. We need to show that $\text{M}[i, d] = m$.

First Direction. First, we show that $\text{M}[i, d] \leq m$. Let $C = (c_1, c_2, \ldots, c_m)$ be an optimal (i, d)-CNT, and for all $1 \leq j \leq m$, denote $c_j = (\ell_j, h_j, w_j)$. For all $1 \leq j \leq m$, let $c'_j = (\ell_j, \min\{h_j, \text{prev}(i)\}, w_j)$. Now, define $C' = (c'_1, c'_2, \ldots, c'_m)$. We further let $\widehat{C} = (\widehat{c}_1, \widehat{c}_2, \ldots, \widehat{c}_q)$ denote the CNT obtained from C' by removing all of the CNOs $c = (\ell, h, w)$ such that $h < \ell$. Denote $\widehat{d} = op(\widehat{C}, -1, \text{prev}(i))$. Observe that $\widehat{d} \leq N$ and that $\widehat{C}$ is a $(\text{prev}(i), \widehat{d})$-CNT (because C is an (i, d)-CNT). Therefore, by the induction hypothesis, $\text{M}[\text{prev}(i), \widehat{d}] \leq q$ (recall that $q = |\widehat{C}|$). If $\text{prev}(i) = i - 1$, then $Q_i = 0$ and since C is

ordered and elongated, by Observation 8 we have that $m-q = \max\{d-\widehat{d},0\}+\max\{a(i,d)-a(\text{prev}(i),\widehat{d}),0\}$. Thus, by the recursive formula, in this case we get that $\text{M}[i,d] \leq m$

Now, suppose that $\text{prev}(i) < i-1$. Then, since C is ordered and skips zeros, and by the definition of Q_i, the two following conditions hold.

1. $op(C,-1,i-1) = \max\{Q_i, op(C,-1,\text{prev}(i))\}$.
2. $op(C,1,i-1) = op(C,1,\text{prev}(i))$.

Thus, since C is ordered and elongated, by Observation 8 we have that $m-q = \max\{d-\widehat{d},0\}+\max\{a(i,d)-a(\text{prev}(i),\widehat{d}),0\}+\max\{Q_i-\max\{d,\widehat{d}\},0\}$. Again, by the recursive formula, this implies that $\text{M}[i,d] \leq m$.

Second Direction. Next, we show that $\text{M}[i,d] \geq m$. To this end, it is sufficient to show that there exists an (i,d)-CNT C such that $\text{M}[i,d] \geq |C|$. Let $\widehat{d}$ be an argument d' at which the value computed by using the recursive formula is minimized. By the inductive hypothesis, there exists a $(\text{prev}(i),\widehat{d})$-CNT $\widehat{C} = (\widehat{c}_1,\widehat{c}_2,\ldots,\widehat{c}_q)$ such that $\text{M}[\text{prev}(i),\widehat{d}] \geq q$. For all $1 \leq j \leq q$, denote $\widehat{c}_j = (\ell_j,h_j,w_j)$. Now, if $\text{prev}(i) = i-1$, define $\widetilde{C} = \widehat{C}$, and else define $\widetilde{C}$ as follows. For all $1 \leq j \leq q$, let $\widetilde{c}_j = (\ell_j,\widetilde{h},w_j)$, where $\widetilde{h} = h_j$ if $h_j < \text{prev}(i)$ and $\widetilde{h} = i-1$ otherwise. Let $\widetilde{C} = (\widetilde{c}_1,\widetilde{c}_2,\ldots,\widetilde{c}_q)$. Moreover, as long as there exists $\text{prev}(i) < j < i$ such that $op(\widetilde{C},-1,j) < s_j$, choose the smallest such j, and append to the beginning of $\widetilde{C}$ the CNO $(j,i-1,-1)$. Let C' be the CNT obtained at the end of this process. Denote $C' = (c'_1,c'_2,\ldots,c'_r)$, and for all $1 \leq j \leq r$, denote $c'_j = (\ell'_j,h'_j,w'_j)$. Now, let p and q be the number of deletions and amplifications in C' whose segments include $i-1$, respectively. If $p < d$, append to the beginning of C' $d-p$ "dummy" deletions of the form $(i,i-1,-1)$, and if $a(i,d) < q$, append to the end of C' $a(i,d)-q$ "dummy" amplifications of the form $(i,i-1,1)$. Let $C'' = (c''_1,c''_2,\ldots,c''_k)$ be the resulting CNT, and for all $1 \leq j \leq k$, denote $c''_j = (\ell''_j,h''_j,w''_j)$. Finally, we define C as follows. Let D (A) be a set of exactly d deletions (resp. amplifications) in C'' whose second argument is $i-1$. We let C be defined as C'', except that each CNO $(\ell,h,w) \in D \cup A$ is replaced by the CNO (ℓ,i,w). It is straightforward to verify that C is an (i,d)-CNT such that $|C| = q+\max\{d-\widehat{d},0\}+\max\{a(i,d)-a(\text{prev}(i),\widehat{d}),0\}+\max\{Q_i-\max\{d,\widehat{d}\},0\}$, which concludes the correctness of the second direction. ◀

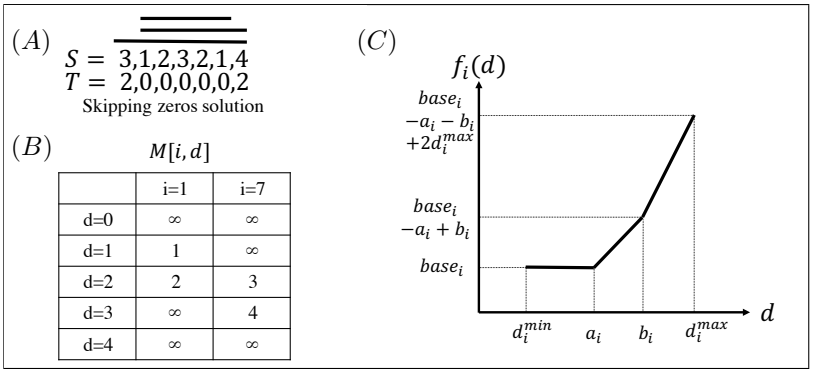
Now, we turn to consider the correctness and running time of Alg1.

▶ **Theorem 14.** Alg1 *solves* CNTP *in time* $O(nN^2)$ *and space* $O(N)$.

Proof. The table M contains $O(nN)$ entries, and each entry can be computed in time $O(N)$. Therefore, the time complexity of Alg1 is bounded by $O(nN^2)$. Moreover, for the computation of $\text{M}[i,\cdot]$, it is only necessary to keep $O(N)$ entries for position $\text{prev}(i)$, and therefore the space complexity is bounded by $O(N)$. Since every (n,d)-CNT C satisfies $C(S) = T$, and since for every good optimal CNT C, there exists $0 \leq d \leq N$ such that C is an (n,d)-CNT, we have that Lemma 13 implies that Alg1 returns the smallest size of a good optimal CNT (if such a CNT exists). By Propositions 2–10, such a CNT indeed exists, and therefore Alg1 solves CNTP. ◀

3.3 A Linear-Time Algorithm for CNTP

In this section we show how to modify Alg1 in order to obtain an algorithm, called Alg2, that solves CNTP in linear time. The central lemma that leads to this improvement states each column in the table M can be described by a piecewise linear function of at most three segments.



	i=1	i=7
d=0	∞	∞
d=1	1	∞
d=2	2	3
d=3	∞	4
d=4	∞	∞

Figure 2 (A) A skipping-zeros solution. Bold lines indicate deletions. (B) The DP $M[i,d]$ matrix for the two CNPs in (A). (C) An example of the piecewise linear function $f_i(d)$ described in Lemma 15. The number of segments is three but can be smaller, depending on the values involved.

To present this lemma, we need the following notation. For all $i \in \{1,2,\ldots,n\}$ such that $t_i > 0$, let $d_i^{min} = \max\{u_i, 0\}$ and $d_i^{max} = \max\{s_i - 1, 0\}$ be the least and largest values of d for which $M[i,d]$ is finite. Now, the function $f_i : \{d_i^{min}, \ldots, d_i^{max}\} \to \mathbb{N} \cup \{0\}$ will satisfy $f_i(d) = \mathrm{M}[i,d]$. Observe that the function f_i is discrete. We stress that in this section, we do not explicitly compute the entries of M – the definition of the functions concerns the values that would have been stored in these entries if they were computed by using Alg1.

▶ **Lemma 15.** *For each $i \in \{1,2,\ldots,n\}$ such that $t_i > 0$, there exist $base_i, a_i, b_i \in \mathbb{N} \cup \{0\}$ such that for all $d \in \{d_i^{min}, \ldots, d_i^{max}\}$:*

$$f_i(d) = \begin{cases} base_i & \text{if } d_i^{min} \le d \le a_i \\ (base_i - a_i) + d & \text{if } a_i \le d \le b_i \\ (base_i - a_i - b_i) + 2d & \text{if } b_i \le d \le d_i^{max} \end{cases}$$

Moreover, $base_1, a_1$ and b_1 can be computed in constant time, and for each $i \in \{2,3,\ldots,n\}$ such that $t_i > 0$, given $base_{\mathrm{prev}(i)}, a_{\mathrm{prev}(i)}$ and $b_{\mathrm{prev}(i)}$, $base_i, a_i$ and b_i can be computed in constant time.

An example is given in Fig. 2(C). The proof is based on Lemma 13 and an exhaustive case analysis, which, for the sake of clarity of presentation, is handled separately in Section 3.4.

Our algorithm, Alg2, performs the following computation:

1. Let $base_0 = a_0 = b_0 = 0$.
2. For $i = 1,2,\ldots,n$:
   a. If $t_i = 0$, skip the rest of the current iteration.
   b. Compute $base_i, a_i$ and b_i using $base_{\mathrm{prev}(i)}, a_{\mathrm{prev}(i)}$ and $b_{\mathrm{prev}(i)}$.
3. Return $base_n$.

We are now ready to prove our main result.

▶ **Theorem 16.** Alg2 *solves* CNTP *in time $O(n)$ and space $O(1)$.*

Proof. According to Lemma 15, the function $f_i(d) = \mathrm{M}[i,d]$ is a piecewise linear function described by three values. The correctness of Lemma 15 shows that step 3 calculates these values in constant time and space given the previous values. The time and space complexity of Alg2 follow directly.

Now, by the correctness of Alg1, it is sufficient to prove that Alg2 returns the value $\min_{0\leq d\leq N} \mathrm{M}[n,d]$. By Observation 11, $\min_{0\leq d\leq N} \mathrm{M}[n,d] = \min_{d_n^{min}\leq d\leq d_n^{max}} \mathrm{M}[n,d]$. By Lemma 15, we further have that $\min_{d_n^{min}\leq d\leq d_n^{max}} \mathrm{M}[n,d] = base_n$. Thus, by the inductive proof of Lemma 15, we conclude that Alg2 solves CNTP. ◀

3.4 Case Analysis

The purpose of this section is to prove the correctness of Lemma 15. That is, we want to show that $f_i(d)$ is a piecewise linear function described by three parameters, and these parameters can be calculated in constant time. To this end, let $j = \mathrm{prev}(i)$ and $R_i = u_j - u_i$. Accordingly, the term $a(i,d) - a(j,d')$ can be written as $R_i + d - d'$. Moreover, let d'_{opt} be the argument d' that minimizes the recursive formula we use to compute $\mathrm{M}[i,d]$ under certain conditions that will be clear from context.

We prove Lemma 15 by induction on i. To simplify the proof, let $a_0 = b_0 = base_0 = 0$ and $f_0(d) = 2d$ for every $0 \leq d \leq N$. This definition is equivalent to adding the new entries $s_0 = t_0 = N + 1$ (which do not affect the distance from S to T), and thus, it can serve as the basis of our induction. Next, suppose that Lemma 15 holds for $j = \mathrm{prev}(i) < i$, and we will prove that it holds for i.

The proof is based on an exhaustive case analysis that examines the position of Q_i relative to d_j^{min}, a_j, b_j and d_j^{max}, as well as the sign of R_i. For example, one of the cases is defined by the conditions $d_j^{min} \leq Q_i \leq a_j$, $R_i \geq 0$ and $a_j - R_i \leq Q_i$. In each case, we analyze the behavior of $\mathrm{M}[i,d]$ as we increase d. More precisely, we examine several intervals that together contain all of the values that can be assigned to d. For example, in the above mentioned case, we consider the intervals $d \leq a_j - R_j$, $a_j - R_j \leq d \leq Q_i$ and $Q_i \leq d$. For each interval, we let d'_{opt} be an argument d' that minimizes $\mathrm{M}[i,d]$ under the conditions of the examined case. These conditions along with d'_{opt} allow us to remove the minimization and maximization functions from the formula defining $\mathrm{M}[i,d]$, and thus we obtain $f_i(d)$. In the latter example, if $d \leq a_j - R_j$ we can choose $d'_{opt} = a_j$ and get $f_i(d) = \mathrm{M}[i,d] = \mathrm{M}[j,a_j] + \max\{d - a_j, 0\} + \max\{R_i + d - a_j, 0\} + \max\{Q_i - \max\{d, a_j\}, 0\}\} = base_j$. As a corollary of the analysis, we get that indeed $f_i(d)$ is piecewise linear, and that a_i, b_i and $base_i$ can be calculated in constant time given a_j, b_j, $base_j$, R_i and Q_i.

Due to lack of space, the details of the case analysis are omitted. The analysis shows that in all cases, $f_i(d)$ is indeed a piecewise linear function with at most three linear segments defined by some a_i, b_i, $base_i$. After applying straightforward operations that reorganize the analysis (to present the results in a compact manner), we obtain the algorithm PiecewiseAlg, whose pseudocode is given below. This algorithm performs step 2b of Alg2, i.e., it calculates a_i, b_i, $base_i$ given a_j, b_j, $base_j$ and Q_i in constant time and space.

PiecewiseAlg first calculates R_i, d_i^{min} and d_i^{max} based on s_i and t_i. Next, according to the sign of R_i and the relative position of Q_i in comparison to the previous a_j and b_j, the algorithm calculates the structure of $f_i(d)$ defined by a_i and b_i. Finally, since $f_i(d)$ is defined only for the range $d_i^{min} \leq d \leq d_i^{max}$, we calculate $base_i = f_i(d_i^{min})$. Similarly, we limit the values of a_i and b_i to that range.

4 Conclusion

In this paper, we initiated the study of distances between CNPs from a theoretical point of view. We focused on one fundamental problem, CNTP, and showed that it is solvable in linear time and constant space. To this end, we proved several properties of CNTP that may be useful in solving other problems involving CNPs. Our algorithm can be modified to return

Algorithm 1 PiecewiseAlg

Input: $s_i, t_i, Q_i, a_j, b_j, base_j$
Output: $a_i, b_i, base_i$

$R_i \leftarrow u_j - u_i$
$d_i^{min} \leftarrow \max\{u_i, 0\}$
$d_i^{max} \leftarrow \max\{s_i - 1, 0\}$
if $R_i \geq 0$ **then**
  if $Q_i \leq a_j$ **then**
    $a_i \leftarrow a_j - R_i$; $b_i \leftarrow b_j$.
  else if $a_j < Q_i \leq b_j$ **then**
    $a_i \leftarrow Q_i - R_i$; $b_i \leftarrow b_j$.
  else if $b_j < Q_i$ **then**
    $a_i \leftarrow b_j - R_i$; $b_i \leftarrow Q_i$.
  end if
else if $R_i < 0$ **then**
  if $Q_i \leq a_j$ **then**
    $a_i \leftarrow a_j$; $b_i \leftarrow b_j - R_i$.
  else if $a_j < Q_i \leq b_j$ **then**
    $a_i \leftarrow Q_i$; $b_i \leftarrow b_j - R_i$.
  else if $b_j < Q_i$ **then**
    $a_i \leftarrow \min\{Q_i, b_j - R_i\}$; $b_i \leftarrow \max\{Q_i, b_j - R_i\}$.
  end if
end if

$$base_i \leftarrow base_j + \max\{Q_i - a_j, 0\} + \begin{cases} 0 & \text{if } d_i^{min} \leq a_i \\ d_i^{min} - a_i & \text{if } a_i < d_i^{min} \leq b_i \\ 2d_i^{min} - a_i - b_i & \text{if } b_i < d_i^{min} \leq d_i^{max} \end{cases}$$

$a_i \leftarrow \max\{d_i^{min}, \min\{a_i, d_i^{max}\}\}$; $b_i \leftarrow \max\{a_i, \min\{b_i, d_i^{max}\}\}$.

a transformation that realizes $\mathrm{dist}(S, T)$ in linear time and linear space by backtracking the dynamic programming vector. We have implemented the algorithm as well as an ILP formulation of CNTP (the implementations are available upon request), and we intend to assess the performance of these approaches.

Many computational and combinatorial aspects in the analysis of distances between CNPs require further research. Indeed, this paper can be viewed as a first step towards understanding them. We intend to investigate variants of CNTP where one seeks a CNP that minimizes the overall distance from it to two (or more) CNPs that are given as input. Such variants are relevant to phylogenetic reconstruction in cancer (see [13]). Additional directions for further research involve the introduction of edit operations other than basic segmental deletions and amplifications, dealing with phasing of the profiles, as well as the handling of noise.

Acknowledgment. We thank the referees for many helpful comments. This study was supported by the Israeli Science Foundation (grant 317/13) and the Dotan Hemato-Oncology Research Center at Tel Aviv University. RZ was supported by fellowships from the Edmond J. Safra Center for Bioinformatics at Tel Aviv University and from the Israeli Center of Research Excellence (I-CORE) Gene Regulation in Complex Human Disease (Center No 41/11). MZ was supported by a fellowship from the I-CORE in Algorithms and the Simons

Institute for the Theory of Computing in Berkeley and by the Postdoctoral Fellowship for Women of Israel's Council for Higher Education.

References

1 Ryan P Abo, Matthew Ducar, Elizabeth P Garcia, Aaron R Thorner, Vanesa Rojas-Rudilla, Ling Lin, Lynette M Sholl, William C Hahn, Matthew Meyerson, Neal I Lindeman, Paul Van Hummelen, and Laura E MacConaill. BreaKmer: detection of structural variation in targeted massively parallel sequencing data using kmers. *Nucleic Acids Research*, nov 2014. doi:10.1093/nar/gku1211.

2 Salim Akhter Chowdhury, E Michael Gertz, Darawalee Wangsa, Kerstin Heselmeyer-Haddad, Thomas Ried, Alejandro A Schäffer, and Russell Schwartz. Inferring models of multiscale copy number evolution for single-tumor phylogenetics. *Bioinformatics*, 31(12):i258–67, jun 2015. doi:10.1093/bioinformatics/btv233.

3 Salim Akhter Chowdhury, Stanley E Shackney, Kerstin Heselmeyer-Haddad, Thomas Ried, Alejandro A Schäffer, and Russell Schwartz. Phylogenetic analysis of multiprobe fluorescence in situ hybridization data from tumor cell populations. *Bioinformatics*, 29(13):i189–98, jul 2013. doi:10.1093/bioinformatics/btt205.

4 Salim Akhter Chowdhury, Stanley E Shackney, Kerstin Heselmeyer-Haddad, Thomas Ried, Alejandro A Schäffer, and Russell Schwartz. Algorithms to model single gene, single chromosome, and whole genome copy number changes jointly in tumor phylogenetics. *PLoS Computational Biology*, 10(7):e1003740, jul 2014. doi:10.1371/journal.pcbi.1003740.

5 Guillaume Fertin, Anthony Labarre, Irena Rusu, Eric Tannier, and Stéphane Vialette. *Combinatorics of Genome Rearrangements*. MIT Press, 2009.

6 Eric Letouzé, Yves Allory, Marc A Bollet, François Radvanyi, and Frédéric Guyon. Analysis of the copy number profiles of several tumor samples from the same patient reveals the successive steps in tumorigenesis. *Genome Biology*, 11(7):R76, 2010. doi:10.1186/gb-2010-11-7-r76.

7 Andrew McPherson, Chunxiao Wu, Alexander W Wyatt, Sohrab Shah, Colin Collins, and S Cenk Sahinalp. nFuse: discovery of complex genomic rearrangements in cancer using high-throughput sequencing. *Genome Research*, 22(11):2250–61, nov 2012. doi:10.1101/gr.136572.111.

8 M Mohri. Weighted finite-state transducer algorithms. An overview. *Formal Languages and Applications*, 2004. URL: http://link.springer.com/chapter/10.1007/978-3-540-39886-8_29.

9 Mehryar Mohri. Edit-distance of weighted automata: General definitions and algorithms. *International Journal of Foundations of Computer Science*, 14(06):957–982, 2003.

10 Layla Oesper, Anna Ritz, Sarah J Aerni, Ryan Drebin, and Benjamin J Raphael. Reconstructing cancer genomes from paired-end sequencing data. *BMC Bioinformatics*, 13 Suppl 6(Suppl 6):S10, jan 2012. doi:10.1186/1471-2105-13-S6-S10.

11 D. Pinkel, T. Straume, and J. W. Gray. Cytogenetic analysis using quantitative, high-sensitivity, fluorescence hybridization. *Proceedings of the National Academy of Sciences*, 83(9):2934–2938, may 1986. doi:10.1073/pnas.83.9.2934.

12 Olivier Tremblay Savard, Yves Gagnon, Denis Bertrand, and Nadia El-Mabrouk. Genome halving and double distance with losses. *Journal of Computational Biology*, 18(9):1185–99, 2011. doi:10.1089/cmb.2011.0136.

13 Roland F Schwarz, Anne Trinh, Botond Sipos, James D Brenton, Nick Goldman, and Florian Markowetz. Phylogenetic quantification of intra-tumour heterogeneity. *PLoS Computational Biology*, 10(4):e1003535, apr 2014. doi:10.1371/journal.pcbi.1003535.

14 Mingfu Shao and Yu Lin. Approximating the edit distance for genomes with duplicate genes under DCJ, insertion and deletion. *BMC Bioinformatics*, 13(Suppl 19):S13, 2012. doi:10.1186/1471-2105-13-S19-S13.

15 Eric Tannier, Chunfang Zheng, and David Sankoff. Multichromosomal median and halving problems under different genomic distances. *BMC Bioinformatics*, 10(1):120, 2009. URL: http://www.biomedcentral.com/1471-2105/10/120, doi:10.1186/1471-2105-10-120.

16 The Cancer Genome Atlas Research Network. Integrated genomic analyses of ovarian carcinoma. *Nature*, 474(7353):609–15, jun 2011. doi:10.1038/nature10166.

17 Alexander Eckehart Urban, Jan O Korbel, Rebecca Selzer, Todd Richmond, April Hacker, George V Popescu, Joseph F Cubells, Roland Green, Beverly S Emanuel, Mark B Gerstein, Sherman M Weissman, and Michael Snyder. High-resolution mapping of DNA copy alterations in human chromosome 22 using high-density tiling oligonucleotide arrays. *Proceedings of the National Academy of Sciences*, 103(12):4534–9, mar 2006. doi:10.1073/pnas.0511340103.

On Almost Monge All Scores Matrices*

Amir Carmel[1], Dekel Tsur[2], and Michal Ziv-Ukelson[3]

1 **Department of Computer Science, Ben-Gurion University of the Negev, Israel**
karmela@cs.bgu.ac.il
2 **Department of Computer Science, Ben-Gurion University of the Negev, Israel**
dekelts@cs.bgu.ac.il
3 **Department of Computer Science, Ben-Gurion University of the Negev, Israel**
michaluz@cs.bgu.ac.il

Abstract

The all scores matrix of a grid graph is a matrix containing the optimal scores of paths from every vertex on the first row of the graph to every vertex on the last row. This matrix is commonly used to solve diverse string comparison problems. All scores matrices have the Monge property, and this was exploited by previous works that used all scores matrices for solving various problems. In this paper, we study an extension of grid graphs that contain an additional set of edges, called bridges. Our main result is to show several properties of the all scores matrices of such graphs. We also give an $O(r(nm+n^2))$ time algorithm for constructing the all scores matrix of an $m \times n$ grid graph with r bridges.

1998 ACM Subject Classification F.2.0 Nonnumerical Algorithms and Problems

Keywords and phrases Sequence alignment, longest common subsequences, DIST matrices, Monge matrices, all path score computations.

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.17

1 Introduction

String comparison is a fundamental problem in computer science that has applications in computational biology, computer vision, and other areas. String comparison is often performed using *sequence alignment*: The characters of two input strings are aligned to each other, and a *scoring function* gives a score to the alignment according to pairs of the aligned characters and unaligned characters. The goal of the string alignment problem is to seek an alignment that maximizes (or minimizes) the score. Common scoring functions are the *edit distance* score, and the *LCS* (longest common subsequence) score.

All scores matrices were introduced by Apostolico et al. [2] in order to obtain fast parallel algorithms for LCS computation. The *all scores matrix* of two strings A and B is a $(|B|+1) \times (|B|+1)$ matrix that stores the optimal alignment scores between A and every substring of B. More precisely, the element at row i and column j in the matrix is the optimal alignment score between A and $B[i..j]$. All scores matrices are also called DIST matrices [2] or semi-local score matrices [30].

The problem of efficiently constructing the all scores matrix of two strings has been studied in several papers [29, 1, 2, 17, 19, 20, 21, 22, 26, 31, 30]. All scores matrices provide a very powerful tool that can be also used for solving many problems on strings: optimal

* The research of A.C and D.T was partially supported by ISF grant 981/11. The research of A.C and M.Z-U was partially supported by ISF grant 478/10. The research of A.C, D.T, and M.Z-U was partially supported by the Frankel Center for Computer Science at Ben Gurion University of the Negev.

Figure 1 The crossing paths property yielding the Monge property in grid graphs. In Figure (a), the dark gray path is an optimal path from $(0,1)$ to $(4,5)$, and the light gray path is an optimal path from $(0,2)$ to $(4,4)$. These two paths cross at the vertex v. Figure (b) shows that a path from $(0,1)$ to $(4,4)$ can be obtained by taking the prefix of the dark gray path until v, and the suffix of the light gray path from v. Similarly, a path from $(0,2)$ to $(4,5)$ can be obtained by taking the prefix of the light gray path until v, and the suffix of the dark gray path from v. The sum of scores of the new paths is equal to the sum of scores of the former paths, which is equal to $D[1,5]+D[2,4]$. Since the new paths are not necessarily optimal, we obtain that $D[1,4]+D[2,5] \geq D[1,5]+D[2,4]$.

sequence alignment computation [9], approximate tandem repeats [24, 29], approximate non-overlapping repeats [5, 15, 29], common substring alignment [23, 25], sparse spliced alignment [16, 28], alignment of compressed strings [12], fully-incremental string comparison [14, 30], and other problems.

The alignment problem on strings A and B can be represented by using an $(|A|+1) \times (|B|+1)$ grid graph, known as the *alignment graph* (cf. [29]). Vertical (respectively, horizontal) edges correspond to alignment of a character in A (respectively, B) with a gap, and diagonal edges correspond to alignment of two characters in A and B. A path from the j-th vertex on row i to the j'-th vertex on row i' corresponds to an alignment of $A[i..i']$ and $B[j..j']$. The all scores matrix is therefore a matrix that contains the maximum (or minimum) scores of paths from vertices on the first row of the alignment graph to the vertices on the last row.

For an $n \times n$ matrix D, its *density* matrix $D^{\square}$ is an $(n-1) \times (n-1)$ matrix, where $D^{\square}[i,j] = D[i,j] + D[i-1,j-1] - D[i-1,j] - D[i,j-1]$. A matrix is called *Monge* if its density matrix is either non-negative or non-positive, and *unit Monge* if every row or column of the density matrix contains at most one non-zero element, and all the non-zero elements are equal to 1. All scores matrices of grid graphs are Monge matrices, this follows from the *crossing paths property* of the grid graph: If P_1 and P_2 are two paths from vertices on the first row to vertices on the last row of the graph, where on the first row the endpoint of P_1 appears before the endpoint of P_2, and on the last row the endpoint of P_1 appears after the endpoint of P_2, then the paths P_1 and P_2 must cross. This is illustrated in Figure 1. The Monge property is crucial for many of the algorithms for constructing all score matrices and for their applications. When the scoring function is the LCS score, the all scores matrix is unit Monge [31].

In this paper we extend the classical grid graphs to include an additional set of edges. These additional edges are of form $((i,j),(i',j'))$ where $i' \geq i$ and $j' \geq j$, and either $i' > i+1$ or $j' > j+1$ (see Figure 2a). We call these edges *bridges*. The bridges represent correspondence between pairs of substrings, one per each input sequence, which could be precomputed using an auxiliary adviser. In grid graphs enhanced with bridges, the crossing paths property no longer holds, and so the all scores matrix does not necessarily have the Monge property (see Figure 2).

Motivating examples of grid graphs enhanced with bridges are found in the domain of computational biology. Here, bridges are often used to incorporate additional information

that is known about the function and the physical structure of the aligned biomolecules and of their components [6, 11, 27]. One such example is found in a problem denoted "sequence alignment guided by motifs". Here, each one of the input sequences is first subjected to a parsing step in which meaningful substrings within it are identified and labeled. Substrings sharing the same label could be instantiations of the same motif shared by members of a protein family [13], particular DNA or RNA substrings of similar structure or function [4], or conserved molecular binding sites shared by multiple sequences that are combinatorially regulated in some biological pathway. Note that two substrings identified as belonging to the same motif family could be quite diverged in sequence, as it is the function, rather than the exact sequence, that is conserved in functional motifs. Yet, pairs of substrings sharing the same motif label are expected to be highly conserved in their location and order of occurrences within homologous genomic sequences. To incorporate this information, the alignment grid graph is enhanced with bridges reflecting pairs of substrings belonging to the same motif family, one from each sequence, and weights are assigned to these additional edges based on some a-priori scoring scheme expressing the importance of conserving the motifs in the alignment [4, 3, 8].

Our contribution and roadmap

In this paper, we consider grid graphs with bridges, and we assume that the non-bridge edges have 0/1 weights. We note that grid graphs with arbitrary bounded integer weights on the non-bridge edges can be reduced to grid graphs with 0/1 weights [30], and thus we will only consider the 0/1 weights scheme. However, this reduction is only quasi-polynomial: If the weights of non-bridge edges in the original grid graph are integers between $-C$ and C, the reduction increases the size of graph by a factor of $\Theta(C^2)$.

Our main result is to show the following properties of the non-zero values in the density matrix of an all scores matrix of a grid graph with r bridges (see Figure 2 for an example).

1. All the non-zero values in the density matrix are -1 or 1, except for $O(r^2)$ values in specific locations in the matrix.
2. In every row or column, except for r specific rows and r specific columns, the number of non-zero values is $O(r)$.

In particular, the number of non-zero values in the density matrix is $O(rn)$. Thus, if $r = o(n)$, the all scores matrix is "almost Monge". Property 1 will be proved in Section 2 (Theorem 3), and Property 2 in Sections 3. Due to space constraints, we only prove Property 2 for the case of a single bridge.

As a consequence of our main result, we obtain an algorithm for computing the all scores matrix for grid graphs with bridges in $O(r(nm + n^2))$ time. This algorithm is based on Schmidt's algorithm [29] for grid graphs with no bridges, and utilizes the properties described above. See below for comparison of this algorithm with previous results. The algorithm is given in Section 4 (Theorem 20).

Due to space constraints, some proofs were omitted.

Related work

Our algorithm mentioned above computes the optimal scores of paths from every vertex in a specific set of vertices (the vertices on the first row) to every vertex in the graph. This problem is called *multiple source shortest paths* (MSSP) problem. Algorithms for solving MSSP were proposed by several previous works. Schmidt [29] gave an MSSP algorithm for grid graphs with general weights. This algorithm constructs the all scores matrix in

$O((nm+n^2)\log n)$ time. For grid graphs with bounded integers weights, Schmidt gave an algorithm that constructs the all scores matrix in $O(mn)$ time. Tiskin [30] gave an MSSP algorithm for grid graphs with bounded integer weights that constructs the all scores matrix in $O(mn(\log\log n/\log n)^2)$ time. The results on grid graphs have been extended to general planar graphs. Klein [18] gave an algorithm for MSSP on planar graphs with general weights. The algorithm constructs the all scores matrix of a grid graph in $O((nm+n^2)\log n)$ time. Eisenstat and Klein [10] gave an algorithm for MSSP on undirected planar graphs with bounded integer weights, which is faster than the algorithm of Klein by a factor of $\Theta(\log n)$. Cabello et al [7] extended the result of Klein to graphs that can be embedded on a surface with genus g. Since a grid graph with r bridges can be embedded on a surface with genus r, the algorithm of Cabello et al. constructs the all scores matrix of a grid graph with r bridges and general weights in $O(rn^2\log^2 n)$ time. Cabello et al. also gave a randomized algorithm whose running time is $O(rn^2\log n)$ with high probability. Our algorithm improves the result of Cabello et al. by a factor of $\Theta(\log^2 n)$ for the case of bounded integer weights.

2 Preliminaries and basic problem properties

A *grid graph with bridges* is a directed graph $G=(V,E)$ whose vertex set is $V=\{(i,j): 0\le i\le m, 0\le j\le n\}$, and whose edge set consists of four types of edges:

1. Horizontal edges: $((i,j),(i,j+1))$ for every pair of indices i,j satisfying $0\le i\le m$ and $0\le j<n$.
2. Vertical edges: $((i,j),(i+1,j))$ for every pair of indices i,j satisfying $0\le i<m$ and $0\le j\le n$.
3. Diagonal edges: Edges of the form $((i,j),(i+1,j+1))$.
4. Bridges: Edges of the form $((i,j),(i',j'))$ where $i\le i'$ and $j\le j'$, and either $i+1<i'$ or $j+1<j'$.

In our framework, the horizontal and vertical edges have weight 0, the diagonal edges have weight 1, and each bridge has a positive integer weight. The *score* of a path is the sum of the weights of its edges. The 0/1 weights of the non-bridge edges correspond to the LCS scoring scheme for sequence alignment.

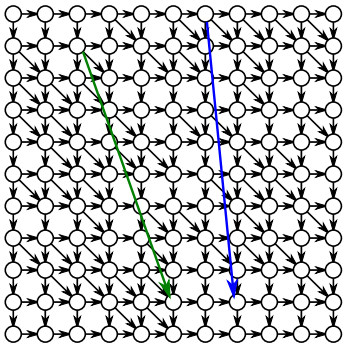
Let G be a grid graph with bridges $f_1,\ldots,f_r$. For a path P in G, if the first bridge P passes through is f_s, we say that P is an *s-path*. If P does not pass through bridges, we say that P is a *0-path*. The reason for focusing on the first bridge is to obtain a variant of the crossing path property which will be given in Lemma 12.

We define matrices D, $D^{\square}$, and D_{first} as follows (see Figure 2).

1. For $0\le i\le j\le n$, $D[i,j]$ is the maximum score of a path from $(0,i)$ to (m,j). For $i>j$, $D[i,j]=j-i$. The matrix D is called the *all scores matrix of G*.
2. For $1\le i,j\le n$, $D^{\square}[i,j]=(D[i,j]+D[i-1,j-1])-(D[i-1,j]+D[i,j-1])$. The matrix $D^{\square}$ is called the *density matrix of D*.
3. For $0\le i,j\le n$, $D_{\text{first}}[i,j]$ is a subset of the set $S=\{0,1,\ldots,r\}$ of bridge indices. For every $s\in S$, $s\in D_{\text{first}}[i,j]$ if and only if there is an s-path from $(0,i)$ to (m,j) with score $D[i,j]$.

To illustrate the importance of this matrix, consider a region in D_{first} in which all elements contain the same symbol s. Then, the crossing path property holds for indices in the region (since the two paths pass through f_s), so we obtain that the Monge property holds inside the region.

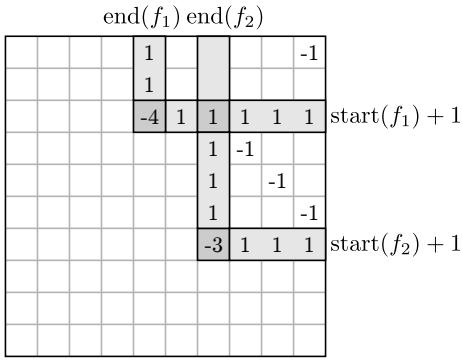
Next, we point out the entries in D and in $D^{\square}$ that are affected by a bridge in G. For some bridge $f_k=((i,j),(i',j'))$, we define $\text{start}(f_k)=j$ and $\text{end}(f_k)=j'$. We also define



(a) Grid graph

0	1	2	3	4	7	7	7	7	7	8
-1	0	1	2	3	7	7	7	7	7	7
-2	-1	0	1	2	7	7	7	7	7	7
-3	-2	-1	0	1	2	3	4	5	6	7
-4	-3	-2	-1	0	1	2	4	4	5	6
-5	-4	-3	-2	-1	0	1	4	4	4	5
-6	-5	-4	-3	-2	-1	0	4	4	4	4
-7	-6	-5	-4	-3	-2	-1	0	1	2	3
-8	-7	-6	-5	-4	-3	-2	-1	0	1	2
-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0

(b) D



(c) $D^{\square}$

0	0	0	0	0	1	1	1	1	0,1	0
	0	0	0	0	1	1	1	1	1	0,1
		0	0	0	1	1	1	1	1	0,1
			0	0	0	0	0,2	0	0	0
				0	0	0	2	0,2	0	0
					0	0	2	2	0,2	0
						0	2	2	2	0,2
							0	0	0	0
								0	0	0
									0	0
										0

(d) D_{first}

Figure 2 Figure (a) contains an example of a grid graph with two bridges. The weight of the bridge $f_1 = ((1,2),(9,5))$ is 7, and the weight of the bridge $f_2 = ((0,6),(9,7))$ is 4. The matrices D, $D^{\square}$, and D_{first} of the graph are shown in Figures (b), (c), and (d), respectively. Only the non-zero values of the density matrix are shown. The boundary indices are marked in grey, and the intersection indices are marked with darker grey. As stated in the text, each column or row of the density matrix can contain at most two negative values in non-boundary indices, and these values must be -1. The value of $D^{\square}[3,5]$ is $2+2-7-1=-4$, and these four values in D are marked in (b). The cause of the negative value in $D^{\square}[3,5]$ is that the corresponding optimal paths do not cross.

$E_k = \{(i,j) : i \leq \text{start}(f_k), j \geq \text{end}(f_k)\}$. In other words, E_k contains all indices (i,j) in D such that paths from $(0,i)$ to (m,j) can pass through f_k. The *boundary* of f_k is a set of indices in $D^{\square}$, defined as $B_k = \{(i, \text{end}(f_k)) : i \leq \text{start}(f_k)+1\} \cup \{(\text{start}(f_k)+1, j) : j \geq \text{end}(f_k)\}$. The two sets in the definition of B_k are called the *left boundary* and *bottom boundary* of f_k, respectively. We say that an index (i,j) in D is a *boundary index* if it is inside the boundary of some f_k. An index (i,j) is an *intersection index* if there are k, k' (possibly $k = k'$) such that (i,j) is in the left boundary of f_k and in the bottom boundary of $f_{k'}$ (see Figure 2c).

In the introduction we gave two properties of the density matrix. We now restate these properties using the definitions above.

1. Non-zero values other than $-1, 1$ can appear only at intersection indices.
2. In every row or column, the number of -1 values in non-boundary indices is at most r, and the number of 1 values in these indices is at most r.

Due to space constraints, we show these properties only for negative values. We will prove the properties for the positive values in the full version of the paper. Due to symmetry, we will show Property 2 only for columns.

Note that if $i > j+1$, $D^{\square}[i,j] = (j-i)+((j-1)-(i-1))-(j-(i-1))-((j-1)-i) = 0$. If $i = j+1$ then $D^{\square}[i,j] = -D[j,j]$, so in this case $D^{\square}[i,j] = 0$ unless there is a bridge f_k with $\text{start}(f_k) = \text{end}(f_k) = j$, in which case (i,j) is an intersection index. Similarly, for $i = j$, $D^{\square}[i,j] \in \{0,1\}$ unless one of the following two cases occurs: (1) There is a bridge f_k with $\text{start}(f_k) = j-1$ and $\text{end}(f_k) = j$. (2) There are bridges f_k and $f_{k'}$ with $\text{start}(f_k) = \text{end}(f_k) = j-1$ and $\text{start}(f_{k'}) = \text{end}(f_{k'}) = j$. In both cases (i,j) is an intersection index. Therefore, the properties stated above are satisfied for indices (i,j) with $i \geq j$. In the rest of the paper we will implicitly assume that indices (i,j) in $D^{\square}$ satisfy $i < j$.

We now give a proof for Property 1. For this goal, we need the following definition and lemma.

▶ **Definition 1.** A pair of indices $(i_1,j_1),(i_2,j_2)$ in the matrix D are said to be *bridge equivalent* if for every $1 \leq k \leq r$, $(i_1,j_1) \in E_k$ if and only if $(i_2,j_2) \in E_k$. In other words, $(i_1,j_1),(i_2,j_2)$ are bridge equivalent if paths from $(0,i_1)$ to (m,j_1) and paths from $(0,i_2)$ to (m,j_2) can pass through the same set of bridges.

▶ **Lemma 2.** *For every i,j,*
1. *If $(i,j-1)$ and (i,j) are bridge equivalent, $D[i,j-1] \leq D[i,j] \leq D[i,j-1]+1$.*
2. *If $(i-1,j)$ and (i,j) are bridge equivalent, $D[i,j] \leq D[i-1,j] \leq D[i,j]+1$.*

Property 1 is now obtained.

▶ **Theorem 3.** *Negative values other than -1 can appear only at intersection indices.*

Proof. Let (i,j) be an index that is not an intersection index. We have that either (1) $(i,j-1),(i,j)$ are bridge equivalent, and $(i-1,j-1),(i-1,j)$ are bridge equivalent, or (2) $(i-1,j),(i,j)$ are bridge equivalent, and $(i-1,j-1),(i,j-1)$ are bridge equivalent. In the former case we can rearrange the terms in the definition of $D^{\square}[i,j]$ and obtain that $D^{\square}[i,j] = \Delta_1 - \Delta_2$, where $\Delta_1 = D[i,j] - D[i,j-1]$ and $\Delta_2 = D[i-1,j] - D[i-1,j-1]$. We have $\Delta_1 - \Delta_2 < 0$, and by Lemma 2, $\Delta_1, \Delta_2 \in \{0,1\}$. It follows that $\Delta_1 = 0$ and $\Delta_2 = 1$, so $D^{\square}[i,j] = -1$. In the latter case we write $D^{\square}[i,j] = \Delta'_1 - \Delta'_2$ where $\Delta'_1 = D[i,j] - D[i-1,j]$ and $\Delta_2 = D[i,j-1] - D[i-1,j-1]$. By Lemma 2, in this case $\Delta'_1 = -1$ and $\Delta'_2 = 0$, so again $D^{\square}[i,j] = -1$. ◀

We next give several lemmas which will be used later to prove Property 2 in Section 3.

▶ **Definition 4.** An index (i,j) which is not a boundary index and for which $D^{\square}[i,j] < 0$ is called an *injury*. The submatrices $D[i-1..i, j-1..j]$ and $D_{\text{first}}[i-1..i, j-1..j]$ are called the submatrices of D and D_{first} corresponding to the injury, respectively.

▶ **Lemma 5.** *For an injury (i,j), $D[i-1..i, j-1..j] = \left(\begin{smallmatrix} x & x+1 \\ x & x \end{smallmatrix}\right)$ for some x.*

Proof. As in the proof of Theorem 3, $D^{\square}[i,j] = \Delta_1 - \Delta_2$, where $\Delta_1 = D[i,j] - D[i,j-1] = 0$ and $\Delta_2 = D[i-1,j] - D[i-1,j-1] = 1$. Thus, $D[i-1..i, j-1..j]$ is of the form $\left(\begin{smallmatrix} y & y+1 \\ x & x \end{smallmatrix}\right)$. We also have $D^{\square}[i,j] = \Delta'_1 - \Delta'_2$ where $\Delta'_1 = D[i,j] - D[i-1,j] = -1$ and $\Delta'_2 = D[i,j-1] - D[i-1,j-1] = 0$. The lemma follows. ◀

Our next goal is to show that every column in the density matrix contains at most r injuries. Consider a fixed column, and assume that this column has k injuries.

▶ **Definition 6.** Let $D_i = \left(\begin{smallmatrix} \gamma_i & \beta_i \\ \alpha_i & \delta_i \end{smallmatrix}\right)$ be the submatrix of D_{first} corresponding to the i-th injury, where the injuries are numbered in increasing row indices.

Our approach for proving that $k \leq r$ is based on showing properties of the D_{first} matrix. One of our techniques is showing that there are forbidden *structures* in D_{first}. For example, Lemma 10 below states that a structure consisting of a symbol $s \in \beta_i$ and $s \in \alpha_j$ for $j \geq i$ is forbidden. For the case of $r = 1$, applying this lemma with $i = j$ implies that there are only two possible values for α_i, β_i: either $\{0\}, \{1\}$ or $\{1\}, \{0\}$. If we assume conversely that there are $k = 2$ injuries, then there are four possible values for $\alpha_1, \beta_1, \alpha_2, \beta_2$. We then use Lemma 10 and an additional lemma (Lemma 12) that gives another forbidden structure in D_{first}, and show that each of these four cases cannot occur. This is a contradiction, and therefore there cannot be two injuries.

▶ **Lemma 7.** *For every i, j,*

1. *If $(i, j-1)$ and (i, j) are bridge equivalent,*
   - **(a)** *If $D[i, j-1] = D[i, j]$ then $D_{\text{first}}[i, j-1] \subseteq D_{\text{first}}[i, j]$.*
   - **(b)** *If $D[i, j-1] + 1 = D[i, j]$ then $D_{\text{first}}[i, j] \subseteq D_{\text{first}}[i, j-1]$.*
2. *If $(i-1, j)$ and (i, j) are bridge equivalent,*
   - **(a)** *If $D[i, j] = D[i-1, j]$ then $D_{\text{first}}[i, j] \subseteq D_{\text{first}}[i-1, j]$.*
   - **(b)** *If $D[i, j] + 1 = D[i-1, j]$ then $D_{\text{first}}[i-1, j] \subseteq D_{\text{first}}[i, j]$.*

Proof. We first prove the first part of the lemma. Choose a value $s \in D_{\text{first}}[i, j-1]$. Let P an s-path from $(0, i)$ to $(m, j-1)$ with score $D[i, j-1]$. The path P' obtained by appending the vertex (m, j) to P is an s-path from $(0, i)$ to (m, j) with score $D[i, j-1] = D[i, j]$ Therefore, $s \in D_{\text{first}}[i, j]$.

We next prove the second part of the lemma. Let $s \in D_{\text{first}}[i, j]$, and let P be an s-path from $(0, i)$ to (m, j) with score $D[i, j]$. Since $(i, j-1)$,(i, j) are bridge equivalent, P cannot pass through a bridge f with $\text{end}(f) > j-1$, so P has vertices on column $j-1$. Denote by k the maximal index such that $(k, j-1) \in P$. The path P' obtained by taking the prefix of P until $(k, j-1)$, and appending the vertices $(k+1, j-1), \ldots, (m, j-1)$ is an s-path from $(0, i)$ to $(m, j-1)$ with score at least $D[i, j] - 1 = D[i, j-1]$. It follows that $s \in D_{\text{first}}[i, j-1]$.

The proofs of the third and fourth parts of the lemma are symmetrical to the proofs of the first two parts, and thus they are omitted. ◀

The following lemma follows directly from Lemmas 5 and 7.

▶ **Lemma 8.** *For every i, $\alpha_i \subseteq \gamma_i \cap \delta_i$ and $\beta_i \subseteq \gamma_i \cap \delta_i$*

In order to restrict values of D in indices for which the entries in D_{first} contain the same symbol s, we define a matrix D_s as follows. For a symbol $s \in S$, let D_s be a matrix in which for every $(i, j) \in E_s$, $D_s[i, j]$ is the maximum score of an s-path from $(0, i)$ to (m, j). For $s = 0$, D_s is defined as above, except that $D_s[i, j]$ is defined for every $0 \leq i, j \leq n$. Note that $D_s[i, j] \leq D[i, j]$ for every (i, j) for which $D_s[i, j]$ is defined.

▶ **Lemma 9.** *For every $s \in S$, the matrix D_s has the Monge property.*

Proof. For $s = 0$ the lemma is true due to the crossing paths property for grid graphs with no bridges. For $s > 0$ we also have the crossing paths property: For every index (i, j), a maximum score s-path from $(0, i-1)$ to (m, j) must cross a maximum score s-path from $(0, i)$ to $(m, j-1)$ as both paths pass through f_s. Thus, the lemma follows. ◀

▶ **Lemma 10.** *For every $1 \leq i \leq j \leq k$, $\beta_i \cap \alpha_j = \emptyset$.*

Proof. Fix $i \leq j$, and assume conversely that $s \in \beta_i \cap \alpha_j$. By Lemma 5, the submatrices of D corresponding to injuries i and j are $D' = \left(\begin{smallmatrix} x & x+1 \\ x & \end{smallmatrix}\right)$ for some x, and $D'' = \left(\begin{smallmatrix} y & y+1 \\ y & \end{smallmatrix}\right)$ for some

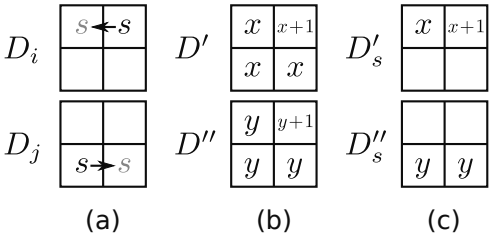


Figure 3 An illustration of the proof of Lemma 10. The grey s symbols in figure (a) represent values that are obtained using Lemma 8.

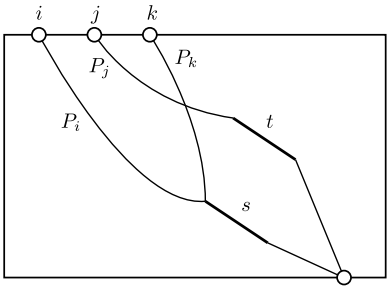


Figure 4 An illustration of the proof of Lemma 12.

y, respectively (see Figure 3). Let D'_s and D''_s be the submatrices of D_s that correspond to D' and D'', respectively. From the assumption $s \in \beta_i$ and Lemma 8, we have that $s \in \gamma_i$. Thus, the first row of D'_s is equal to the first row of D'. Similarly, we have that $s \in \delta_j$ and therefore the last row of D''_s is equal to the last row of D''. By taking the first row of D'_s and the last row of D''_s, we obtain that D_s contains a submatrix $\left(\begin{smallmatrix} x & x+1 \\ y & y \end{smallmatrix}\right)$ and therefore D_s does not have the Monge property. This contradicts Lemma 9. ◀

Finally, we give another forbidden structure in D_{first}, based on a variant of the crossing path property.

▶ **Definition 11.** Let $\preceq$ be a linear order on $S = \{0, 1, \ldots, r\}$ defined as follows. For every $i \neq j$, $i \preceq j$ if and only if $\text{start}(f_i) \leq \text{start}(f_j)$, where $\text{start}(f_0) = \infty$.

▶ **Lemma 12.** *Let d_i, d_j, d_k be values on rows i, j, k of some column i' of D_{first}, where $i < j < k$. Then, there are no $s, t \in S$ such that $s \preceq t$, $s \in d_i \cap d_k$, $t \notin d_i \cup d_k$, and $t \in d_j$.*

Proof. Assume conversely that there are $s, t \in S$ such that $s \preceq t$, $s \in d_i \cap d_k$, $t \notin d_i \cup d_k$, and $t \in d_j$. Note that $s \neq 0$ since by definition, $0 \not\preceq t$.

Let P_i, P_k be maximum score s-paths from $(0, i)$ and $(0, k)$ to (m, i'), respectively. Let P_j be a maximum score t-path from $(0, j)$ to (m, i'). Since $s \preceq t$, in the subgraph of G that contains the vertices above and to the left of the start vertex of f_s, the paths P_i, P_j, P_k do not pass through bridges (see Figure 4). Thus, P_j must cross one of the paths P_i and P_k. Assume without loss of generality that P_j crosses P_k.

Let P_j^1, P_k^1 denote the prefixes of P_j, P_k until the crossing point, and let P_j^2, P_k^2 denote the suffixes of P_j, P_k from the crossing point. Let y, z denote the scores of the paths P_j, P_k, respectively, and let a, b denote the score of the paths P_j^1, P_k^1, respectively.

We have that the path $P_k^1 \cup P_j^2$ is a t-path from $(0, k)$ to (m, i'). Since $t \notin d_k$, we conclude that $b + (y - a) < z$. Furthermore, due to the path $P_j^1 \cup P_k^2$ we have $a + (z - b) \leq y$. Summing the two inequalities above we obtain $y + z < y + z$, a contradiction. ◀

3 Properties of the one bridge case

In this section we assume the grid graph has a single bridge, $f = ((i_{\text{beg}}, j_{\text{beg}}), (i_{\text{end}}, j_{\text{end}}))$, and show that there is at most one injury in every column of $D^{\square}$.

▶ **Theorem 13.** *There is at most one injury in every column of $D^{\square}$.*

Proof. Fix some column of $D^{\square}$, and suppose conversely that there are at least two injuries in this column. Recall that $D_i = \left(\begin{smallmatrix} \gamma_i & \beta_i \\ \alpha_i & \delta_i \end{smallmatrix}\right)$ is the submatrix of D_{first} corresponding to the i-th

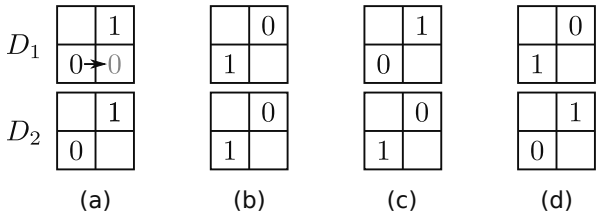


Figure 5 The four cases for two injuries in the proof of Theorem 13. The grey 0 in figure (a) represents a value that is obtained using Lemma 8.

injury. By Lemma 10, $\alpha_i \cap \beta_i = \emptyset$, and since α_i and β_i are non empty subsets of $S = \{0, 1\}$, it follows that either D_i is of the form $\left(\begin{smallmatrix} \cdot & 1 \\ 0 & \cdot \end{smallmatrix}\right)$ or D_i is of the form $\left(\begin{smallmatrix} \cdot & 0 \\ 1 & \cdot \end{smallmatrix}\right)$. Considering the first two injuries, there are four possible cases (see Figure 5):

1. D_1, D_2 are of the form $\left(\begin{smallmatrix} \cdot & 1 \\ 0 & \cdot \end{smallmatrix}\right)$.
2. D_1, D_2 are of the form $\left(\begin{smallmatrix} \cdot & 0 \\ 1 & \cdot \end{smallmatrix}\right)$.
3. D_1 is of the form $\left(\begin{smallmatrix} \cdot & 1 \\ 0 & \cdot \end{smallmatrix}\right)$ and D_2 is of the form $\left(\begin{smallmatrix} \cdot & 0 \\ 1 & \cdot \end{smallmatrix}\right)$.
4. D_1 is of the form $\left(\begin{smallmatrix} \cdot & 0 \\ 1 & \cdot \end{smallmatrix}\right)$ and D_2 is of the form $\left(\begin{smallmatrix} \cdot & 1 \\ 0 & \cdot \end{smallmatrix}\right)$.

We now show that each of the cases above yields a contradiction. In Case 1, we have from Lemma 8 that $0 \in \delta_1$. We now apply Lemma 12 on $\beta_1, \delta_1, \beta_2$ and obtain a contradiction (taking $s = 1$ and $t = 0$). Case 2 yields a contradiction using similar arguments. In Cases 3 and 4, we have $1 \in \beta_1 \cap \alpha_2$ and $0 \in \beta_1 \cap \alpha_2$, respectively, which is a contradiction to Lemma 10. ◀

Theorem 13 implies the following corollary.

▶ **Corollary 14.** *For $j \neq j_{\text{end}}$ there are at most two negative values in column j of $D^{\square}$. Moreover, the negative values can occur only in rows $1, \ldots, j_{\text{beg}} + 1$, and if there are two negative values, one of the values must be in row $j_{\text{beg}} + 1$.*

Proof. The column j can contain at most one injury. The column j has at most one boundary index, so there is at most one negative value in addition to the injury. ◀

4 Algorithm for constructing all-scores matrices

In this section we give an algorithm for computing the all scores matrix of a grid graph with bridges. Our algorithm is an extension of the algorithm of Schmidt for a grid graph without bridges [29]. We follow the presentation of Schmidt's algorithm which was given in Matarazzo et al. [26]. For clarity of presentation, we will first describe an algorithm for the case of a single bridge, and we will later handle the case of $r > 1$ bridges.

Let $f = ((i_{\text{beg}}, j_{\text{beg}}), (i_{\text{end}}, j_{\text{end}}))$ be the single bridge of the grid graph, and let W_f denote its weight.

Let $G_0, \ldots, G_m$ be grid graphs, where G_i is the subgraph of G induced by all the vertices (i', j) with $i' \leq i$. Let $D_0, \ldots, D_m$ be the all scores matrices of $G_0, \ldots, G_m$, respectively.

For $0 \leq k \leq n$, define

$$\textsc{DiffC}_{i,j}(k) = D_i[k, j+1] - D_i[k, j] \quad \text{and} \quad \textsc{DiffR}_{i,j}(k) = D_{i+1}[k, j] - D_i[k, j].$$

The following lemma follows from the definition above.

▶ **Lemma 15.** *For $i \leq m$, if all $\textsc{DiffC}_{i,j}(k)$ values are known for all j and k, then the matrix D_i can be constructed in $O(n^2)$ time.*

Our algorithm for constructing the all-scores matrix of G computes all $\textsc{DiffC}_{m,j}$ functions and then applies Lemma 15. The algorithm is based on the following properties of the $\textsc{DiffC}_{i,j}$ and $\textsc{DiffR}_{i,j}$ functions.

1. Most $\textsc{DiffC}_{i,j}$ and $\textsc{DiffR}_{i,j}$ functions have compact representations of size $O(1)$.
2. The compact representations of $\textsc{DiffC}_{i+1,j}$ and $\textsc{DiffR}_{i,j+1}$ can be computed efficiently from the compact representations of $\textsc{DiffC}_{i,j}$ and $\textsc{DiffR}_{i,j}$.

Property 1, stated in Lemma 18, is obtained from Lemmas 16 and 17 below. Property 2 is shown in Lemma 19. Similar properties were used in the algorithm of Schmidt for grid graphs with no bridges. In that case, *all* the $\textsc{DiffC}_{i,j}$ and $\textsc{DiffR}_{i,j}$ functions have compact representations, and the size of each representation is *exactly* 1. In the case of a grid graph with a bridge, we need additional steps to handle the $\textsc{DiffC}_{i,j}$ and $\textsc{DiffR}_{i,j}$ functions that do not have compact representations.

▶ **Lemma 16.** *For every $j \neq j_{\text{end}} - 1$, $\textsc{DiffC}_{i,j}(k) \in \{0, 1\}$, and for every $i \neq i_{\text{end}} - 1$, $\textsc{DiffR}_{i,j}(k) \in \{0, 1\}$.*

Proof. Follows immediately from Lemma 2. ◀

▶ **Lemma 17.**

1. *For every i and $j \neq j_{\text{end}} - 1$ there are $k_1 < k_2$ (where $k_2 = j_{\text{beg}} + 1$) such that for every $k \neq k_1, k_2$, $\textsc{DiffC}_{i,j}(k-1) \leq \textsc{DiffC}_{i,j}(k)$.*
2. *For every $i \neq i_{\text{end}} - 1$ and j there are $k_1 < k_2$ (where $k_2 = j_{\text{beg}} + 1$) such that for every $k \neq k_1, k_2$, $\textsc{DiffR}_{i,j}(k-1) \geq \textsc{DiffR}_{i,j}(k)$.*

Based on the previous two lemmas, we now give a compact representation for the $\textsc{DiffR}_{i,j}$ and $\textsc{DiffC}_{i,j}$ functions. The compact representation $\text{SR}_{i,j}$ of $\textsc{DiffR}_{i,j}$ is an array of "step" indices, i.e., the indices in which the value of $\textsc{DiffR}_{i,j}$ change. Formally, let I be the set of all indices k such that $\textsc{DiffR}_{i,j}(k) \neq \textsc{DiffR}_{i,j}(k-1)$. Then, $\text{SR}_{i,j}[l]$ is the l-th smallest element of I. The arrays $\text{SC}_{i,j}$ are defined similarly.

▶ **Lemma 18.** *For every $i \neq i_{\text{end}} - 1$ and $j \neq j_{\text{end}} - 1$, the arrays $\text{SR}_{i,j}$ and $\text{SC}_{i,j}$ have $O(1)$ elements each.*

In the following lemma we show that $\text{SC}_{i+1,j}$ and $\text{SR}_{i,j+1}$ can be computed efficiently from $\text{SC}_{i,j}$ and $\text{SR}_{i,j}$. For every $(i,j) \neq (i_{\text{end}} - 1, j_{\text{end}} - 1)$ and $k \leq j$, the optimal path from $(0,k)$ to $(i+1, j+1)$ passes through either $(i+1, j)$, (i,j), or $(i, j+1)$. Thus,

$$D_{i+1}[k, j+1] = \max\{D_{i+1}[k,j], D_i[k,j] + W_{i,j}, D_i[k, j+1]\},$$

where $W_{i,j} = 1$ if there is a diagonal edge entering (i,j) and $W_{i,j} = 0$ otherwise. From the equality above, the following formulas for $\textsc{DiffC}_{i+1,j}$ and $\textsc{DiffR}_{i,j+1}$ are obtained (see [26]).

▶ **Lemma 19.** *For $0 \leq k \leq j$ and $(i,j) \neq (i_{\text{end}} - 1, j_{\text{end}} - 1)$,*

$$\textsc{DiffC}_{i+1,j}(k) = \textsc{Max}_{i,j}(k) - \textsc{DiffR}_{i,j}(k) \text{ and } \textsc{DiffR}_{i,j+1}(k) = \textsc{Max}_{i,j}(k) - \textsc{DiffC}_{i,j}(k)$$

where $\textsc{Max}_{i,j}(k) = \max\{\textsc{DiffR}_{i,j}(k), W_{i,j}, \textsc{DiffC}_{i,j}(k)\}$.

We will use compact representations $\text{SMAX}_{i,j}$ for the $\textsc{Max}_{i,j}$ functions, which are defined similarly to the $\text{SR}_{i,j}$ arrays. From the definition of $\textsc{Max}_{i,j}$, every step of $\textsc{Max}_{i,j}$ corresponds to a step of either $\textsc{DiffC}_{i,j}$ or $\textsc{DiffR}_{i,j}$, and thus the number of elements in $\text{SMAX}_{i,j}$ is less then or equal to the number of elements in both $\text{SC}_{i,j}$ and $\text{SR}_{i,j}$. Therefore, $\text{SMAX}_{i,j}$ has $O(1)$ elements for $i \neq i_{\text{end}} - 1$ and $j \neq j_{\text{end}} - 1$.

Our algorithm for computing the arrays $\mathrm{SC}_{m,j}$, traverses every i, j and computes $\mathrm{SC}_{i+1,j}$ and $\mathrm{SR}_{i,j+1}$ from $\mathrm{SC}_{i,j}$ and $\mathrm{SR}_{i,j}$ using Lemma 19. When $i \neq i_{\mathrm{end}} - 1$ and $j \neq j_{\mathrm{end}} - 1$, this computation takes $O(1)$ time by Lemma 18. There are two cases which require a special treatment. The first case is $(i, j) = (i_{\mathrm{end}} - 1, j_{\mathrm{end}} - 1)$. In this case Lemma 19 can not be applied and thus $\mathrm{SC}_{i+1,j}$ and $\mathrm{SR}_{i,j+1}$ must be computed differently. Here we compute $D_{i+1}[k, j]$, $D_i[k, j+1]$, and $D_{i+1}[k, j+1]$, for every $0 \leq k \leq n$. Then, we use these values to compute $\textsc{DiffC}_{i+1,j}(k)$ and $\textsc{DiffR}_{i,j+1}(k)$ for all k, and finally we compute $\mathrm{SC}_{i+1,j}$ and $\mathrm{SR}_{i,j+1}$ from $\textsc{DiffC}_{i+1,j}$ and $\textsc{DiffR}_{i,j+1}$.

The values $D_{i+1}[k, j]$ and $D_i[k, j+1]$ are obtained using Lemma 15 in $O(n^2)$ time. To compute the $D_{i+1}[k, j+1]$ values, we use the equality

$$D_{i+1}[k, j+1] = \max\{D_i[k, j+1], D_i[k, j] + W_{i,j}, D_{i+1}[k, j], D_{i_{\mathrm{beg}}}[k, j_{\mathrm{beg}}] + W_f\}.$$

The second special case is when $i = i_{\mathrm{end}} - 1$ or $j = j_{\mathrm{end}} - 1$. In this case Lemma 18 does not apply. Therefore, we can only bound the time to compute $\mathrm{SC}_{i+1,j}$ and $\mathrm{SR}_{i,j+1}$ by $O(n)$. Since there are $O(n + m)$ pairs i, j for which this case occurs, the total contribution of this case to the time complexity of the algorithm is $O(n^2 + nm)$.

Extension to r bridges

The algorithm presented above can be extended to the case of $r > 1$ bridges. In this case, using the results of the next section we get that for every non-boundary pair (i, j), $\textsc{DiffC}_{i,j}$ and $\textsc{DiffR}_{i,j}$ are partitioned to $O(r)$ monotone regions and thus their compact representations $\mathrm{SC}_{i,j}, \mathrm{SR}_{i,j}$ have $O(r)$ elements. Therefore, the computation of $\mathrm{SC}_{i,j}, \mathrm{SR}_{i,j}$ for non-boundary indices takes $O(rnm)$ time. As for boundary indices, the technique remains as in the case of one bridge, only that now there are $O(r)$ intersection indices and $O(r(n+m))$ boundary indices. Summing the above, the following theorem is obtained.

▶ **Theorem 20.** *The all scores matrix for an $m \times n$ grid graph with r bridges can be constructed in $O(r(nm + n^2))$ time.*

Acknowledgments. We thank the anonymous CPM 2016 reviewers for their helpful comments.

References

1. C. E. R. Alves, E. N. Cáceres, and S. W. Song. An all-substrings common subsequence algorithm. *Discrete Applied Mathematics*, 156(7):1025–1035, 2008.
2. A. Apostolico, M. J. Atallah, L. L. Larmore, and S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. on Computing*, 19(5):968–988, 1990.
3. A. N. Arslan. Sequence alignment guided by common motifs. In *Proceedings of the fourth Biotechnology and Bioinformatics Symposium*, page 30. University of Colorado at Colorado Springs, 2007.
4. A. N. Arslan. Sequence alignment guided by common motifs described by context free grammars. In *Biotechnology and Bioinformatics Symp. (BIOT'07)*, 2007.
5. G. Benson. A space efficient algorithm for finding the best nonoverlapping alignment score. *Theoretical Computer Science*, 145(1&2):357–369, 1995.
6. H. L. Bodlaender, M. R. Fellows, and P. A. Evans. Finite-state computability of annotations of strings and trees. In *Combinatorial Pattern Matching*, pages 384–391. Springer, 1996.
7. S. Cabello, E. W. Chambers, and J. Erickson. Multiple-source shortest paths in embedded graphs. *SIAM J. on Computing*, 42(4):1542–1571, 2013.
8. J.-P. Comet and J. Henry. Pairwise sequence alignment using a prosite pattern-derived similarity score. *Computers & chemistry*, 26(5):421–436, 2002.

9 M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. *SIAM J. on Computing*, 32(5):1654–1673, 2003.

10 D. Eisenstat and P. N. Klein. Linear-time algorithms for max flow and multiple-source shortest paths in unit-weight planar graphs. In *Proc. 45th ACM Symposium on Theory Of Computing (STOC)*, pages 735–744, 2013.

11 P. A. Evans. *Algorithms and complexity for annotated sequence analysis*. PhD thesis, Citeseer, 1999.

12 D. Hermelin, G. M. Landau, S. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In *Proc. 26th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 529–540, 2009.

13 N. Hulo, A. Bairoch, V. Bulliard, L. Cerutti, E. De Castro, P. S. Langendijk-Genevaux, M. Pagni, and C. Sigrist. The prosite database. *Nucleic acids research*, 34(suppl 1):D227–D230, 2006.

14 Y. Ishida, S. Inenaga, A. Shinohara, and M. Takeda. Fully incremental LCS computation. In *Proc. 15th Symp. on Fundamentals of Computation Theory (FCT)*, pages 563–574, 2005.

15 S. Kannan and E. W. Myers. An algorithm for locating nonoverlapping regions of maximum alignment score. *SIAM J. on Computing*, 25(3):648–662, 1996.

16 C. Kent, G. M. Landau, and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.

17 S.-R. Kim and K. Park. A dynamic edit distance table. *J. of Discrete Algorithms*, 2(2):303–312, 2004.

18 P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proc. 16th Symposium on Discrete Algorithms (SODA)*, volume 5, pages 146–155, 2005.

19 P. Krusche and A. Tiskin. String comparison by transposition networks. In *Proc. of London Algorithmics Workshop*, pages 184–204, 2008.

20 P. Krusche and A. Tiskin. New algorithms for efficient parallel string comparison. In *Proc. 22nd Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 209–216, 2010.

21 G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM J. on Computing*, 27(2):557–582, 1998.

22 G. M. Landau, E. W. Myers, and M. Ziv-Ukelson. Two algorithms for LCS consecutive suffix alignment. *J. of Computer and System Sciences*, 73(7):1095–1117, 2007.

23 G. M. Landau, B. Schieber, and M. Ziv-Ukelson. Sparse LCS common substring alignment. *Information Processing Letters*, 88(6):259–270, 2003.

24 G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *J. of Computational Biology*, 8(1):1–18, 2001.

25 G. M. Landau and M. Ziv-Ukelson. On the common substring alignment problem. *J. of Algorithms*, 41(2):338–359, 2001.

26 U. Matarazzo, D. Tsur, and M. Ziv-Ukelson. Efficient all path score computations on grid graphs. *Theoretical Computer Science*, 525:138–149, 2014.

27 S. Mozes, D. Tsur, O. Weimann, and M. Ziv-Ukelson. Fast algorithms for computing tree lcs. *Theoretical Computer Science*, 410(43):4303–4314, 2009.

28 Y. Sakai. An almost quadratic time algorithm for sparse spliced alignment. *Theory of Computing Systems*, pages 1–22, 2009.

29 J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. of Computing*, 27(4):972–992, 1998.

30 A. Tiskin. Semi-local string comparison: algorithmic techniques and applications. arXiv:0707.3619v16.

31 A. Tiskin. Semi-local longest common subsequences in subquadratic time. *J. Discrete Algorithms*, 6(4):570–581, 2008.

Tight Tradeoffs for Real-Time Approximation of Longest Palindromes in Streams

Paweł Gawrychowski[1], Oleg Merkurev[2], Arseny M. Shur[3], and Przemysław Uznański[4]

1 **Institute of Informatics, University of Warsaw, Poland**
2 **Institute of Mathematics and Computer Science, Ural Federal University, Ekaterinburg, Russia**
3 **Institute of Mathematics and Computer Science, Ural Federal University, Ekaterinburg, Russia**
4 **Department of Computer Science, ETH Zürich, Switzerland**

Abstract

We consider computing a longest palindrome in the streaming model, where the symbols arrive one-by-one and we do not have random access to the input. While computing the answer exactly using sublinear space is not possible in such a setting, one can still hope for a good approximation guarantee. Our contribution is twofold. First, we provide lower bounds on the space requirements for randomized approximation algorithms processing inputs of length n. We rule out Las Vegas algorithms, as they cannot achieve sublinear space complexity. For Monte Carlo algorithms, we prove a lower bounds of $\Omega(M \log \min\{|\Sigma|, M\})$ bits of memory; here $M = n/E$ for approximating the answer with additive error E, and $M = \frac{\log n}{\log(1+\varepsilon)}$ for approximating the answer with multiplicative error $(1+\varepsilon)$. Second, we design three real-time algorithms for this problem. Our Monte Carlo approximation algorithms for both additive and multiplicative versions of the problem use $\mathcal{O}(M)$ words of memory. Thus the obtained lower bounds are asymptotically tight up to a logarithmic factor. The third algorithm is deterministic and finds a longest palindrome exactly if it is short. This algorithm can be run in parallel with a Monte Carlo algorithm to obtain better results in practice. Overall, both the time and space complexity of finding a longest palindrome in a stream are essentially settled.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases streaming algorithms, space lower bounds, real-time algorithms, palindromes, Monte Carlo algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.18

1 Introduction

In the streaming model of computation, a very long input arrives sequentially in small portions and cannot be stored in full due to space limitation. While well-studied in general, this is a rather recent trend in algorithms on strings. The main goals are minimizing the space complexity, i.e., avoiding storing the already seen prefix of the string explicitly, and designing real-time algorithm, i.e., processing each symbol in worst-case constant time. However, the algorithms are usually randomized and return the correct answer with high probability. The prime example of a problem on string considered in the streaming model is pattern matching, where we want to detect an occurrence of a pattern in a given text. It is somewhat surprising that one can actually solve it using polylogarithmic space in the streaming model, as proved by Porat and Porat [13]. A simpler solution was later given by Ergün et al. [6],

while Breslauer and Galil designed a real-time algorithm [3]. Similar questions studied in such setting include multiple-pattern matching [4], approximate pattern matching [5], and parametrized pattern matching [9].

We consider computing a longest palindrome in the streaming model, where a palindrome is a fragment which reads the same in both directions. This is one of the basic questions concerning regularities in texts and it has been extensively studied in the classical non-streaming setting, see [1, 7, 11, 12] and the references therein. The notion of palindromes, but with a slightly different meaning, is very important in computational biology, where one considers strings over $\{A, T, C, G\}$ and a palindrome is a sequence equal to its reverse complement (a reverse complement reverses the sequences and interchanges A with T and C with G); see [8] and the references therein for a discussion of their algorithmic aspects. Our results generalize to biological palindromes in a straightforward manner.

We denote by $\mathsf{LPS}(S)$ the problem of finding the maximum length of a palindrome in a string S (and a starting position of a palindrome of such length in S). Solving $\mathsf{LPS}(S)$ in the streaming model was recently considered by Berenbrink et al. [2], who developed tradeoffs between the bound on the error and the space complexity for additive and multiplicative variants of the problem, that is, for approximating the length of the longest palindrome with either additive or multiplicative error. Their algorithms were Monte Carlo, i.e., returned the correct answer with high probability. They also proved that any Las Vegas algorithm achieving additive error E must necessarily use $\Omega(\frac{n}{E}\log|\Sigma|)$ bits of memory, which matches the space complexity of their solution up to a logarithmic factor in the $E \in [1, \sqrt{n}]$ range, but leaves a few questions. Firstly, does the lower bound still hold for Monte Carlo algorithms? Secondly, what is the best possible space complexity when $E \in (\sqrt{n}, n]$ in the additive variant, and what about the multiplicative version? Finally, are there real-time algorithms achieving these optimal space bounds? We answer all these questions.

Our main goal is to settle the space complexity of LPS. We start with the lower bounds in Sect. 2. First, we show that Las Vegas algorithms cannot achieve sublinear space complexity at all. Second, we prove a lower bound of $\Omega(M \log\min\{|\Sigma|, M\})$ bits of memory for Monte Carlo algorithms; here $M = n/E$ for approximating the answer with additive error E, and $M = \frac{\log n}{\log(1+\varepsilon)}$ for approximating the answer with multiplicative error $(1+\varepsilon)$. Then, in Sect. 3 we design real-time Monte Carlo algorithms matching these lower bounds up to a logarithmic factor. Our real-time Monte Carlo algorithm for LPS with additive error E uses $\mathcal{O}(n/E)$ words of space, and our real time Monte Carlo algorithm for LPS with multiplicative error $\varepsilon \leq 1$ uses $\mathcal{O}\big(\frac{\log n}{\varepsilon}\big)$ words of space. Finally we present, for any m, a deterministic $\mathcal{O}(m)$-space real-time algorithm solving LPS exactly if the answer is less than m and detecting a palindrome of length $\geq m$ otherwise. The last result implies that if the input stream is fully random, then with high probability its longest palindrome can be found exactly by a real-time algorithm within logarithmic space.

Notation and Definitions. Let S denote a string of length n over an alphabet $\Sigma = \{1, \ldots, N\}$, where N is polynomial in n. We write $S[i]$ for the ith symbol of S and $S[i..j]$ for its *substring* (or *factor*) $S[i]S[i+1]\cdots S[j]$; thus, $S[1..n] = S$. A prefix (resp. suffix) of S is a substring of the form $S[1..j]$ (resp., $S[j..n]$). A string S is a *palindrome* if it equals its *reversal* $S[n]S[n-1]\cdots S[1]$. By $L(S)$ we denote the length of a longest palindrome which is a factor of S. log stands for the binary logarithm.

We consider the *streaming model* of computation: the input string $S[1..n]$ (called the *stream*) is read left to right, one symbol at a time, and cannot be stored, because the available space is sublinear in n. The space is counted as the number of $\mathcal{O}(\log n)$-bit machine words. An algorithm is *real-time* if the number of operations between two reads is bounded by a

constant. An approximation algorithm for a maximization problem has *additive error* E (resp., *multiplicative error* ε) if it finds a solution with the cost at least $OPT - E$ (resp., $\frac{OPT}{1+\varepsilon}$), where OPT is the cost of optimal solution; here both E and ε can be functions of the size of the input. In the $\mathsf{LPS}(S)$ problem, $OPT = L(S)$.

A *Las Vegas algorithm* always returns a correct answer, but its memory usage on the inputs of length n is a random variable. A *Monte Carlo algorithm* gives a correct answer with high probability and has deterministic working time. Here we call "high" the probability greater than $1 - 1/n$.

2 Lower Bounds

In this section we use Yao's minimax principle [15] to prove lower bounds on the space complexity of the LPS problem in the streaming model, where the length n and the alphabet Σ of the input stream are specified. We denote this problem by $\mathsf{LPS}_\Sigma[n]$.

▶ **Theorem 1** (Yao's minimax principle for randomized algorithms). *Let $\mathcal{X}$ be the set of inputs for a problem and $\mathcal{A}$ be the set of all deterministic algorithms solving it. Then, for any $x \in \mathcal{X}$ and $A \in \mathcal{A}$, the cost of running A on x is denoted by $c(a, x) \geq 0$.*

Let p be the probability distribution over $\mathcal{A}$, and let A be an algorithm chosen at random according to p. Let q be the probability distribution over $\mathcal{X}$, and let X be an input chosen at random according to q. Then $\max_{x\in\mathcal{X}} \mathbf{E}[c(A, x)] \geq \min_{a\in\mathcal{A}} \mathbf{E}[c(a, X)]$.

We use the above theorem for both Las Vegas and Monte Carlo algorithms. For Las Vegas algorithms, we consider only correct algorithms, and $c(x, a)$ is the memory usage. For Monte Carlo algorithms, we consider all algorithms (not necessarily correct) with memory usage not exceeding a certain threshold, and $c(x, a)$ is the correctness indicator function, i.e., $c(x, a) = 0$ if the algorithm is correct and $c(x, a) = 1$ otherwise.

Our proofs will be based on appropriately chosen padding. In some cases the padding requires a larger (but constant) alphabet, which can be always reduced to binary while increasing the size of the input by a constant factor. For the padding we will often use an infinite string $\nu = 0^1 1^1 0^2 1^2 0^3 1^3 \ldots$, or more precisely its prefixes of length d, denoted $\nu(d)$. Here 0 and 1 should be understood as two characters not belonging to the original alphabet. The longest palindrome in $\nu(d)$ has length $\mathcal{O}(\sqrt{d})$.

▶ **Theorem 2** (Las Vegas approximation). *Let $\mathcal{A}$ be a Las Vegas streaming algorithms solving $\mathsf{LPS}_\Sigma[n]$ with additive error $E \leq 0.99n$ or multiplicative error $(1 + \varepsilon) \leq 100$ using $s(n)$ bits of memory. Then $\mathbb{E}[s(n)] = \Omega(n \log |\Sigma|)$.*

Proof. By Theorem 1, it is enough to construct a probability distribution $\mathcal{P}$ over Σ^n such that for any deterministic algorithm $\mathcal{D}$, its expected memory usage on a string chosen according to $\mathcal{P}$ is $\Omega(n \log |\Sigma|)$ in bits.

Consider solving $\mathsf{LPS}_\Sigma[n]$ with additive error E. We define $\mathcal{P}$ as the uniform distribution over $\nu(\frac{E}{2})x\$\$y\nu(\frac{E}{2})^R$, where $x, y \in \Sigma^{n'}$, $n' = \frac{n}{2} - \frac{E}{2} - 1$, and \$ is a special character not in Σ. Let us look at the memory usage of $\mathcal{D}$ after having read $\nu(\frac{E}{2})x$. We say that x is "good" when the memory usage is at most $\frac{n'}{2} \log |\Sigma|$ and "bad" otherwise. Assume that $\frac{1}{2}|\Sigma|^{n'}$ of all x's are good, then there are two strings $x \neq x'$ such that the state of $\mathcal{D}$ after having read both $\nu(\frac{E}{2})x$ and $\nu(\frac{E}{2})x'$ is exactly the same. Hence the behavior of $\mathcal{D}$ on $\nu(\frac{E}{2})x\$\$x^R\nu(\frac{E}{2})^R$ and $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ is exactly the same. The former is a palindrome of length $n = 2n' + E + 2$, so $\mathcal{D}$ must answer at least $2n' + 2$, and consequently the latter also must contain a palindrome of length at least $2n' + 2$. A palindrome inside $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ is either fully contained

within $\nu(\frac{E}{2})$, x', x^R or it is a middle palindrome. But the longest palindrome inside $\nu(\frac{E}{2})$ is of length $\mathcal{O}(\sqrt{E}) < 2n'+2$ (for n large enough) and the longest palindrome inside x or x^R is of length $n' < 2n'+2$, so since we have exluced other possibilities, $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ contains a middle palindrome of length $2n'+2$. This implies that $x = x'$, which is a contradiction. Therefore, at least $\frac{1}{2}|\Sigma|^{n'}$ of all x's are bad. But then the expected memory usage of $\mathcal{D}$ is at least $\frac{n'}{4}\log|\Sigma|$, which for $E \leq 0.99n$ is $\Omega(n\log|\Sigma|)$ as claimed.

Now consider solving $\mathsf{LPS}_\Sigma[n]$ with multiplicative error $(1+\varepsilon)$. An algorithm with multiplicative error $(1+\varepsilon)$ can also be considered as having additive error $E = n\cdot\frac{\varepsilon}{1+\varepsilon}$, so if the expected memory usage of such an algorithm is $o(n\log|\Sigma|)$ and $(1+\varepsilon)\leq 100$ then we obtain an algorithm with additive error $E \leq 0.99n$ and expected memory usage $o(n\log|\Sigma|)$, which we already know to be impossible. ◀

Now we move to Monte Carlo algorithms. We first consider exact algorithms solving $\mathsf{LPS}_\Sigma[n]$; lower bounds on approximation algorithms will be then obtained by padding the input appropriately. We introduce an auxiliary problem $\mathsf{midLPS}_\Sigma[n]$, which is to compute the length of the middle palindrome in a string of even length n over an alphabet Σ.

▶ **Lemma 3.** *There exists a constant γ such that any randomized Monte Carlo streaming algorithm $\mathcal{A}$ solving $\mathsf{midLPS}_\Sigma[n]$ or $\mathsf{LPS}_\Sigma[n]$ exactly with probability $1-\frac{1}{n}$ uses at least $\gamma\cdot n\log\min\{|\Sigma|,n\}$ bits of memory.*

Proof. First we prove that if $\mathcal{A}$ is a Monte Carlo streaming algorithm solving $\mathsf{midLPS}_\Sigma[n]$ exactly using less than $\lfloor\frac{n}{2}\log|\Sigma|\rfloor$ bits of memory, then its error probability is at least $\frac{1}{n|\Sigma|}$.

By Theorem 1, it is enough to construct probability distribution $\mathcal{P}$ over Σ^n such that for any deterministic algorithm $\mathcal{D}$ using less than $\lfloor\frac{n}{2}\log|\Sigma|\rfloor$ bits of memory, the expected probability of error on a string chosen according to $\mathcal{P}$ is at least $\frac{1}{n|\Sigma|}$.

Let $n' = \frac{n}{2}$. For any $x\in\Sigma^{n'}$, $k\in\{1,2,\ldots,n'\}$ and $c\in\Sigma$ we define

$$w(x,k,c) = x[1]x[2]x[3]\ldots x[n']x[n']x[n'-1]x[n'-2]\ldots x[k+1]cx[k-1]\ldots x[2]x[1].$$

Now $\mathcal{P}$ is the uniform distribution over all such $w(x,k,c)$.

Choose an arbitrary maximal matching of strings from $\Sigma^{n'}$ into pairs (x,x') such that $\mathcal{D}$ is in the same state after reading either x or x'. At most one string per state of $\mathcal{D}$ is left unpaired, that is at most $2^{\lfloor\frac{n}{2}\log|\Sigma|\rfloor-1}$ strings in total. Since there are $|\Sigma|^{n'} = 2^{n'\log|\Sigma|} \geq 2\cdot 2^{\lfloor\frac{n}{2}\log|\Sigma|\rfloor-1}$ possible strings of length n', at least half of the strings are paired. Let s be longest common suffix of x and x', so $x = vcs$ and $x' = v'c's$, where $c\neq c'$ are single characters. Then $\mathcal{D}$ returns the same answer on $w(x,n'-|s|,c)$ and $w(x',n'-|s|,c)$, even though the length of the middle palindrome is exactly $2|s|$ in one of them, and at least $2|s|+2$ in the other one. Therefore, $\mathcal{D}$ errs on at least one of these two inputs. Similarly, it errs on either $w(x,n'-|s|,c')$ or $w(x,n'-|s|,c')$. Thus the error probability is at least $\frac{1}{2n'|\Sigma|} = \frac{1}{n|\Sigma|}$.

Now we can prove the lemma for $\mathsf{midLPS}_\Sigma[n]$ with a standard amplification trick. Say that we have a Monte Carlo streaming algorithm, which solves $\mathsf{midLPS}_\Sigma[n]$ exactly with error probability ε using $s(n)$ bits of memory. Then we can run its k instances simultaneously and return the most frequently reported answer. The new algorithm needs $\mathcal{O}(k\cdot s(n))$ bits of memory and its error probability ε_k satisfies:

$$\varepsilon_k \leq \sum_{2i<k}\binom{k}{i}(1-\varepsilon)^i\varepsilon^{k-i} \leq 2^k\cdot\varepsilon^{k/2} = (4\varepsilon)^{k/2}.$$

Let us choose $\kappa = \frac{1}{6}\frac{\log(4/n)}{\log(1/(n|\Sigma|))} = \frac{1}{6}\frac{1-o(1)}{1+\log|\Sigma|/\log n} = \Theta(\frac{\log n}{\log n+\log|\Sigma|}) = \gamma\cdot\frac{1}{\log|\Sigma|}\log\min\{|\Sigma|,n\}$, for some constant γ. Now we can prove the theorem. Assume that $\mathcal{A}$ uses less than

$\kappa \cdot n \log |\Sigma| = \gamma \cdot n \log \min\{|\Sigma|, n\}$ bits of memory. Then running $\lfloor \frac{1}{2\kappa} \rfloor \geq \frac{3}{4} \frac{1}{2\kappa}$ (which holds since $\kappa < \frac{1}{6}$) instances of $\mathcal{A}$ in parallel requires less than $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ bits of memory. But then the error probability of the new algorithm is bounded from above by

$$\left(\frac{4}{n}\right)^{\frac{3}{16\kappa}} = \left(\frac{1}{n|\Sigma|}\right)^{\frac{18}{16}} \leq \frac{1}{n|\Sigma|}$$

which we have already shown to be impossible.

The lower bound for $\mathsf{midLPS}_\Sigma[n]$ can be translated into a lower bound for solving $\mathsf{LPS}_\Sigma[n]$ exactly by padding the input so that the longest palindrome is centered in the middle. Let $x = x[1]x[2]\dots x[n]$ be the input for $\mathsf{midLPS}_\Sigma[n]$. We define

$$w(x) = x[1]x[2]x[3]\dots x[n/2]1\underbrace{000\dots0}_{n}1x[n/2+1]\dots x[n].$$

Now if the length of the middle palindrome in x is k, then $w(x)$ contains a palindrome of length at least $n+k+2$. In the other direction, any palindrome inside $w(x)$ of length $\geq n$ must be centered somewhere in the middle block consisting of only zeroes and both ones are mapped to each other, so it must be the middle palindrome. Thus, the length of the longest palindrome inside $w(x)$ is exactly $n+k+2$, so we have reduced solving $\mathsf{midLPS}_\Sigma[n]$ to solving $\mathsf{LPS}_\Sigma[2n+2]$. We already know that solving $\mathsf{midLPS}_\Sigma[n]$ with probability $1-\frac{1}{n}$ requires $\gamma \cdot n \log \min\{|\Sigma|, n\}$ bits of memory, so solving $\mathsf{LPS}_\Sigma[2n+2]$ with probability $1-\frac{1}{2n+2} \geq 1-\frac{1}{n}$ requires $\gamma \cdot n \log\{|\Sigma|, n\} \geq \gamma' \cdot (2n+2) \log \min\{|\Sigma|, 2n+2\}$ bits of memory. Notice that the reduction needs $\mathcal{O}(\log n)$ additional bits of memory to count up to n, but for large n this is much smaller than the lower bound if we choose $\gamma' < \frac{\gamma}{4}$. ◀

To obtain a lower bound for Monte Carlo additive approximation, we observe that any algorithm solving $\mathsf{LPS}_\Sigma[n]$ with additive error E can be used to solve $\mathsf{LPS}_\Sigma[\frac{n-E}{E+1}]$ exactly by inserting $\frac{E}{2}$ zeroes between every two characters, in the very beginning, and in the very end. However, this reduction requires $\log(\frac{E}{2}) \leq \log n$ additional bits of memory for counting up to $\frac{E}{2}$ and cannot be used when the desired lower bound on the required number of bits $\Omega(\frac{n}{E} \log \min(|\Sigma|, \frac{n}{E})$ is significantly smaller than $\log n$. Therefore, we need a separate technical lemma which implies that both additive and multiplicative approximation with error probability $\frac{1}{n}$ require $\Omega(\log n)$ bits of space.

▶ **Lemma 4.** *Let $\mathcal{A}$ be any randomized Monte Carlo streaming algorithm solving $\mathsf{LPS}_\Sigma[n]$ with additive error at most $0.99n$ or multiplicative error at most $n^{0.49}$ and error probability $\frac{1}{n}$. Then $\mathcal{A}$ uses $\Omega(\log n)$ bits of memory.*

Combining the reduction with the technical lemma and taking into account that we are reducing to a problem with string length of $\Theta(\frac{n}{E})$, we obtain the following.

▶ **Theorem 5** (Monte Carlo additive approximation)**.** *Let $\mathcal{A}$ be any randomized Monte Carlo streaming algorithm solving $\mathsf{LPS}_\Sigma[n]$ with additive error E with probability $1-\frac{1}{n}$. If $E \leq 0.99n$ then $\mathcal{A}$ uses $\Omega(\frac{n}{E} \log \min\{|\Sigma|, \frac{n}{E}\})$ bits of memory.*

Proof. Define $\sigma = \min\{|\Sigma|, \frac{n}{E}\}$.

Because of Lemma 4 it is enough to prove that $\Omega(\frac{n}{E} \log \sigma)$ is a lower bound when

$$E \leq \frac{\gamma}{2} \cdot \frac{n}{\log n} \log \sigma. \tag{1}$$

Assume that there is a Monte Carlo streaming algorithm $\mathcal{A}$ solving $\mathsf{LPS}_\Sigma[n]$ with additive error E using $o(\frac{n}{E} \log \sigma)$ bits of memory and probability $1-\frac{1}{n}$. Let $n' = \frac{n-E/2}{E/2+1} \geq \frac{n}{E}$ (the

last inequality, equivalent to $n \geq E \cdot \frac{E}{E-2}$ holds because $E \leq 0.99n$ and because we can assume that $E \geq 200$). Given a string $x[1]x[2]\dots x[n']$, we can simulate running $\mathcal{A}$ on $0^E x[1]0^{E/2}x[2]0^{E/2}x[3]\dots 0^{E/2}x[n']0^{E/2}$ to calculate R (using $\log(E/2) \leq \log n$ additional bits of memory), and then return $\left\lfloor \frac{R}{E/2+1} \right\rfloor$. We call this new Monte Carlo streaming algorithm $\mathcal{A}'$. Recall that $\mathcal{A}$ reports the length of the longest palindrome with additive error E. Therefore, if the original string contains a palindrome of length r, the new string contains a palindrome of length $\frac{E}{2} \cdot (r+1) + r$, so $R \geq r(E/2+1)$ and $\mathcal{A}'$ will return at least r. In the other direction, if $\mathcal{A}'$ returns r, then the new string contains a palindrome of length $r(E/2+1)$. If such palindrome is centered so that $x[i]$ is matched with $x[i+1]$ for some i, then it clearly corresponds to a palindrome of length r in the original string. But otherwise every $x[i]$ within the palindrome is matched with 0, so in fact the whole palindrome corresponds to a streak of consecutive zeroes in the new string and can be extended to the left and to the right to start and end with 0^E, so again it corresponds to a palindrome of length r in the original string. Therefore, $\mathcal{A}'$ solves $\mathsf{LPS}_\Sigma[n']$ exactly with probability $1 - \frac{1}{(n'(E/2+1)+E/2)} \geq 1 - \frac{1}{n'}$ and uses $o(\frac{n'(E/2+1)+E/2}{E/2} \log \sigma) + \log n = o(n' \log \sigma) + \log n$ bits of memory. Observe that by Lemma 3 we get a lower bound

$$\gamma \cdot n' \log \min\{|\Sigma|, n'\} \geq \frac{\gamma}{2} \cdot n' \log \sigma + \frac{\gamma}{2} \cdot \frac{n}{E} \log \sigma \geq \frac{\gamma}{2} \cdot n' \log \sigma + \log n$$

(where the last inequality holds because of Eq.(1)). Then, for large n we obtain contradiction as follows

$$o(n' \log \sigma) + \log n < \frac{\gamma}{2} \cdot n' \log \sigma + \log n. \qquad \blacktriangleleft$$

Finally, we consider multiplicative approximation. The proof follows the same basic idea as of Theorem 5, however is more technically involved. The main difference is that due to uneven padding, we are reducing to $\mathsf{midLPS}_\Sigma[n']$ instead of $\mathsf{LPS}_\Sigma[n']$.

▶ **Theorem 6** (Monte Carlo multiplicative approximation). *Let $\mathcal{A}$ be any randomized Monte Carlo streaming algorithm solving $\mathsf{LPS}_\Sigma[n]$ with multiplicative error $(1+\varepsilon)$ with probability $1 - \frac{1}{n}$. If $n^{-0.98} \leq \varepsilon \leq n^{0.49}$ then $\mathcal{A}$ uses $\Omega(\frac{\log n}{\log(1+\varepsilon)} \log \min\{|\Sigma|, \frac{\log n}{\log(1+\varepsilon)}\})$ bits of memory.*

3 Real-Time Algorithms

In this section we design real-time Monte Carlo algorithms within the space bounds matching the lower bounds from Sect. 2 up to a factor bounded by $\log n$. The algorithms make use of the hash function known as the *Karp-Rabin fingerprint* [10]. Let p be a fixed prime from the range $[n^{3+\alpha}, n^{4+\alpha}]$ for some $\alpha > 0$, and r be a fixed integer randomly chosen from $\{1, \dots, p-1\}$. For a string S, its forward hash and reversed hash are defined, respectively, as

$$\phi^F(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^i \right) \bmod p \quad \text{and} \quad \phi^R(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^{n-i+1} \right) \bmod p\,.$$

Clearly, the forward hash of a string coincides with the reversed hash of its reversal. Thus, if u is a palindrome, then $\phi^F(u) = \phi^R(u)$. The converse is also true modulo the (improbable) collisions of hashes, because for two strings $u \neq v$ of length m, the probability that $\phi^F(u) = \phi^F(v)$ is at most m/p. This property allows one to detect palindromes with high probability by comparing hashes. (This approach is somewhat simpler than the one of [2]; in particular, we don't need "fingerprint pairs" used there.) In particular, a real-time algorithm makes

Algorithm 1 Algorithm ABasic, ith iteration.

```
if i mod t_E = 0 then
    add I to the beginning of SP
read S[i]; compute I(i+1) from I; I ← I(i+1)
for all elements v of SP do
    if S[v.i..i] is a palindrome and answer.len < i−v.i+1 then
        answer ← (v.i, i−v.i+1)
```

$\mathcal{O}(n)$ comparisons and thus faces a collision with probability $\mathcal{O}(n^{-1-\alpha})$ by the choice of p. All further considerations assume that no collisions happen. For an input stream S, we denote $F^F(i,j) = \phi^F(S[i..j])$ and $F^R(i,j) = \phi^R(S[i..j])$. Let $I(i)$ denote the tuple $(i, F^F(1,i-1), F^R(1,i-1), r^{-(i-1)} \bmod p, r^i \bmod p)$. The proposition below is immediate from definitions and simple arithmetical manipulations.

▶ **Proposition 7.** *1. Given $I(i)$ and $S[i]$, the tuple $I(i+1)$ can be computed in $\mathcal{O}(1)$ time.*
2. Given $I(i)$ and $I(j+1)$, the string $S[i..j]$ can be checked for palindromicity in $\mathcal{O}(1)$ time.

3.1 Additive Error

▶ **Theorem 8.** *There is a real-time Monte Carlo algorithm solving the problem* LPS(S) *with the additive error $E = E(n)$ using $\mathcal{O}(n/E)$ space, where $n = |S|$.*

First we present a simple (and slow) algorithm which solves the posed problem, i.e., finds in S a palindrome of length $\ell(S) \geq L(S) - E$, where $L(S)$ is the length of the longest palindrome in S. Later this algorithm will be converted into a real-time one. We store the sets $I(j)$ for some values of j in a doubly-linked list SP in the decreasing order of j's. The longest palindrome currently found is stored as a pair $answer = (pos, len)$, where pos is its initial position and len is its length. Let $t_E = \lfloor \frac{E}{2} \rfloor$.

In Algorithm ABasic we add $I(j)$ to the list SP for each j divisible by t_E. This allows us to check for palindromicity, at ith iteration, all factors of the form $S[kt_E..i]$. We assume throughout the section that at the beginning of ith iteration the value $I(i)$ is stored in a variable I.

▶ **Proposition 9.** *Algorithm ABasic finds in S a palindrome of length $\ell(S) \geq L(S) - E$ using $\mathcal{O}(n/E)$ time per iteration and $\mathcal{O}(n/E)$ space.*

Proof. Both the time and space bounds arise from the size of the list SP, which is bounded by $n/t_E = \mathcal{O}(n/E)$; the number of operations per iteration is proportional to this size due to Proposition 7. Now let $S[i..j]$ be a longest palindrome in S. Let $k = \lceil \frac{i}{t_E} \rceil t_E$. Then $i \leq k < i + t_E$. At the kth iteration, $I(k)$ was added to SP; then the palindrome $S[k..j-(k-i)]$ was found at the iteration $j - (k-i)$. Its length is

$$j - (k-i) - k + 1 = j - i - 2(k-i) + 1 > (j - i + 1) - 2t_E = L(S) - 2\left\lfloor \frac{E}{2} \right\rfloor \geq L(S) - E,$$

as required. ◀

The resource to speed up Algorithm ABasic stems from the following

▶ **Lemma 10.** *During one iteration, the length answer.len increases by at most $2 \cdot t_E$.*

Algorithm 2 Algorithm A, ith iteration.

```
1: if i mod t_E = 0 then
2:     add I to the beginning of SP
3:     if i = t_E then
4:         sp ← first(SP)
5: read S[i]; compute I(i+1) from I; I ← I(i+1)
6: sp ← previous(sp)                                   ▷ if exists
7: while i − sp.i + 1 ≤ answer.len and (sp ≠ last(SP)) do
8:     sp ← next(sp)
9: for all existing v in {sp, next(sp)} do
10:    if S[v.i..i] is a palindrome and answer.len < i−v.i+1 then
11:        answer ← (v.i, i−v.i+1)
```

Proof. Let $S[j..i]$ be the longest palindrome found at the ith iteration. If $i-j+1 \le 2t_E$ then the statement is obviously true. Otherwise the palindrome $S[j+t_E..i-t_E]$ of length $i-j+1-2t_E$ was found before (at the $(i-t_E)$th iteration), and the statement holds again. ◀

Lemma 10 implies that at each iteration SP contains only two elements that can increase *answer.len*. Hence we get the following Algorithm A.

Due to Lemma 10, the cycle at lines 9–11 of Algorithm A computes the same sequence of values of *answer* as the cycle at lines 4–6 of Algorithm ABasic. Hence it finds a palindrome of required length by Proposition 9. Clearly, the space used by the two algorithms differs by a constant. To prove that an iteration of Algorithm A takes $\mathcal{O}(1)$ time, it suffices to note that the cycle in lines 7–8 performs at most two iterations. Theorem 8 is proved.

3.2 Multiplicative Error

▶ **Theorem 11.** *There is a real-time Monte Carlo algorithm solving the problem* $\mathsf{LPS}(S)$ *with multiplicative error* $\varepsilon = \varepsilon(n) \in (0,1]$ *using* $\mathcal{O}\big(\frac{\log n}{\varepsilon}\big)$ *space, where* $n = |S|$.

As in the previous section, we first present a simpler algorithm MBasic with non-linear working time and then upgrade it to a real-time algorithm. The algorithm must find a palindrome of length $\ell(S) \ge \frac{L(S)}{1+\varepsilon}$. The next lemma is straightforward.

▶ **Lemma 12.** *If* $\varepsilon \in (0,1]$, *the condition* $\ell(S) \ge L(S)(1-\varepsilon/2)$ *implies* $\ell(S) \ge \frac{L(S)}{1+\varepsilon}$.

We set $q_\varepsilon = \big\lceil \log \frac{2}{\varepsilon} \big\rceil$. The main difference in the construction of algorithms with the multiplicative and additive error is that here all sets $I(i)$ are added to the list SP, but then, after a certain number of steps, are deleted from it. The number of iterations the set $I(i)$ is stored in SP is determined by the time-to-live function $ttl(i)$ defined below. This function is responsible for both the correctness of the algorithm and the space bound.

Let $\beta(i)$ be the position of the rightmost 1 in the binary representation of i (the position 0 corresponds to the least significant bit). We define

$$ttl(i) = 2^{q_\varepsilon+2+\beta(i)}\,. \tag{2}$$

The definition is illustrated by Fig. 1. Now we state a few properties of the list SP.

▶ **Lemma 13.** *For any integers* $a \ge 1$ *and* $b \ge 0$, *there exists a unique integer* $j \in [a, a+2^b)$ *such that* $ttl(j) \ge 2^{q_\varepsilon+2+b}$.

Algorithm 3 Algorithm MBasic, ith iteration.

```
1: add I to the beginning of SP
2: for all v in SP do
3:     if v.i + ttl(v.i) = i then
4:         delete v from SP
5: read S[i]; compute I(i+1) from I; I ← I(i+1)
6: for all v in SP do
7:     if S[v.i..i] is a palindrome and answer.len < i−v.i+1 then
8:         answer ← (v.i, i−v.i+1)
```

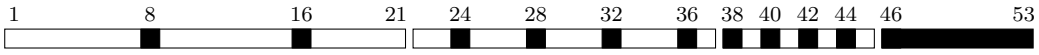


Figure 1 The state of the list SP after the iteration $i = 53$ ($q_\varepsilon = 1$ is assumed). Black squares indicate the numbers j for which $I(j)$ is currently stored. For example, (2) implies $ttl(28) = 2^{1+2+2} = 32$, so $I(28)$ will stay in SP until the iteration $28 + 32 = 60$.

Proof. By (2), $ttl(j) \geq 2^{q_\varepsilon+2+b}$ if and only if $\beta(j) \geq b$, i.e., j is divisible by 2^b by the definition of β. Among any 2^b consecutive integers, exactly one has this property. ◀

Figure 1 shows the partition of the range $(0, i]$ into intervals having lengths that are powers of 2 (except for the leftmost interval). In general, this partition consists of the following intervals, right to left:

$$(i-2^{q_\varepsilon+2}, i], (i-2^{q_\varepsilon+3}, i-2^{q_\varepsilon+2}], \ldots, (i-2^k, i-2^{k-1}], (0, i-2^k], \text{ where } k = \lceil \log n \rceil - 1. \quad (3)$$

Lemma 13 and (2) imply the following lemma on the distribution of the elements of SP.

▶ **Lemma 14.** *After each iteration, the first interval (resp., the last interval; each of the remaining intervals) in* (3) *contains* $2^{q_\varepsilon+2}$ *(resp., at most* $2^{q_\varepsilon+1}$*; exactly* $2^{q_\varepsilon+1}$*) elements of the list* SP*.*

The number of the intervals in (3) is $\mathcal{O}(\log n)$, so from Lemma 14 and the definition of q_ε

▶ **Lemma 15.** *After each iteration, the size of the list* SP *is* $\mathcal{O}(\frac{\log n}{\varepsilon})$*.*

▶ **Proposition 16.** *Algorithm MBasic finds a palindrome of length* $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$ *using* $\mathcal{O}(\frac{\log n}{\varepsilon})$ *time per iteration and* $\mathcal{O}(\frac{\log n}{\varepsilon})$ *space.*

Proof. Both the time per iteration and the space are dominated by the size of the list SP. Hence the required complexity bounds follow from Lemma 15. For the proof of correctness, let $S[i..j]$ be a palindrome of length $L(S)$. Further, let $d = \lfloor \log L(S) \rfloor$.

If $d < q_\varepsilon + 2$, the palindrome $S[i..j]$ will be found exactly, because $I(i)$ is in SP at the jth iteration:

$$i + ttl(i) \geq i + 2^{q_\varepsilon+2} \geq i + 2^{d+1} > i + L(S) > j\,.$$

Otherwise, by Lemma 13 there exists a unique $k \in [i, i + 2^{d-q_\varepsilon-1})$ such that $ttl(k) \geq 2^{d+1}$. Hence at the iteration $j - (k - i)$ the palindrome $S[i+(k-i)..j-(k-i)]$ will be found, because $I(k)$ is in SP at this iteration:

$$k + ttl(k) \geq i + ttl(k) \geq i + 2^{d+1} > j \geq j - (k - i)\,.$$

The length of this palindrome satisfies the requirement of the proposition:

$$j-(k-i)-(i+(k-i))+1 = L(S)-2(k-i) \geq L(S)-2^{d-q_\varepsilon} \geq L(S)-\frac{L(S)}{2^{q_\varepsilon}} \geq L(S)\big(1-\frac{\varepsilon}{2}\big)\,.$$

The reference to Lemma 12 finishes the proof. ◀

Now we speed up Algorithm MBasic. It has two slow parts: deletions from the list SP and checks for palindromes. By Lemmas 17, 18, $\mathcal{O}(1)$ checks is enough at each iteration.

▶ **Lemma 17.** *Suppose that at some iteration the list SP contains consecutive elements $I(d), I(c), I(b), I(a)$. Then $b-a \leq d-b$.*

Proof. Let j be the number of the considered iteration. Note that $a < b < c < d$. Consider the interval in (3) containing a. If $a \in (j-2^{q_\varepsilon+2}, j]$, then $b-a=1$ and $d-b=2$, so the required inequality holds. Otherwise, let $a \in (j-2^{q_\varepsilon+2+x}, j-2^{q_\varepsilon+2+x-1}]$. Then by (2) $\beta(a) \geq x$; moreover, any $I(k)$ such that $a < k \leq j$ and $\beta(k) \geq x$ is in SP. Hence, $b-a \leq 2^x$. By Lemma 14 each interval, except for the leftmost one, contains at least $2^{q_\varepsilon+1} \geq 4$ elements. Thus each of the numbers b, c, d belongs either to the same interval as a or to the previous interval $(j-2^{q_\varepsilon+2+x-1}, j-2^{q_\varepsilon+2+x-2}]$. Again by (2) we have $\beta(b), \beta(c), \beta(d) \geq x-1$. So $c-b, d-c \geq 2^{x-1}$, whence the result. ◀

We call an element v of SP *valuable at ith iteration* if $i - v.i + 1 > answer.len$ and $S[v.i..i]$ can be a palindrome. (That is, Algorithm MBasic does not store enough information to predict that the condition in its line 7 is false for v.)

▶ **Lemma 18.** *At each iteration, SP contains at most three valuable elements. Moreover, if $I(d'), I(d)$ are stored in consecutive elements of SP and $i-d' < answer.len \leq i-d$, where i is the number of the current iteration, then the valuable elements are consecutive in SP, starting with the one containing $I(d)$.*

Proof. Let d be as in the condition of the lemma and v be the element containing $I(d)$. If v is followed in SP by at most two elements, we are done. If it is not the case, let the three next elements be v_1, v_2, v_3, containing $I(c), I(b), I(a)$ respectively. If $S[v_3.i..i] = S[a..i]$ is a palindrome then $S[a+(b-a)..i-(b-a)]$ is also a palindrome. At the iteration $i-(b-a)$ the set $I(b)$ was in SP, so this palindrome was found. Hence, at the ith iteration the value $answer.len$ is at least the length of this palindrome, which is $i-a+1-2(b-a)$. By Lemma 17, $b-a \leq d-b$, implying $answer.len \geq i-a+1-(b-a)-(d-b) = i-d+1$. This inequality contradicts the definition of d; hence, $S[a..i]$ is not a palindrome. By the same argument, the elements following v_3 in SP do not produce palindromes as well. Thus, only the elements v, v_1, v_2 are valuable. ◀

Now we turn to deletions. The function $ttl(x)$ has the following nice property.

▶ **Lemma 19.** *The function $x \to x + ttl(x)$ is injective.*

Lemma 19 implies that at most one element is deleted from SP at each iteration. To perform this deletion in $\mathcal{O}(1)$ time, we need an additional data structure. By $BS(x)$ we denote a linked list of maximal segments of 1's in the binary representation of x. For example, the binary representation of $x = 12345$ and $BS(x)$ are as follows:

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	1	1	1	0	0	1

$BS(12345) = \{[0,0], [3,5], [10,10], [12,13]\}$

Clearly, $BS(x)$ uses $\mathcal{O}(\log x)$ space. The following lemma is easy.

Algorithm 4 Algorithm M, ith iteration.

```
1: add I to the beginning of SP
2: if i = 1 then
3:     sp ← first(SP)
4: compute BS[i] from BS; BS ← BS[i]; compute β(i) from BS
5: if QU(β(i)) is not empty then
6:     v ← element of SP pointed by first(QU(β(i)))
7:     if v = sp then
8:         sp ← next(sp)
9:     delete v; delete first(QU(β(i)))
10: add pointer to first(SP) to QU(β(i))
11: read S[i]; compute I(i+1) from I; I ← I(i+1)
12: sp ← previous(sp)                                   ▷ if exists
13: while i − sp.i + 1 ≤ answer.len and sp ≠ last(SP) do
14:     sp ← next(sp)
15: for all existing v in {sp, next(sp), next(next(sp))} do
16:     if S[v.i..i] is a palindrome and answer.len < i−v.i+1 then
17:         answer ← (v.i, i−v.i+1)
```

▶ **Lemma 20.** *Both $\beta(x)$ and $BS(x+1)$ can be obtained from $BS(x)$ in $\mathcal{O}(1)$ time.*

Thus, if we support one list BS which is equal to $BS(i)$ at the end of the ith iteration, we have $\beta(i)$. If $I(a)$ should be deleted from SP at this iteration, then $\beta(a) = \beta(i)$ (see Lemma 19). The following lemma is trivial.

▶ **Lemma 21.** *If $a < b$ and $ttl(a) = ttl(b)$, then $I(a)$ is deleted from SP before $I(b)$.*

By Lemma 21, the information about the positions with the same *ttl* (in other words, with the same β) are added to and deleted from SP in the same order. Hence it is possible to keep a queue $QU(x)$ of the pointers to all elements of SP corresponding to the positions j with $\beta(j) = x$. These queues constitute the last ingredient of our real-time Algorithm M.

Proof of Theorem 11. After every iteration, Algorithm M has the same list SP (see Fig. 1) as Algorithm MBasic, because these algorithms add and delete the same elements. Due to Lemma 18, Algorithm M returns the same answer as Algorithm MBasic. Hence by Proposition 16 Algorithm M finds a palindrome of required length. Further, Algorithm M supports the list BS of size $\mathcal{O}(\log n)$ and the array QU containing $\mathcal{O}(\log n)$ queues of total size equal to the size of SP. Hence, it uses $\mathcal{O}(\frac{\log n}{\varepsilon})$ space in total by Lemma 15. The cycle in lines 13–14 performs at most three iterations. Indeed, let z be the value of sp after the previous iteration. Then this cycle starts with $sp = previous(z)$ (or with $sp = z$ if z is the first element of SP) and ends with $sp = next(next(z))$ at the latest. By Lemma 20, both $BS(i)$ and $\beta(i)$ can be computed in $\mathcal{O}(1)$ time. Therefore, each iteration takes $\mathcal{O}(1)$ time. ◀

▶ Remark. Since for $\varepsilon \le 1$ the classes $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ and $\mathcal{O}\big(\frac{\log n}{\varepsilon}\big)$ coincide, Algorithm M uses space within a $\log n$ factor from the lower bound of Theorem 6. Further, let $\varepsilon = \varepsilon(n)$ be a growing function. Algorithm M can be transformed, with some additional technicalities, into a real-time algorithm which solves $\mathsf{LPS}(S)$ with the multiplicative error ε using $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ space. The basic idea of transformation is to replace all binary representations with those in base proportional to $1+\varepsilon$, and thus shrink the size of the lists SP and BS.

3.3 The Case of Short Palindromes

A typical string contains only short palindromes, as Lemma 22 below shows (for more on palindromes in random strings, see [14]). Knowing this, it is quite useful to have a deterministic real-time algorithm which finds a longest palindrome exactly if it is "short", otherwise reporting that it is "long". This idea is formalized in Theorem 23. Its proof is based on a modification of the Manacher's algorithm with a sliding window and lazy computation.

▶ **Lemma 22.** *If an input stream $S \in \Sigma^*$ is picked up uniformly at random among all strings of length n, where $n \geq |\Sigma|$, then for any positive constant c the probability that S contains a palindrome of length greater than $\frac{2(c+1)\log n}{\log |\Sigma|}$ is $\mathcal{O}(n^{-c})$.*

▶ **Theorem 23.** *Let m be a positive integer. There exists a deterministic real-time algorithm working in $\mathcal{O}(m)$ space, which solves $\mathsf{LPS}(S)$ exactly if $L(S) < m$, and otherwise finds a palindrome of length m or $m+1$ as an approximate solution to $\mathsf{LPS}(S)$.*

▶ Remark. Lemma 22 and Theorem 23 show a "practical" way to solve LPS. For example, one can run Algorithm M and Algorithm E, both in $\mathcal{O}(\log n)$ space, in parallel. Then either Algorithm E will give an exact answer (which happens with high probability if the input stream is a "typical" string) or both algorithms will produce approximations: one of fixed length and one with an approximation guarantee (modulo the hash collision).

References

1. A. Apostolico, D. Breslauer, and Z. Galil. Parallel detection of all palindromes in a string. *Theoret. Comput. Sci.*, 141:163–173, 1995.
2. P. Berenbrink, F. Ergün, F. Mallmann-Trenn, and E. Sadeqi Azer. Palindrome recognition in the streaming model. In *STACS 2014*, volume 25 of *LIPIcs*, pages 149–161, 2014.
3. D. Breslauer and Z. Galil. Real-time streaming string-matching. In *CPM 2011*, volume 6661 of *LNCS*, pages 162–172. Springer, 2011.
4. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *ESA 2015*, volume 9294 of *LNCS*, pages 361–372. Springer, 2015.
5. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. The k-mismatch problem revisited. In *SODA 2016*, pages 2039–2052. SIAM, 2016.
6. Funda Ergün, Hossein Jowhari, and Mert Saglam. Periodicity in streams. In *RANDOM 2010*, volume 6302 of *LNCS*, pages 545–559. Springer, 2010.
7. Z. Galil and J. Seiferas. A linear-time on-line recognition algorithm for "Palstar". *J. ACM*, 25:102–111, 1978.
8. Paweł Gawrychowski, Florin Manea, and Dirk Nowotka. Testing Generalised Freeness of Words. In *STACS 2014*, LIPIcs, pages 337–349, 2014.
9. Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *STACS 2015*, LIPIcs, pages 400–411, 2013.
10. R. Karp and M. Rabin. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.
11. D. E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.
12. G. Manacher. A new linear-time on-line algorithm finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975.
13. B. Porat and E. Porat. Exact and approximate pattern matching in the streaming model. In *FOCS 2009*, pages 315–323. IEEE Computer Society, 2009.

14 M. Rubinchik and A. M. Shur. The number of distinct subpalindromes in random words. arXiv:1505.08043 [math.CO], 2015.

15 Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *FOCS 1977*, pages 222–227. IEEE Computer Society, 1977.

Finding Maximal 2-Dimensional Palindromes

Sara H. Geizhals[1] and Dina Sokol[2]

1 **The Graduate Center of the City University of New York, 365 Fifth Avenue, New York, NY 10016, USA , shgeizhals@gmail.com**
2 **Brooklyn College of the City University of New York, 2900 Bedford Avenue, Brooklyn, NY 11210, USA , sokol@sci.brooklyn.cuny.edu**

Abstract

This paper extends the problem of palindrome searching into a higher dimension, addressing two definitions of 2D palindromes. The first definition implies a square, while the second definition (also known as a *centrosymmetric factor*), can be any rectangular shape. We describe two algorithms for searching a 2D text for maximal palindromes, one for each type of 2D palindrome. The first algorithm is optimal; it runs in linear time, on par with Manacher's linear time 1D palindrome algorithm. The second algorithm searches a text of size $n_1 \times n_2$ ($n_1 \geq n_2$) in $O(n_2)$ time for each of its $n_1 \times n_2$ positions. Since each position may have up to $O(n_2)$ maximal palindromes centered at that location, the second result is also optimal in terms of the worst-case output size.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases palindrome, pattern matching, 2-Dimensional, centrosymmetric factor

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.19

1 Introduction

Palindromes are strings that read the same forwards and backwards. Formally, a string P is a palindrome if it is of the form uau^R, where u is a non-empty string and u^R is its reverse; a is the empty string or a single character. a is called the *gap*, while u and u^R are called respectively the *left arm* and *right arm* of the palindrome. Palindromes have long drawn the attention of computer science researchers. The classical online and linear time palindrome algorithm is due to Manacher [21] in 1975. A palindrome variation called a *palstar*, which is loosely defined as the concatenation of palindromes, was studied as well in the 1970's by [19] and [11]. There is later research concerning searching for palindromes when there is a parallel model [3][4].

Other variations of palindrome search that have been studied more recently include gapped palindromes, complementary palindromes, approximate palindromes, and compressed palindromes. A *gapped palindrome* is when the size of the gap $|a| \geq 2$ [20]. *Complementary palindromes* are relevant in DNA, and it is where a character matches its complementary character instead of itself, e.g. *AACGTT*. [20]'s gapped palindrome algorithm can be adapted to find complementary gapped palindromes (which they refer to as *biological gapped palindromes*). *Approximate palindromes* have an allowed number of variations between the arms, and they have been studied in run-length compressed texts [6] as well as in the online model [2]. An interesting algorithm that searches for palindromes with edit distance of k is presented in [15]. Compressed palindromes have been studied as well under straight line programs [22].

Extending the concept of a palindrome to two dimensions has various applications. For example, face recognition technology exploits symmetry characteristics of the human face

Figure 1 Two examples of *sq2DP*s. The one on the left is written in Latin, while the one on the right is in Hebrew.

in order to extract a set of significant features [7]. Determining the global maximum of local reflectional symmetry in grey level images is related to genetic algorithms [18]. [14] creates palindromic shapes as representations of the intrinsic and extrinsic symmetries of 2D articulated planar shapes.

This paper presents algorithms that work with two different definitions of *2D palindromes*. The first definition dates back to the early Romans, and it can apply only to a square pattern; hence, we refer to it as a *sq2DP*. The second definition is termed a *centrosymmetric factor*[1] in [5]. This type of 2D palindrome can take on any rectangular shape and thus we refer to it as a *rect2DP*. To the best of our knowledge, the problem of searching a 2D text for maximal 2D palindromes has not been previously studied.

▶ **Definition 1.** A *sq2DP* is an $m \times m$ 2D pattern that admits four symmetries: identity, two diagonal reflections, and 180° rotation.

For example, Figure 1 portrays two famous *sq2DP*s. The one on the left is the first dateable representation of this type of 2D palindrome, and it was found in the ruins of Pompeii. The language is Latin, and it means, "the sower [planter] Arepo works with the help of wheel [a plough]" [10][24]. The one on the right is a *sq2DP* formed of five Hebrew words, of five characters each. It was written by Rabbi Abraham ibn Ezra (1089-1164) in response to the question as to whether a fly landing in honey makes the honey not kosher. Its translation is: "We have explained that the glutton [fly] who is in the honey was burned and incinerated [i.e., it disintegrated and therefore does not make it not kosher]" [23].

This problem is important to group theorists, in the field of mathematics. A *sq2DP* is a 2D pattern invariant under the subgroup generated by the two diagonal reflections of the dihedral group known as D_8. The D_8 group is one that is formed by the set of a square's eight symmetries (four rotations and four reflections).

▶ **Definition 2.** A *rect2DP* is a rectangular block of m_1 rows and m_2 columns that admits the two symmetries of identity and 180° rotation.

Each 2D palindrome has a *center*, which is the point that results in an equal number of columns to the left and right, as well as an equal number of rows above and below. The technical definition of the center differs slightly depending on the type and size of the 2D palindrome. Given an $m \times m$ *sq2DP*, if m is odd, the center is at location $(\lceil \frac{m}{2} \rceil, \lceil \frac{m}{2} \rceil)$. If m

[1] The paper studies the complexity of 2D Sturmian sequences in terms of the number of centrosymmetric factors that can occur in a 2D Sturmian sequence. Although their definition uses a binary alphabet due to its context, this paper assumes any bounded alphabet.

Table 1 Two examples of *rect2DP*s. The left one's center is between the two 3's.

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

n	e	v	e	r	o	d
d	o	r	e	v	e	n

is even, the center is in between rows and columns. Similarly for a *rect2DP*, if the number of rows (resp. columns) is even, the center is placed between rows (resp. columns). For example, the center of the left *rect2DP* in Table 1 is in the second row between the two 3's.

We present one algorithm for *sq2DP*, and one for *rect2DP*. Both algorithms consider each possible position of a center, and then locate the 2D palindrome(s) centered there. As with 1D palindromes, we are interested only in the 2D palindromes that are *maximal*. A *sq2DP* of size $m \times m$ is *maximal* if enlarging it by one on all sides – to size $(m+2) \times (m+2)$ – results in a pattern that is not a *sq2DP*. There is exactly one maximal *sq2DP* centered at each possible center position. Similarly, a *rect2DP* is maximal if it is not contained within a larger *rect2DP* with the same center. For a given text position, a maximal *rect2DP* is the highest *rect2DP* for its width or the widest *rect2DP* for its height. Thus, there may be several maximal *rect2DP* centered at a given position.

The remainder of this paper is organized as follows: Section 2 presents an algorithm for locating all maximal *sq2DP* in a given 2D text. Its input is T of size $n \times n$, and its runtime is linear, i.e. $O(n^2)$. This is on par with Manacher's linear palindrome algorithm and stems from the fact that there is exactly one maximal palindrome centered at each position. In Section 3, we describe a different algorithm that searches for maximal *rect2DP*. Its input is T of size $n_1 \times n_2$ (where $n_1 \geq n_2$), and its runtime is $O(n_1 n_2^2)$. We conclude in Section 4 with our plans for future work.

2 Square 2D Palindrome

The input to the algorithm is a 2D text T over a bounded alphabet Σ. For simplicity, we assume T is of size $n \times n$, however, the algorithm can be used for any rectangular text. The algorithm searches T for all maximal *sq2DP* that occur in T.

The basis of the algorithm is that the symmetry property of palindromes in one dimension also applies to *sq2DP*. In 1D, the palindromes that are substrings of the left arm of a palindrome will appear as well in the right arm. To illustrate, consider the lengths of the maximal palindromes centered at each position in the string *abacaba*: 1,3,1,7,1,3,1, and note the symmetry of this numerical list (around its center).

Henceforth we distinguish between the two diagonals of a square as follows. The diagonal that extends from the upper left corner to the lower right corner is called the *main diagonal*, and the diagonal that extends from the upper right corner to the lower left corner is called the *anti-diagonal*. Assume that P is a maximal *sq2DP* in T centered at position (C_i, C_j). Suppose we are considering location (i, j) of T as a possible center for a palindrome, and (i, j) is contained in the bottom right triangle of the larger palindrome P that is centered at (C_i, C_j). Further assume that both the maximal palindrome centered at (i, j) and at the mirror position of (i, j) over the anti-diagonal of P are completely contained within P. We can conclude the following:

▶ Observation 1. The maximal palindrome centered at a location (i, j) is identical to the maximal palindrome centered at the mirror position of (i, j) over the anti-diagonal of P.

The observation follows directly from the symmetry that the palindrome has over its anti-diagonal. As in 1D, the smaller palindromes contained in the upper triangle are mirrored exactly in the lower triangle. Note that this is true whether we use either diagonal, but our algorithm uses only the values over the anti-diagonal.

The idea of the algorithm is to use Observation 1 as follows. When searching for a palindrome centered at location (i, j) that is contained in a larger palindrome P, we first consider the value from the mirror image of position (i, j) over the anti-diagonal of P. If the maximal palindrome at the mirror image extends beyond the left boundary of P, then we take the minimum of the value at the mirror image and the boundary of P. Following this initial setting, we use the naive method to check whether it is possible to extend the palindrome centered at (i, j) beyond the right boundary of the containing palindrome. This mimics the algorithm of Manacher in two dimensions, but of course additional techniques are needed to render the algorithm linear time.

2.1 Preprocessing Stage

The text T is preprocessed by constructing a generalized suffix tree (GST) for the columns of T, from bottom to top and from top to bottom, and for the rows of T, from left to right and right to left. Then, it is preprocessed to allow $O(1)$-time longest common prefix (LCP) queries.

We define forward subcolumns and subrows (resp.) beginning at any location (i, j) as: $c(i, j) = T[i, j] \dots T[n, j]$, $r(i, j) = T[i, j] \dots T[i, n]$. Similarly, the reverse subcolumns and subrows are denoted by: $c'(i, j) = T[i, j] \dots T[1, j]$, $r'(i, j) = T[i, j] \dots T[i, 1]$.

Using these subcolumns and subrows, we can define four directions of L's cornered at a particular location (i, j), as subcolumn-subrow pairs[2].

1. A "backwards L," denoted $\text{⅃}_{i,j} = \langle c'(i, j), r'(i, j)\rangle$, consists of a pair of T's reverse subcolumn and reverse subrow.
2. An "upside down L," denoted $\Gamma_{i,j} = \langle c(i, j), r(i, j)\rangle$ consists of a pair of T's forward subcolumn and forward subrow.
3. An L, denoted $\text{L}_{i,j} = \langle c'(i, j), r(i, j)\rangle$, consists of a pair of T's reverse subcolumn and forward subrow.
4. An "upside down backwards L," denoted $\text{⅂}_{i,j} = \langle c(i, j), r'(i, j)\rangle$, consists of a pair of T's forward subcolumn and reverse subrow.

We also define constant time symmetry checking between Γ and ⅃. This can be done by taking the minimum value of the LCP of the corresponding sides of the L's when reflected over the anti-diagonal. Specifically, LCP $(\Gamma_{i,j}, \text{⅃}_{p,q}) = min(LCP(c'(p, q), r(i, j)), LCP(c(i, j), r'(p, q)))$. Similarly, in the other direction, the longest symmetric prefix between L and ⅂, reflected over the main diagonal, can be found in constant time.

2.2 Scanning Stage

In the scanning stage of the algorithm, we define a set of forward diagonals in the text, parallel to the main diagonal, $d = -(n - 1)$ to $(n - 1)$. This is similar to the method used by Amir and Farach [1] for multiple pattern matching of square patterns. We number each

[2] These L's are similar to the L's defined by Amir and Farach in [1]; the L-suffix tree of Giancarlo [12] uses a similar concept.

forward diagonal $d = i - j$, the difference between its row and column coordinates. Note that $d = 0$ is the main diagonal, $d > 0$ are the diagonals below the main diagonal, and $d < 0$ are the diagonals above the main diagonal. Each diagonal contains $n - |d|$ positions, where $|d|$ represents the absolute value of d.

Since the same procedure is performed on each forward diagonal, we describe the algorithm for a single forward diagonal d. The goal of the algorithm is to fill d's integer array $pals$ which corresponds to the $n-|d|$ positions on diagonal d in T. Each element in $pals$ will contain a value representing the maximal *sq2DP* centered at the corresponding position in T. Value v indicates that it consists of the position itself, plus v in the four directions (up, down, left, and right) – i.e., a *sq2DP* of size $(v * 2 - 1)$, with this position as its center.

We explain how Algorithm 1 works on diagonal $d \geq 0$. For $d < 0$, the same algorithm works with minor modifications to indices. The variable $maxCenter$ is the center of the *sq2DP* that has extended the farthest; $maxCenter + pals[maxCenter]$ is the rightmost (and lowest) position it reaches. j is a pointer that moves along the positions on the diagonal one at a time, and at each position we determine the size of the maximal *sq2DP* centered at the position pointed to by j.

For each j, the value in $pals[j]$ is set in a way similar to Manacher [21]: if the position is past $maxCenter + pals[maxCenter]$, then it has never been seen yet, and therefore its value in $pals$ is initialized to 1. If the position is before $maxCenter + pals[maxCenter]$, then it is known to be part of a palindrome, and therefore its value in $pals$ is initialized to the value of its mirror image over $maxCenter$; but if that value, when added to j, would extend beyond $maxCenter + pals[maxCenter]$, then the value is reduced so that it doesn't extend. Following the initial setting of the value in $pals[j]$, a while loop continually performs constant-time symmetry checking between the L's of different orientation to check how far the current palindrome extends.

One such square is demonstrated in Figure 2. Diagonal $d > 0$ is depicted and location j on diagonal d is depicted as the large dot. The LCP queries start one beyond $j + pals[j]$ and $j - pals[j]$: one involves ⅃ (straight vertical and horizontal lines) with Γ (dashed vertical and horizontal lines), and the other query involves L (dashed vertical and straight horizontal lines) with ⅂ (straight vertical and dashed horizontal lines).

Although both reflectional symmetries must be checked individually, it is not necessary to explicitly check the 180° rotation, since it is implied by transitivity from the reflectional symmetries. Specifically, location (i, j) must match its symmetric location over the main diagonal, which is (j, i). By the anti-diagonal symmetry, $T[j, i] = T[n - i, n - j]$ which is exactly the location symmetric to (i, j) by the 180° rotation.

Note that the algorithm works with *sq2DP*s of odd × odd dimensions; for even × even ones, include the following modifications: before the preprocessing stage, add a row to the top and the bottom of T, plus a row between every two rows. Also add a column on the left and the right of T, plus a column between every two columns. The added rows and columns are filled with a character that does not appear in T; and T of size $n \times n$ is now of size $(2n + 1) \times (2n + 1)$. When the scanning stage outputs a *sq2DP* of size $(2v + 1) \times (2v + 1)$, where v is even, that is indicative of a *sq2DP* of size $v \times v$, once the added rows and columns are removed.

2.3 Example

Using Table 2, we will demonstrate the scanning stage with an example, at the point where $d = 0$ and $j = 6$ (for position $T[6, 6]$; it is underlined). This position is contained in a palindrome, as $maxCenter + pals[maxCenter] = 5 + 5 = 10$ extends beyond it. Therefore,

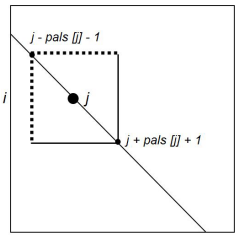


Figure 2 LCP queries on position j (large dot) of diagonal $d > 0$. One query involves a backwards L (straight vertical and horizontal lines) with an upside down L (dashed vertical and horizontal lines), and the other involves an L (dashed vertical and straight horizontal lines) with an upside down backwards L (straight vertical and dashed horizontal lines).

Algorithm 1: Algorithm for *sq2DP*.

input : GST of the columns and rows of T in forward and reverse order, diagonal d

output: diagonal d's integer array $pals$, of size $n - |d|$, containing the values of the maximal *sq2DP* centered at the corresponding positions in T

$maxCenter = 1$
$pals[1] = 1$
for $j = 2$ *to* $n - |d|$ **do** //for positions j on diagonal d
  $i = d + j$ //jth position on diagonal d is at $T[d + j, j]$ (if $d \geq 0$)
  if $maxCenter + pals[maxCenter] \leq j$ `/* position not known to be part of palindrome */`
  then
    $pals[j] = 1$
  else
    $pals[j] = \min\{pals[2 * maxCenter - j], maxCenter + pals[maxCenter] - j\}$
  while $(j + pals[j] < n)$ *and* $(j - pals[j] > 1)$ *and* `/* in bounds */`
  `/* The following two LCP queries check each of the diagonal symmetries, verifying whether the current palindrome can be enlarged by one layer all around. */`
  $LCP\ (\Gamma_{i-pals[j]-1, j-pals[j]-1}, \lrcorner_{i+pals[j]+1, j+pals[j]+1}) \geq 2 \times pals[j]$ *and*
  $LCP\ (\mathrm{L}_{i+pals[j]+1, j-pals[j]-1}, \urcorner_{i-pals[j]-1, j+pals[j]+1}) \geq 2 \times pals[j]$
  do
    $pals[j]{+}{+}$
  end
  if $j + pals[j] > maxCenter + pals[maxCenter]$ **then**
    $maxCenter = j$
end

Table 2 Text T (left) and $d = 0$'s *pals* array (right), at the point where the algorithm will calculate the value for $T[6,6]$ in $pals[6]$.

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

index	1	2	3	4	5	6	7	8	9	10
value	1	1	3	1	5	?	?			

its value in *pals* is that of its mirror image over *maxCenter*: 1. The two LCP queries indicate no extensions. Then $j = 7$, and that refers to $T[7,7]$ (underlined). Its value in *pals* is that of its mirror image over *maxCenter* – $T[3,3]$'s value of 3. LCP queries are performed, in an effort for a larger *sq2DP*, and they start with a square of size 7×7 (as sizes 3×3 and 5×5 are already known to be part of the *sq2DP*). They do indicate a *sq2DP* of size 7×7, but then the algorithm cannot continue as it would go out of bounds. Thus, *maxCenter* is set to point to this position and $pals[7]$ is set to 4. Then j is 8, and the algorithm will calculate the value of $pals[8]$ for $T[8,8]$.

2.4 Runtime

▶ **Theorem 3.** *The time complexity for finding all maximal* sq2DP *in a text of size* $n \times n$ *is* $O(n^2)$.

Proof. The runtime of the preprocessing stage is as follows: the construction of the GST is in time linear to the size of T [8]. Then it takes $O(n)$-time to preprocess to allow for constant-time LCP queries [13].

The scanning stage runs in $O(n^2)$ time. This is because Algorithm 1 is run on each of the $2n - 1$ diagonals. There are $O(n)$ positions j per diagonal (as seen by the number of iterations of the **for** loop in Algorithm 1). As in Manacher's algorithm, the initial value in *pals* is copied from the mirror image around *maxCenter*, and therefore each matching L is compared exactly once. Each mismatching L can be charged to the center for which it mismatched since each center encounters at most one mismatch. ◀

3 Rectangle 2D Palindrome

Working with *rect2DP*s is different than working with *sq2DP*s, as the mirror image property of Observation 1 is unique to squares and does not hold for general rectangles. Therefore, we use a different approach to finding them. The input is a 2D text T over a bounded alphabet Σ, with n_1 rows and n_2 columns. We assume $n_1 \geq n_2$; otherwise, it is possible to first rotate T by 90°. The preprocessing and scanning stages are described in this section, and all maximal *rect2DP* in T are reported.

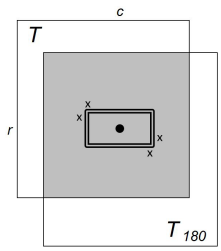


Figure 3 Placing of T_{180} on top of T, with $T[r, c]$ (represented as the dot) as the anchor. Some mismatches are demarcated with x's, and a *rect2DP* (enclosed in double lines) is placed within their bounds.

3.1 Preprocessing Stage

In the preprocessing stage, we construct a GST for the columns in T, both from top to bottom and from bottom to top. Then the GST is preprocessed to allow for constant-time LCP queries.

3.2 Scanning Stage

The scanning stage is run on each position of T. We describe the algorithm for a given position (r, c). The scanning stage outputs integer tuples whose values represent the height and width of maximal *rect2DP*(s) centered at position $T[r, c]$.

The underlying idea is visually depicted in Figure 3. Place T_{180}, which is T rotated by 180°, on top of T, with $T[r, c]$ as the *anchor* (that is, $T[r, c]$ must be placed on top of itself). Let T_{ov} be the overlapping region, which is shaded. For each column in T_{ov}, find the first mismatch between T and T_{180} that is above row r in T, and the first mismatch that is below row r of T. In Figure 3, some mismatches are demarcated with x's. Then, for each width possible, attempt to place a *rect2DP* whose center is position $T[r, c]$ and which is bounded on top and on bottom by mismatches. In Figure 3, such a *rect2DP* (enclosed with double lines) is drawn.

The idea is implemented in Algorithm 2 as follows: create T_{ov} as a subtext of T. It has $T[r, c]$ as its center, and it has the coordinates

top left: $T[r - min_r, c - min_c]$ top right: $T[r - min_r, c + min_c]$
bottom left: $T[r + min_r, c - min_c]$ bottom right: $T[r + min_r, c + min_c]$

where $min_r = \min(r - 1, n_1 - r)$ and $min_c = \min(c - 1, n_2 - c)$. Thus, T_{ov} has $min_r * 2 + 1$ rows and $min_c * 2 + 1$ columns. Note that since position $T[r, c]$ is the center of T_{ov}, it is at position $T_{ov}[min_r + 1, min_c + 1]$.

Then, finding mismatches between T and T_{180} is performed as constant-time LCP queries on the GST of the columns of T. Specifically, let $0 \leq k \leq min_c$. Every query involves row r; it is between the column that is k to the left of $T[r, c]$, from row r and above, with the column that is k to the right of $T[r, c]$, from row r and below. The results are stored in the *colLcp* array.

Finally, for every width, beginning with the widest possible, perform a range minimum query (RMQ) on *colLcp*. The resulting value bounds a rectangle on top and on bottom.

Algorithm 2: Algorithm for *rect2DP*.

input : T, GST of T's columns in forward and reverse order, r, c

output: integer tuples whose values represent the maximal *rect2DP*(s) centered at position $T[r, c]$

```
1  for k = 1 to (min_c * 2 + 1) do //for columns k
2      colLcp[k] = LCP(T[r, c − min_c + k], ..., T[1, c − min_c + k];
3                      T[r, c + min_c − k], ..., T[n_1, c + min_c − k])
4  end
5  maxHeight = 0
6  for w = min_c ... 0 do //for widths w, in decreasing order
7      height = RMQ (colLcp, (min_c + 1) − w, (min_c + 1) + w)
8      if height > maxHeight    /* if height ≤ maxHeight, then there is no rect2DP; or
       there is, but it's not maximal */
9      then
10         maxheight = height
11         output ⟨(height × 2 − 1), (w × 2 + 1)⟩
12 end
```

Table 3 The algorithm is at the point of locating the *rect2DP*s for position $T[3, 5]$ (underlined). The T_{ov} subtext (bold) is centered at that position. The *colLcp* array is also shown.

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

	1	2	3	4	5	6	7
colLcp	0	1	3	3	3	3	0

If the height is less than or equal to a previously found height then the rectangle is not a *rect2DP* (as it is not maximal). Otherwise, the algorithm outputs an integer tuple – height and width – representing this maximal *rect2DP*.

The algorithm above works with *rect2DP*s of odd × odd dimensions. For the case where one or both of the dimensions is even, similar modifications can be done to the text as provided in the *sq2DP* case. Alternatively, each possible center, including in between rows and columns, can be considered.

3.3 Example

In Table 3, T has $n_1 = 6$ rows and $n_2 = 8$ columns. We will demonstrate the scanning stage, at the point of the algorithm where r = 3 and c = 5 (position $T[3, 5]$, which is underlined). That position is the center of T_{ov} (which is bold), by having 5 rows (since $min_r = \min(3-1, 6-3)*2+1 = 5$) and 7 columns (since $min_c = \min(5-1, 8-5)*2+1 = 7$).

In particular, they are T's rows 1-5 and columns 2-8.

The *colLcp* array is shown. In detail: *colLcp*[1] is the result of the LCP query between *dee* and *bee* (which is 0), *colLcp*[2] is from the LCP query between *cde* and *cee* (which is 1), *colLcp*[3] is from the LCP query between *cde* and *cde* (which is 3), and so on.

Then, we set $maxHeight$ to 0. We will look for *rect2DP*s in w widths, in decreasing order. When $w = 3$, we are looking for a *rect2DP* that is centered at this position and is of width 7. No such *rect2DP* exists, and this is found by the algorithm ($height = 0$; since $height \not> maxHeight$, there is no output). When $w = 2$, $height = 1$. $1 > maxHeight$, and so this *rect2DP* is maximal: $maxHeight$ is set to 1, and $\langle(1 \times 2 - 1), (2 \times 2 + 1)\rangle = \langle 1, 5\rangle$ is outputted. It refers to the *rect2DP* of size 1×5: *ccacc*. When $w = 1$, we are looking for a *rect2DP* that is centered at this position and is of width 3. $height = 3$, and $3 > maxHeight$, which means that there is such a *rect2DP*. It is of size 5×3 – from $T[1, 4]$ through $T[5, 6]$. Lastly, when $w = 0$, $height = 3$. Because $3 \not> 3$ there is no output. This correlates, as the *rect2DP* of size 3×3 isn't maximal.

3.4 Runtime

▶ **Lemma 4.** *The time complexity for finding all maximal* rect2DP *in a text of size $n_1 \times n_2$ (where $n_1 \geq n_2$) is $O(n_1 n_2^2)$.*

Proof. The runtime of the preprocessing stage is as follows: the construction of the GST is in time linear to the size of T [8]. Then it takes $O(n)$-time to preprocess to allow for constant-time LCP queries [13].

The scanning stage takes $O(n_1 n_2^2)$-time. This is because there are $n_1 \times n_2$ positions, and each takes $O(n_2)$ time for each **for** loop in Algorithm 2 (they run $O(min_c)$ times and $min_c < n_2$). Note that, following linear time preprocessing, a RMQ takes $O(1)$-time [9]. ◀

▶ **Lemma 5.** *A text T of size $n_1 \times n_2$ can have $O(n_1 n_2^2)$ maximal* rect2DP*.*

Proof. We prove by providing one such example. See Table 4 for an $n \times n$ text that has $O(n^3)$ maximal *rect2DP*. It contains a diamond composed of 0's, and the rest of the text has *'s which indicate unique, unused characters. On the right is a partial table of counts of how many *rect2DP*s are centered at the corresponding text positions. The other three quadrants of the diamond (whose counts are not shown) are reflections and have the same values. Beginning at the center ($T[\lceil n/2 \rceil, \lceil n/2 \rceil]$) and moving outward, an element's count is one less than that of its neighbor. Let i represent a row and j a column; summing the top left quadrant (in this case, from [1, 1] through [7, 6]) of the counts table is: $\sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{j=1}^{i-1} j = \sum_{i=1}^{\lfloor n/2 \rfloor} \frac{i(i+1)}{2} = O(\sum_{i=1}^{n} i^2)$. Since the sum of the squares of 1 to n is $(n)(n+1)(2n+1)/6 = O(n^3)$, this $n \times n$ input text has $O(n^3)$ maximal *rect2DP*. ◀

▶ **Theorem 6.** *Algorithm 2 has worst case running time proportional to the worst case output size.*

Proof. Combining Lemmas 4 and 5 results in the proof. ◀

4 Conclusion

In this paper, we discussed two types of 2D palindromes and presented efficient algorithms to find them. By unlocking the world of 2D palindromes, we released many research opportunities. Essentially all of the variations of the 1D palindrome problem can now be applied

Table 4 Shown on the left is a text that contains a cubic number of maximal *rect2DP*. The *'s indicate unique, unused characters. On the right is a partial table with counts, which are the exact number of *rect2DP* that are centered at the corresponding text positions.

	1	2	3	4	5	6	7	8	9	0	1	2	3
1	*	*	*	*	*	*	0	*	*	*	*	*	*
2	*	*	*	*	*	0	0	0	*	*	*	*	*
3	*	*	*	*	0	0	0	0	0	*	*	*	*
4	*	*	*	0	0	0	0	0	0	0	*	*	*
5	*	*	0	0	0	0	0	0	0	0	0	*	*
6	*	0	0	0	0	0	0	0	0	0	0	0	*
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	*	0	0	0	0	0	0	0	0	0	0	0	*
9	*	*	0	0	0	0	0	0	0	0	0	*	*
0	*	*	*	0	0	0	0	0	0	0	*	*	*
1	*	*	*	*	0	0	0	0	0	*	*	*	*
2	*	*	*	*	*	0	0	0	*	*	*	*	*
3	*	*	*	*	*	*	0	*	*	*	*	*	*

	1	2	3	4	5	6	7	8	9	0	1	2	3
1							1						
2						1	2						
3					1	2	3						
4				1	2	3	4						
5			1	2	3	4	5						
6		1	2	3	4	5	6						
7	1	2	3	4	5	6	7	6	5	4	3	2	1
8							6						
9							5						
0							4						
1							3						
2							2						
3							1						

to the 2D setting. First, we would like to look at how both types of 2D palindromes relate to palstars and gapped palindromes. Additionally, searching for approximate palindromes is something that would be interesting in 2D. Yet another extension is from [16] and [17], who study *pal-equivalence*. Two strings of the same length are pal-equivalent iff the length of the maximal palindrome at every center in the strings is equal.

Another angle for further research, in terms of *rect2DP*, is to reduce the runtime. Although we proved that the output size is potentially asymptotically larger than the input, an optimal algorithm would take time proportional to the actual number of non-trivial palindromes reported.

Finally, it would be interesting to define and study additional geometric shapes of 2D palindromes, e.g. triangular, circular and perhaps a hexagonal 2D palindrome.

References

1 Amihood Amir and Martin Farach. Two-dimensional Dictionary Matching. *Information Processing Letters*, 44(5):233–239, 1992. doi:10.1016/0020-0190(92)90206-B.

2 Amihood Amir and Benny Porat. Approximate On-line Palindrome Recognition, and Applications. In *number 8486 in Lecture Notes in Computer Science*, pages 21–29. Springer International Publishing, 2014. doi:10.1007/978-3-319-07566-2_3.

3 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Optimal parallel algorithms for periods, palindromes and squares. In *Automata, Languages and Programming*, number 623 in Lecture Notes in Computer Science, pages 296–307. Springer Berlin Heidelberg, 1992.

4 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel Detection of all Palindromes in a String. *Comput. Sci*, pages 497–506, 1994.

5 Valérie Berthé and Laurent Vuillon. Palindromes and two-dimensional sturmian sequences. *Journal of Automata, Languages and Combinatorics*, 6(2):121–138, 2001.

6 Kuan-Yu Chen, Ping-Hui Hsu, and Kun-Mao Chao. Efficient retrieval of approximate palindromes in a run-length encoded string. *Theor. Comput. Sci.*, 432:28–37, 2012.

7 F.G.B. De Natale, Daniele D. Giusto, and Fabrizio Maccioni. A symmetry-based approach to facial features extraction. In *1997 13th International Conference on Digital Signal Processing Proceedings*, volume 2, pages 521–525, 1997. doi:10.1109/ICDSP.1997.628390.

8 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS'97, pages 137–143. IEEE Computer Society, 1997.

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 fun-with words. History of Palindromes, 1999. URL: http://fun-with-words.com/palin_history.html.

11 Zvi Galil and Joel Seiferas. A Linear-Time On-Line Recognition Algorithm for "Palstar". *Journal of the ACM (JACM)*, 25(1):102–111, 1978. doi:10.1145/322047.322056.

12 Raffaele Giancarlo. A Generalization of the Suffix Tree to Square Matrices, with Applications. *SIAM Journal on Computing*, 24(3):520–562, 1995. doi:10.1137/S0097539792231982.

13 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2), 1984.

14 Amit Hooda, Michael M. Bronstein, Alexander M. Bronstein, and Radu P. Horaud. Shape Palindromes: Analysis of Intrinsic Symmetries in 2d Articulated Shapes. In *Lecture Notes in Computer Science*, volume 6667, pages 665–676, Israel, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-24785-9_56.

15 Ping-Hui Hsu, Kuan-Yu Chen, and Kun-Mao Chao. Finding All Approximate Gapped Palindromes. In *Algorithms and Computation*, number 5878 in Lecture Notes in Computer Science, pages 1084–1093. Springer Berlin Heidelberg, 2009.

16 Tomohiro I, Shunsuke Inenaga, and Masayuki Takeda. Palindrome Pattern Matching. In *Lecture Notes in Computer Science*, pages 232–245. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-21458-5_21.

17 Hwee Kim and Yo-Sub Han. Online Multiple Palindrome Pattern Matching. In *String Processing and Information Retrieval*, pages 173–178. Springer International Publishing, 2013. doi:10.1007/978-3-319-11918-2_17.

18 Nahum Kiryati and Yossi Gofman. Detecting Symmetry in Grey Level Images: The Global Optimization Approach. *International Journal of Computer Vision*, 29(1):29–45, 1998. doi:10.1023/A:1008034529558.

19 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.

20 Roman Kolpakov and Gregory Kucherov. Searching for Gapped Palindromes. In *Combinatorial Pattern Matching*, number 5029 in Lecture Notes in Computer Science, pages 18–30. Springer Berlin Heidelberg, 2008.

21 Glenn 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.

22 Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009. doi:10.1016/j.tcs.2008.12.016.

23 Wikipedia.org. Palindrome, 2015. URL: http://en.wikipedia.org/wiki/Palindrome.

24 Wikipedia.org. Sator Square, 2015. URL: http://en.wikipedia.org/wiki/Sator_Square.

Boxed Permutation Pattern Matching

Mika Amit[1], Philip Bille*[2], Patrick Hagge Cording[3], Inge Li Gørtz†[4], and Hjalte Wedel Vildhøj[5]

1 **University of Haifa, Department of Computer Science**
2 **Technical University of Denmark, DTU Compute**
3 **Technical University of Denmark, DTU Compute**
4 **Technical University of Denmark, DTU Compute**
5 **Technical University of Denmark, DTU Compute**

Abstract

Given permutations T and P of length n and m, respectively, the Permutation Pattern Matching problem asks to find all m-length subsequences of T that are order-isomorphic to P. This problem has a wide range of applications but is known to be NP-hard. In this paper, we study the special case, where the goal is to only find the *boxed subsequences* of T that are order-isomorphic to P. This problem was introduced by Bruner and Lackner who showed that it can be solved in $O(n^3)$ time. Cho et al. [CPM 2015] gave an $O(n^2 m)$ time algorithm and improved it to $O(n^2 \log m)$. In this paper we present a solution that uses only $O(n^2)$ time. In general, there are instances where the output size is $\Omega(n^2)$ and hence our bound is optimal. To achieve our results, we introduce several new ideas including a novel reduction to 2D offline dominance counting. Our algorithm is surprisingly simple and straightforward to implement.

1998 ACM Subject Classification F.2.2 Computations and discrete structures, Geometrical problems and computations, Pattern matching. G.2.1 Combinatorial algorithms, Permutations and combinations

Keywords and phrases Permutation, Subsequence, Pattern Matching, Order Preserving, Boxed Mesh Pattern

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.20

1 Introduction

Consider a permutation $T = (t_1, t_2, \ldots, t_n)$ represented in the plane as the set of points $\{(1, t_1), (2, t_2), \ldots, (n, t_n)\}$. An axis-aligned *box* $B = (x_{\min}, x_{\max}, y_{\min}, y_{\max})$ that contains $|B| = k$ points induces a permutation $\sigma(B)$ of the integers $1, \ldots, k$. For example, the box shown in Figure 1 induces the permutation $\sigma(B) = (1, 4, 3, 2)$. Given T and a permutation $P = (p_1, p_2, \ldots, p_m)$ (the pattern), the *boxed permutation pattern matching problem* is to output all boxes where $\sigma(B) = P$. If two boxes contain the same set of points, we consider them the same.

We view boxed permutation pattern matching as a natural 2D computational geometry problem, but it can also be seen and motivated as a generalization of *order-preserving pattern matching* (also known as *consecutive permutation pattern matching*). In this one-dimensional string matching problem the goal is to output all substrings of T that are

* Supported by the Danish Research Council (DFF 1323-00178) and the Danish Research Council under the Sapere Aude Program (DFF 4005-00267)
† Supported by the Danish Research Council (DFF 1323-00178) and the Danish Research Council under the Sapere Aude Program (DFF 4005-00267)

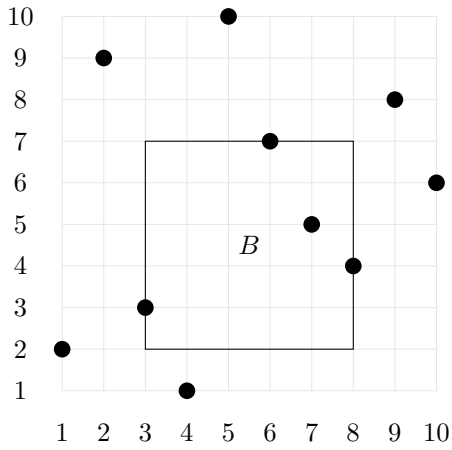


Figure 1 A permutation $T = (2, 9, 3, 1, 10, 7, 5, 4, 8, 6)$ and a box $B = (3, 8, 2, 7)$ with $\sigma(B) = (1, 4, 3, 2)$.

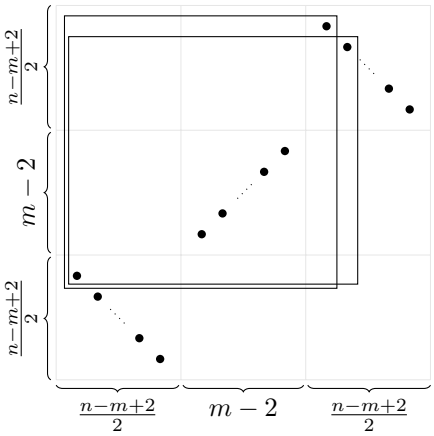


Figure 2 A permutation T with many occurrences of the pattern $P = (1, 2, \ldots, m)$, $2 \leq m \leq n$. Precisely, $(n - m + 2)^2/4$ boxes satisfy $\sigma(B) = P$, the figure shows two of them.

order-isomorphic to P. Order-preserving pattern matching has recently received a lot of attention (see e.g., [9,10,12,13,15,17,18,20]) as it is a natural generalization of classic exact string matching, and since it can be used to search for trends in time series such as stock prices, music or weather data, etc.

Boxed permutation pattern matching solves order-preserving pattern matching if we only output the boxes of the form $B = (x_{\min}, x_{\min} + m - 1, -\infty, \infty)$ where $\sigma(B) = P$. However, contrary to order-preserving pattern matching, which can be solved in $\tilde{O}(n)$ time by KMP-like algorithms [18,20], boxed permutation pattern matching requires $\Omega(n^2)$ time in general, as there are instances with $\Omega(n^2)$ occurrences of P (see e.g. Figure 2).

In this paper we present the first algorithm that solves boxed permutation pattern matching in optimal time, i.e., $O(n^2)$.

1.1 Previous and Related Work

Boxed permutation pattern matching was introduced by Bruner and Lackner [7] under the name *boxed-mesh permutation pattern matching*. They gave a simple $O(n^3)$ time algorithm. Recently, Cho et al. [11] presented a faster $O(n^2m)$-time algorithm and showed how to improve it to $O(n^2 \log m)$ time. Bruner and Lackner, as well as Cho et al., defined the problem in terms of subsequences, but we note that our geometric definition of the problem is equivalent.

Boxed permutation pattern matching is one of a few special cases of *permutation pattern matching* that can be solved in polynomial time. This problem, which is known to be NP-hard [4], asks to output all subsequences of T that are order-isomorphic to P.

Due to the many applications of permutation pattern matching a vast amount of research has studied its generalizations (e.g., vincular [3,19], bivincular [5] and mesh [6] patterns) and special cases (e.g. boxed mesh [2,11] and consecutive patterns [9,12,13,17,18,20] or patterns with certain combinatorial properties [1,16]). We refer the reader to Bruner and Lackner [7] for definitions and a comprehensive in-depth overview of previous work.

1.2 Our Result

We show the following result.

▶ **Theorem 1.** *Boxed permutation pattern matching can be solved in $O(n^2)$ time.*

As there are instances with $\Omega(n^2)$ outputs (see Figure 2), this time bound is optimal.

Our algorithm improves the $O(n^2 \log m)$-time algorithm by Cho et al. [11]. The $\log m$ factor in their time bound comes from their use of an order statistics tree with update time $O(\log m)$ to represent a box. We observe that plugging in the more efficient data structure by Pătraşcu and Thorup [21] immediately improves their time complexity to $O(n^2 \log m / \log \log n)$. However, as their solution inherently requires dynamic rank (or select) queries on the $\Omega(m)$ points inside a box, we cannot hope to further improve the time bound with this approach due to lower bounds on dynamic rank and select queries [14, 21].

We circumvent this apparent problem as follows: Instead of representing a box by the $\Omega(m)$ points it contains, we represent it in constant space by storing its four sides. Hence we can easily update the representation in constant time whenever we add a new point to the box. The challenge is to efficiently check *if* a point can be added or not. We show that implementing this check can be reduced to *2D offline dominance counting* on n subproblems of size $O(n)$. Solving these subproblems individually using the state-of-the-art $O(n\sqrt{\log n})$-time algorithm for 2D offline dominance problem by Chan and Pătraşcu [8] leads to an $O(n^2\sqrt{\log n})$-time solution for boxed permutation pattern matching. To get $O(n^2)$ time, we exploit the close relationship of the n subproblems, and show that it suffices to solve just a single of these subproblems.

Our final algorithm is surprisingly simple and straightforward to implement, as it only relies on a few lookup tables and uses no complicated supporting data structures.

2 Preliminaries

We start by giving some necessary definitions and combinatorial properties. Let P_k, $1 \leq k \leq m$, be the permutation of the integers $1, \ldots, k$ induced by the prefix $(p_1, p_2, \ldots, p_k)$ of P (see Figure 3). For a box $B = (i, j, y_{\min}, y_{\max})$ we define its size $|B|$ to be the number of points it contains. We use $\cdot$ to denote if one or more sides of B are unspecified, i.e., $(i, j, \cdot, \cdot)$

$$\begin{aligned} P = P_7 &= 4,2,1,6,3,5,7 \\ P_6 &= 4,2,1,6,3,5 \\ P_5 &= 4,2,1,5,3 \\ P_4 &= 3,2,1,4 \\ P_3 &= 3,2,1 \\ P_2 &= 2,1 \\ P_1 &= 1 \end{aligned}$$

Figure 3 The permutations induced by the prefixes of the pattern $P = (4,2,1,6,3,5,7)$.

denotes an arbitrary box with $x_{\min} = i$ and $x_{\max} = j$. We say that $B = (i, j, \cdot, \cdot)$ is *anchored* if it includes the point (i, t_i) as its left-most point. Furthermore, we say that B is a *prefix box* if B is anchored and $\sigma(B) = P_{|B|}$, in which case we also say that B *matches* the prefix of P of size $|B|$. We need the following lemma, which is similar to Lemma 2 in [11].

▶ **Lemma 2.** *For fixed integers $1 \le i \le j \le n$ and $1 \le k \le m$, there is at most one prefix box $B = (i, j, y_{\min}, y_{\max})$ of size k.*

Proof. Let i, j and k be fixed, and let s and l denote the number of elements of $\{p_2, \ldots, p_k\}$ that are smaller and larger than p_1, respectively. A prefix box $(i, j, \cdot, \cdot)$ of size k must contain the first s points below t_i and the first l points above t_i, and hence it is unique. ◀

The proof of the lemma uses the fact that for fixed i, j, k there is a unique candidate box $B_k = (i, j, \cdot, \cdot)$ that can match P_k. Figure 4 shows these candidate boxes for $k = 1, \ldots, m$ (and i and j fixed). Observe that only the prefixes P_1,P_2 and P_5 are matched, and that the boxes are nested, i.e., B_{k-1} is contained in B_k.

Given a prefix box $B = (i, j, \cdot, \cdot)$ its *preceding prefix box* is the largest prefix box $B' = (i, j, \cdot, \cdot)$ smaller than B. Observe that the preceding prefix box can be obtained by removing a certain number of points from above and a certain number of points from below, and note that these two numbers only depend on the size of B. Consequently, for a prefix box B of size $k = 2, \ldots, m$, we let (a_k, b_k) be the number of points that needs to be removed from B from above and below, respectively, to obtain the preceding prefix box of B. We will show how to compute these numbers later.

▶ **Example 3.** In Figure 4 there are three prefix boxes on $(i, j) = (2, 8)$: B_5, B_2 and B_1. The preceding prefix box of B_5 is B_2 and to obtain it we need to remove one point from above and two from below, so $(a_5, b_5) = (1, 2)$. Similarly, B_1 is the preceding prefix box of B_2 and to obtain it, we need to remove one point from below, so $(a_2, b_2) = (0, 1)$.

3 The Flattening Box Algorithm

At a high level the algorithm of Cho et al. [11] and our new algorithm can both be seen as implementations of an abstract algorithm, which we call *the flattening box algorithm*. The name comes from the fact that it examines boxes of decreasing height and increasing length. In this section we give a geometric exposition of this abstract algorithm, and in the next section we elaborate on how to implement the three primitives it needs.

Let $B_{\max}(i, j)$ be the largest prefix box $(i, j, \cdot, \cdot)$, e.g., in Figure 4, we have that $B_{\max}(2, 8) = B_5$. We also refer to $B_{\max}(i, j)$ as the *maximum prefix box* on (i, j). Note

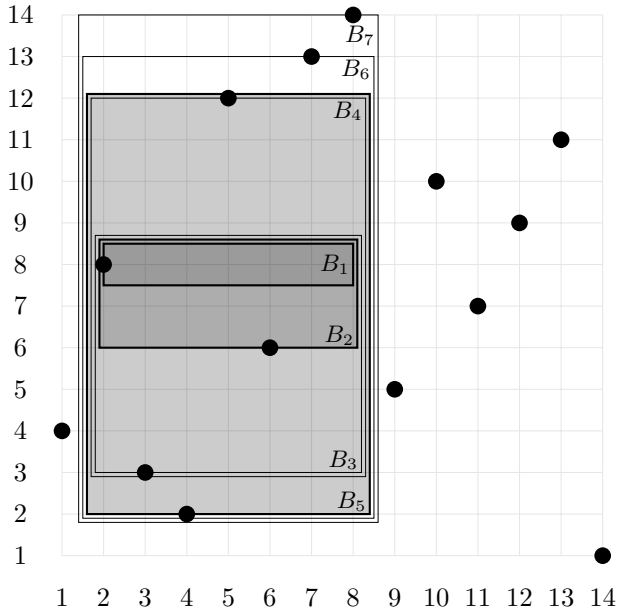


Figure 4 The unique boxes $(2, 8, \cdot, \cdot)$ that can match $P_1, \ldots, P_7$. In this specific case only the three bold boxes B_1, B_2, and B_5 are prefix boxes and match the corresponding prefix of P.

that by Lemma 2, $B_{\max}(i, j)$ is unique, and observe that if $|B_{\max}(i, j)| = m$ then it corresponds to an occurrence of P.

Recall that to solve the boxed permutation pattern matching problem, we need to find all boxes B s.t. $\sigma(B) = P$. The flattening box algorithm actually solves the slightly more general problem of computing $B_{\max}(i, j)$ for all $1 \leq i \leq j \leq n$. We do this in n iterations $(i = 1, \ldots, n)$, and in each iteration we compute $B_{\max}(i, j)$ for all $j = i, \ldots, n$. The main idea is to compute $B_{\max}(i, j)$ as a so-called *extension* of another prefix box $(i, j - 1, \cdot, \cdot)$.

3.1 Extensions of Prefix Boxes

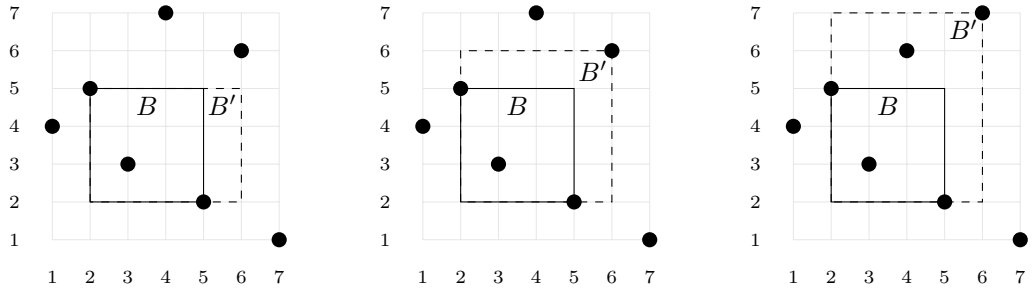
We say that a prefix box $B = (i, j, y_{\min}, y_{\max})$ has an *empty extension* $B' = (i, j + 1, y_{\min}, y_{\max})$ if B' is a prefix box also of size $|B|$. Note that this means B' contains exactly the same points as B. See Figure 5a for an example. Moreover, we say that B has an *increasing extension* $B' = (i, j + 1, \min(y_{\min}, t_{j+1}), \max(y_{\max}, t_{j+1}))$, if B' is a prefix box of size $|B| + 1$, i.e., $\sigma(B') = P_{|B|+1}$. Note, here B' is the box obtained by extending B to include the point $(j + 1, t_{j+1})$. See Figure 5b for an example.

The following lemma shows that we can compute the prefix boxes of the form $(i, j, \cdot, \cdot)$ as the extensions of the prefix boxes of the form $(i, j - 1, \cdot, \cdot)$.

▶ **Lemma 4.** *Let $1 \leq i < j \leq n$. Any prefix box $B = (i, j, \cdot, \cdot)$ is an extension of a prefix box $B' = (i, j - 1, \cdot, \cdot)$.*

Proof. Let $B = (i, j, y_{\min}, y_{\max})$, $j > i$, be a prefix box and consider the box $B' = (i, j - 1, y_{\min}, y_{\max})$. Clearly, B' is also a prefix box, and if $|B'| = |B| - 1$, B is an increasing extension of B', and otherwise $|B'| = |B|$ and B is an empty extension of B'. ◀

Let $B_{\text{ext}}(i, j) = (i, j, \cdot, \cdot)$ denote the largest prefix box that has an extension. We then have the following important corollary, which we will use for computing $B_{\max}(i, j)$.



(a) An empty extension **(b)** An increasing extension **(c)** Not an extension

Figure 5 Illustrating different extensions of a prefix box B of the pattern P shown in Figure 1. (a) B' is an empty extension of B, since $\sigma(B) = P_{|B|} = \sigma(B')$. (b) B' is an increasing extension of B, since $\sigma(B) = P_{|B|}$ and $\sigma(B') = P_{|B|+1}$. (c) B' is not an extension of B.

▶ **Corollary 5.** *$B_{\max}(i,j)$ is an (increasing or empty) extension of $B_{\text{ext}}(i,j-1)$.*

▶ **Example 6.** In Figure 4 there are three prefix boxes B_5, B_2 and B_1 on $(i,j) = (2,8)$. B_5 has no extension, B_2 has both an increasing and an empty extension, and B_1 has an empty extension. Consequently, $B_{\text{ext}}(i,j) = B_2$, and thus the largest prefix box $B_{\max}(i,j+1)$ is the increasing extension of B_2, i.e., $(2,9,5,8)$, matching the prefix P_3.

3.2 The Abstract Algorithm

Our goal is to compute $B_{\max}(i,j)$ assuming we have already computed $B_{\max}(i,j-1)$. According to Corollary 5, we need to first find $B_{\text{ext}}(i,j-1)$. Note that as shown by Example 6, $B_{\text{ext}}(i,j-1)$ is not necessarily equal to $B_{\max}(i,j-1)$. However, we can find $B_{\text{ext}}(i,j-1)$ as follows: Starting with $B_{\max}(i,j-1)$ (which we have computed), we check each of the prefix boxes $(i,j-1,\cdot,\cdot)$ in decreasing order of their size. The first one, which has an extension (increasing or empty) gives us $B_{\text{ext}}(i,j-1)$, and hence also $B_{\max}(i,j)$. Algorithm 1 shows this approach, assuming that we have available a function `precedingPrefixBox`, which takes a prefix box $B = (i,j-1,\cdot,\cdot)$ and returns the largest prefix box $(i,j-1,\cdot,\cdot)$ smaller than B.

4 Implementing the Algorithm

To implement the abstract algorithm we need to describe how to check if a prefix box B has an increasing/empty extension, and how to obtain the preceding prefix box of B. We describe how to do this in the following sections.

4.1 Checking If a Prefix Box Can Be Extended

We can easily check in constant time whether a given prefix box $B = (i,j,y_{\min},y_{\max})$ has an empty extension, since this is the case if and only if $t_{j+1} \notin [y_{\min},y_{\max}]$.

Hence in the remaining part of this section we focus on how to efficiently check whether B has an increasing extension, which is significantly more involved. We need the following definitions. For a permutation $Q = (q_1,\ldots,q_{|Q|})$, we define $D_Q(i,j) = |(i,j,1,q_j)|$, i.e., $D_Q(i,j)$ is the number of points (l,q_l) where $i \le l \le j$ and $q_l \le q_j$ (see Figure 6b). Moreover, for a box $B = (i,j,y_{\min},y_{\max})$, we define $E_Q(B) = |(i,j,1,y_{\min}-1)|$, i.e., $E_Q(B)$ is the number of points below B (see Figure 6a).

Input : Permutations $T = (t_1, \ldots, t_n)$ and $P = (p_1, \ldots, p_m)$
Output: All boxes B s.t. $\sigma(B) = P$

```
1  for i ← 1 to n do
2      B_max(i, i) ← (i, i, t_i, t_i)
3      /Users/mikaamit/Dropbox/Order Preserving/versions/CPM16/paper.tex
4      for j ← i + 1 to n do
5          B ← B_max(i, j − 1)
6          while B_max(i, j) = null do
7              if B has an increasing extension B′ then
8                  B_max(i, j) ← B′
9              else if B has an empty extension B′ then
10                 B_max(i, j) ← B′
11             else
12                 B ← precedingPrefixBox(B)
13             end
14         end
15         if |B_max(i, j)| = m then
16             Output B_max(i, j)                    /* Found an occurrence of P */
17         end
18     end
19 end
```

Algorithm 1: The Flattening Box Algorithm

The following lemma gives the property that we will use for checking if a prefix box B has an increasing extension in constant time.

▶ **Lemma 7.** *A prefix box $B = (i, j, y_{\min}, y_{\max})$ has an increasing extension if and only if $D_T(i, j+1) - E_T(B) = D_P(1, |B|+1)$.*

Proof. We need the following simple *append* operation on permutations. Let $Q = (q_1, \ldots, q_k)$ be a permutation of the integers $1, \ldots, k$. The permutation obtained by *appending* an integer $1 \leq r \leq k+1$ to Q is the permutation $Q \cdot r = (q'_1, \ldots, q'_k, r)$ of the integers $1, \ldots, k+1$, where for $1 \leq i \leq k$, $q'_i = q_i + 1$ if $q_i \geq r$ and $q'_i = q_i$ otherwise.

Now to prove the lemma, we start by observing that $P_{k+1} = P_k \cdot D_P(1, k+1)$ for $1 \leq k \leq m-1$. We need to show that the box $B' = (i, j+1, \min(t_{j+1}, y_{\min}), \max(t_{j+1}, y_{\max}))$ obtained by extending B to include the point $(j+1, t_{j+1})$, induces the permutation $P_{|B|+1}$ if and only if $D_T(i, j+1) - E_T(B) = D_P(1, |B|+1)$. We consider the two cases $|B'| = |B|+1$ and $|B'| > |B|+1$ separately.

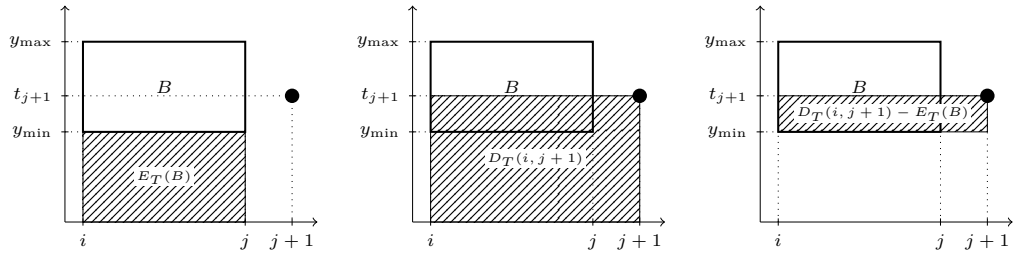
In the first case $|B'| = |B|+1$, which means B', in addition to the points in B, includes only the point $(j+1, t_{j+1})$. It is not hard to see that $D_T(i, j+1) - E_T(B)$ counts the number of points $(l, t_l) \in B$ s.t. $t_l \leq t_{j+1}$ (see Figure 6). Hence the induced permutation of B' is

$$\sigma(B') = \sigma(B) \cdot (D_T(i, j+1) - E_T(B)) \ .$$

At the same time we have that B has an increasing extension if and only if

$$\sigma(B') = P_{|B|+1} = P_{|B|} \cdot D_P(1, k+1) = \sigma(B) \cdot D_P(1, k+1) \ .$$

Combining the two equations yields the lemma.



(a) $E_T(B)$ **(b)** $D_T(i, j+1)$ **(c)** $D_T(i, j+1) - E_T(B)$

Figure 6 Illustrating the boxes that represent $E_T(B)$, $D_T(i, j+1)$, and $D_T(i, j+1) - E_T(B)$. The number of points in these boxes are used to decide if a prefix box B has an increasing extension.

In the other case $|B'| > |B| + 1$. This means that B' in addition to $(j+1, t_{j+1})$ also contains some other points (either above or below B). See Figure 5c for an example. Clearly $\sigma(B') \neq P_{|B|+1}$ and thus B has no increasing extension in this case. To show that $D_T(i, j+1) - E_T(B) \neq D_P(1, |B|+1)$, observe that if points above B were included then $D_T(i, j+1) - E_T(B) > |B| + 1$, and if points below B were included then $D_T(i, j+1) - E_T(B) \leq 0$. Since $1 \leq D_P(1, |B|+1) \leq |B|+1$, we have that $D_T(i, j+1) - E_T(B) \neq D_P(1, |B|+1)$. ◀

In the following we describe how to efficiently obtain the values of $D_T(i, j+1)$, $E_T(B)$ and $D_P(1, |B|+1)$.

4.1.1 The value $D_P(1, |B|+1)$

Prior to running the algorithm, we preprocess a table of size $O(m)$ that stores the value $D_P(1, k)$ for $k = 2, \ldots, m$. Assuming we maintain the size of the current prefix box B in the algorithm (which is straightforward), we can obtain $D_P(1, |B|+1)$ by a constant-time lookup. We describe how to compute the lookup table in $O(m\sqrt{\log m})$ time in Section 4.3.

4.1.2 The value $E_T(B)$

Recall that in the algorithm we consider the prefix boxes $(i, j, \cdot, \cdot)$ in decreasing order of their size until we find $B_{\text{ext}}(i, j)$. For each such prefix box B, we need the value of $E_T(B)$, i.e., the number of points below that prefix box. We maintain this number as we iterate j from i to n as follows.

Initially (for $j = i$) there are no points below the box, so $E_T(B) = 0$. The number only changes in the following two cases: If B does not have an extension, we consider the preceding prefix box of B, and hence $E_T(B)$ increases by b_k (the number of points that are removed from below). The other case in which $E_T(B)$ changes is when B is extended to $j+1$ as an empty extension and $t_{j+1} < y_{\min}$. In this case $E_T(B)$ increases by one.

4.1.3 The value $D_T(i, j+1)$

In the following assume that i is fixed, corresponding to a single iteration of the outer-most loop of the algorithm. As we iterate j from i to n, we need the value $D_T(i, j)$. We compute these values for $j = i, \ldots, n$ in the beginning of iteration i and store them in a table of size $O(n)$. Recall that $D_T(i, j)$ is the number of points with x value between i and j and y-value below t_j. Hence we can compute the value $D_T(i, j)$ from $D_T(i-1, j)$ as follows.

$$D_T(i, j) = \begin{cases} D_T(i-1, j) - 1 & \text{if } t_{i-1} < t_j \\ D_T(i-1, j) & \text{otherwise} \end{cases} \tag{1}$$

Note that computing the table only takes $O(n)$ time, assuming we have the table $D_T(i-1, j)$. Moreover, there is no need to store the old table, so we only need $O(n)$ space over all iterations of the algorithm. However, in the very first iteration ($i = 1$), we need the table $D_T(1, j)$, $j = 1, \ldots, n$. We compute this table in $O(n\sqrt{\log n})$ time in the preprocessing phase of the algorithm as explained in Section 4.3.

4.2 Computing the Preceding Prefix Box

Recall that given a prefix box $B = (i, j, y_{\min}, y_{\max})$ of size k, its preceding prefix box B' can be obtained by removing a_k points from above, and b_k points from below. We compute (a_k, b_k) for all k in the preprocessing phase as will be explained in Section 4.3.

To remove a_k points from above, we decrement $y_{\max}$ in steps of one and keep track of how many points we have excluded. Note that when decrementing $y_{\max}$ we exclude a point from B if and only if $i \leq T^{-1}(y_{\max}) \leq j$, where T^{-1} is the *inverse permuation* of T, i.e., $T^{-1}(t_i) = i$. We remove the b_k points from below by similarly incrementing $y_{\min}$ until b_k points have been excluded.

We compute T^{-1} in the preprocessing phase of the algorithm in $O(n)$ time.

4.3 Preprocessing the Lookup Tables

We now describe and analyze the necessary preprocessing of the text T and the pattern P.

4.3.1 Preprocessing of the Text

For the text, we need two $O(n)$-size tables, its inverse permutation T^{-1}, and the table for $D_T(1, j)$, $j = 1, \ldots, n$. The inverse permutation is easily computed in $O(n)$ time.

Recall that $D_T(1, j)$ is the number of points with strictly smaller coordinates than (j, t_j). The problem of computing this number for all n points is known as *2D offline dominance counting*. A point in the plane dominates another point if each one of its coordinates is strictly larger. In the *2D offline dominance counting* problem we are given a set of n points, and we want to count the number of other points that each point dominates. This problem is solved in $O(n\sqrt{\log n})$ time using the algorithm by Chan and Pătraşcu [8]. In fact, since we only need to compute this table for $i = 1$, we can afford to use the trivial $O(n^2)$-time algorithm as well.

4.3.2 Preprocessing of the Pattern

For the pattern we need two $O(m)$-size tables, the table for $D_P(1, k)$, $k = 1, \ldots, m$, and the table storing the (a_k, b_k) values for $k = 1, \ldots, m$. Computing the table for $D_P(1, k)$ can be done in $O(m\sqrt{\log m})$ time exactly as we did for the text.

We will compute the (a_k, b_k) values incrementally using an algorithm very similar to the flattening box algorithm. Recall that (a_k, b_k) denote the number of points that must be removed from a prefix box B of size k, from above and below, respectively, to obtain its preceding prefix box. Initially, we set $(a_1, b_1) = (0, 0)$. Let $T = P$, i.e., we treat the set of points $\{(1, p_1), \ldots, (m, p_m)\}$ as the text. Now consider a box $B_k = (1, k, 1, m)$. This box is clearly a prefix box, since it matches P_k, and hence it is also the maximum prefix box of the form $(1, k, \cdot, \cdot)$. Let B'_k denote the *second largest* prefix box on $(1, k, \cdot, \cdot)$, i.e., the preceding prefix box of B_k. The idea is to compute B'_k, for $k = 1, \ldots, m$. It is easy to see that this will give us the (a_k, b_k) values as the number of points above and below B'_k, respectively.

It follows from Lemma 2 that we can compute B'_k from B'_{k-1} as follows: Starting with B'_{k-1} (which we have computed), we check each of the prefix boxes $(1, k-1, \cdot, \cdot)$ in decreasing order of their size. The first one, which has an extension (increasing or empty) of size less than k, gives us B'_k. Note that as we have already computed $(a_{k'}, b_{k'})$ for $k' < k$, we can compute the preceding prefix box of B'_{k-1} (and any of its predecessors) by simply removing (a_{k-1}, b_{k-1}) points from above and below using exactly the same approach as in the flattening box algorithm (See Section 4.2). This requires that we also compute the inverse permutation of P in $O(m)$ time and space.

The time for computing all (a_k, b_k) values can be bounded by $O(m^2)$, for the same reason the flattening box algorithm runs in $O(n^2)$ time (See the next section).

5 Analysis

We now summarize the time analysis of our algorithm.

As already argued, we need $O(n\sqrt{\log n})$ time and $O(n)$ space for preprocessing the text, and preprocessing of the pattern takes $O(m^2)$ time and $O(m)$ space. To prove that the algorithm runs in $O(n^2)$ time, we show that a single iteration of the outer-most for-loop only takes $O(n)$ time. Checking if a prefix box can be extended only requires constant-time table lookups. To see that the total time spent computing preceding prefix boxes is $O(n)$, it suffices to note that once a point is excluded, it can never be included again. Hence the total time spent decrementing $y_{\max}$ and incrementing $y_{\min}$ is $O(n)$.

Consequently, the total time complexity is $O(n^2 + m^2) = O(n^2)$.

6 Open Problems

We have shown that boxed permutation pattern matching can be solved in $O(n^2)$ time, which is optimal. Our algorithm uses $O(n)$ space, which leaves open the problem of reducing the space to $O(m)$. The main challenge in doing this is the apparent absence of a suitable decomposition of a problem instance into $O(n^2/m^2)$ independent subproblems of size $O(m^2)$. We do believe that an $O(n^2)$ time and $O(m)$ space algorithm exists, but note that with our current techniques, it seems difficult to reduce the space to $O(m)$ without increasing the time to at least $O(n^2 \log\log n)$, for computing the rank of a point in a given box.

Another interesting direction for future work is the possibility of designing output-sensitive algorithms. That is, can an instance of boxed permutation pattern matching with occ occurrences of the pattern be solved in $O(n^{2-\epsilon} + occ)$ time, for some $\epsilon > 0$?

Finally, we note that indexing and approximate variants of boxed permutation pattern matching also have not been studied yet.

References

1 M. H. Albert, R. E. Aldred, M. D. Atkinson, and D. A. Holton. Algorithms for pattern involvement in permutations. In *Algorithms and Computation*, pages 355–367. Springer, 2001.

2 S. Avgustinovich, S. Kitaev, and A. Valyuzhenich. Avoidance of boxed mesh patterns on permutations. *Discrete App. Math.*, 161(1):43–51, 2013.

3 Eric Babson and Einar Steingrımsson. Generalized permutation patterns and a classification of the mahonian statistics. *Sém. Lothar. Combin*, 44(B44b):547–548, 2000.

4 P. Bose, J. F. Buss, and A. Lubiw. Pattern matching for permutations. *Inform. Process. Lett.*, 65(5):277–283, 1998.

5 M. Bousquet-Mélou, A. Claesson, M. Dukes, and S. Kitaev. (2+ 2)-free posets, ascent sequences and pattern avoiding permutations. *J. Comb. Theory A*, 117(7):884–909, 2010.

6 P. Brändén and A. Claesson. Mesh patterns and the expansion of permutation statistics as sums of permutation patterns. *Electron. J. Combin*, 18(2):P5, 2011.

7 M. L. Bruner and M. Lackner. The computational landscape of permutation patterns. *CoRR*, abs/1301.0340, 2013.

8 T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proc. 21st ACM-SIAM symposium on Discrete Algorithms*, pages 161–173. Society for Industrial and Applied Mathematics, 2010.

9 T. Chhabra and J. Tarhio. Order-preserving matching with filtration. In *Experimental Algorithms*, pages 307–314. Springer, 2014.

10 S. Cho, J. C. Na, K. Park, and J. S. Sim. A fast algorithm for order-preserving pattern matching. *Inform. Process. Lett.*, 115(2):397–402, 2015.

11 S. Cho, J. C. Na, and J. S. Sim. Improved algorithms for the boxed-mesh permutation pattern matching problem. In *Proc. 26th CPM*, pages 138–148. Springer, 2015.

12 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń. Order-preserving incomplete suffix trees and order-preserving indexes. In *Proc. 20th SPIRE*, pages 84–95. Springer, 2013.

13 S. Faro and M. O. Külekci. Efficient algorithms for the order preserving pattern matching problem. *CoRR*, abs/1501.04001, 2015.

14 Michael Fredman and Michael Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proc. 21st STOC*, pages 345–354. ACM, 1989.

15 P. Gawrychowski and P. Uznański. Order-preserving pattern matching with k mismatches. In *Proc. 25th CPM*, pages 130–139. Springer, 2014.

16 L. Ibarra. Finding pattern matchings for permutations. *Inform. Process. Lett.*, 61(6):293–295, 1997.

17 J. Kim, A. Amir, J. C. Na, K. Park, and J. S. Sim. On representations of ternary order relations in numeric strings. In *Proc. 2nd ICABD*, pages 46–52, 2014.

18 J. Kim, P. Eades, R. Fleischer, S. H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order-preserving matching. *Theoret. Comput. Sci.*, 525:68–79, 2014.

19 S. Kitaev. *Patterns in permutations and words*. Springer Science & Business Media, 2011.

20 M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A linear time algorithm for consecutive permutation pattern matching. *Inform. Process. Lett.*, 113(12):430–433, 2013.

21 M. Pătraşcu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th FOCS*, pages 166–175. IEEE, 2014.

Longest Common Substring with Approximately k Mismatches

Tatiana Starikovskaya

University of Bristol, Woodland Road, Bristol, UK, BS8 1UB,
tat.starikovskaya@gmail.com

Abstract

In the longest common substring problem we are given two strings of length n and must find a substring of maximal length that occurs in both strings. It is well-known that the problem can be solved in linear time, but the solution is not robust and can vary greatly when the input strings are changed even by one letter. To circumvent this, Leimester and Morgenstern introduced the problem of the longest common substring with k mismatches. Lately, this problem has received a lot of attention in the literature, and several algorithms have been suggested. The running time of these algorithms is $n^{2-o(1)}$, and unfortunately, conditional lower bounds have been shown which imply that there is little hope to improve this bound.

In this paper we study a different but closely related problem of the longest common substring with approximately k mismatches and use computational geometry techniques to show that it admits a randomised solution with strongly subquadratic running time.

1998 ACM Subject Classification F.2.2. Nonnumerical Algorithms and Problems

Keywords and phrases Randomised algorithms, string similarity measures, longest common substring, sketching, locality-sensitive hashing

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.21

1 Introduction

Understanding how similar two strings are and what they share in common is a central task in stringology, the significance of which is witnessed for example by the 50,000+ citations of the paper introducing BLAST [3], a heuristic algorithmic tool for comparing biological sequences. This task can be formalised in many different ways, from the longest common substring problem to the edit distance problem. The longest common substring problem can be solved in optimal linear time and space, while the best known algorithms for the edit distance problem require $n^{2-o(1)}$ time, which makes the longest common substring problem an attractive choice for many practical applications. On the other hand, the longest common substring problem is not robust and its solution can vary greatly when the input strings are changed even by one letter. To overcome this issue, recently there has been introduced a new problem called the longest common substring with k mismatches. In this paper we continue this line of research.

1.1 Related work

Let us start with a precise statement of the longest common substring problem.

▶ **Problem 1** (The longest common substring)**.** Given two strings T_1, T_2 of length n, find a substring of maximal length that occurs in T_1 and T_2 exactly.

The suffix tree of T_1 and T_2, a data structure containing all suffixes of T_1 and T_2, allows to solve this problem in linear time and space [32, 19, 21], which is optimal as any algorithm

needs $\Omega(n)$ time to read and $\Omega(n)$ space to store the strings. However, if we only account for "additional" space, the space the algorithm uses apart from the space required to store the input, then the suffix tree-based solution is not optimal and has been improved in a series of publications [6, 25, 31].

The major disadvantage of the longest common substring problem is that its solution is not robust. Consider, for example, two pairs of strings: a^n, $a^{n-1}b$ and $a^{(n-1)/2}ba^{(n-1)/2}$, $a^{n-1}b$. (Assume for simplicity that $n-1 \geq 2$ is even.) The longest common substring of the first pair of strings is twice as long as the longest common substring of the second pair of strings, although we changed only one letter. This makes the longest common substring unsuitable to be used as a measure of similarity of two strings: Intuitively, changing one letter must not change the measure of similarity much. To overcome this issue, it is natural to allow the substring to occur in T_1 and T_2 not exactly but with a small number of mismatches.

▶ **Problem 2** (The longest common substring with k mismatches)**.** Given two strings T_1, T_2 of length n and an integer k, find a substring of maximal length that occurs in T_1 and T_2 with at most k mismatches.

The problem can be solved in quadratic time and space by a dynamic programming algorithm, but there have been also shown more efficient solutions. The longest common substring with one mismatch problem was first considered in [7], where an $\mathcal{O}(n^2)$-time and $\mathcal{O}(n)$-space solution was given. This result was further improved by Flouri et al. who showed an $\mathcal{O}(n \log n)$-time and $\mathcal{O}(n)$ space solution to the problem [16]. For a general value of k, the problem was first considered by Leimeister and Morgenstern [28] who presented a greedy heuristic algorithm for the problem. Later Flouri et al. showed that the longest common substring with k mismatches admits a quadratic time and constant (additional) space algorithm [16]. Apart from that, Grabowski presented two output-dependent algorithms with running times $\mathcal{O}(n((k+1)(\ell_0+1))^k)$ and $\mathcal{O}(n^2\ell_k/k)$, where ℓ_0 is the length of the longest common substring of T_1 and T_2 and ℓ_k is the length of the longest common substring with k mismatches of T_1 and T_2 [18]. Finally, Aluru et al. gave an $\mathcal{O}(2^k n)$-space, $\mathcal{O}(n(2\log n)^{k+1})$-time algorithm [4]. Yet, the worst-case running time of all these algorithms is still quadratic. Very recently, Abboud et al. [1] applied the polynomial method to develop a $k^{1.5}n^2/2^{\Omega(\sqrt{\frac{\log n}{k}})}$-time randomised solution to the problem.

The best algorithms for the edit distance problem and its variations (we do not give their precise statements here as it is not essential for the paper) also have $n^{2-o(1)}$ running time [30, 12, 29], and these bounds are tight under the Strong Exponential Time Hypothesis (SETH) of Impagliazzo, Paturi and Zane: [8, 11]:

▶ **Hypothesis 1.** (SETH). For every $\delta > 0$ there exists an integer m such that SAT on m-CNF formulas on n variables cannot be solved in $m^{O(1)}2^{(1-\delta)n}$ time.

1.2 Our contribution

In this paper we introduce a new problem called the longest common substring with approximately k mismatches, inspired by the work of Andoni and Indyk [5].

▶ **Problem 3** (The longest common substring with approximately k mismatches)**.** Given two strings T_1, T_2 of length n, an integer k, and a constant $\varepsilon > 0$. If ℓ_k is the length of the longest common substring with k mismatches of T_1 and T_2, return a substring of length at least ℓ_k that occurs in T_1 and T_2 with at most $(1+\varepsilon)\cdot k$ mismatches.

In their work Andoni and Indyk used the technique of locality-sensitive hashing to develop a space-efficient randomised index for a variant of the approximate pattern matching problem.

We build up on their work with several new ideas in the construction and the analysis to develop a randomised subquadratic-time solution to Problem 3. Assume binary alphabet and let $0 < \varepsilon < 2$ be an arbitrary constant.

▶ **Theorem 1.** *The longest common substring with approximately k mismatches can be solved in $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ space and $\mathcal{O}(n^{1+1/(1+\varepsilon)}\log^2 n)$ time correctly with constant probability.*

If the alphabet is of constant size $\sigma > 2$, we can use a standard trick and encode T_1 and T_2 by replacing each letter a in them with a binary vector $0^{a-1}10^{\sigma-a}$. The Hamming distance (i.e. the number of mismatches) between any two substrings of T_1 and T_2 in the encoded form will be exactly twice as large as the Hamming distance between the original substrings, which allows to extend our solution naturally to this case as well at a cost of an additional constant factor.

We note that although the problem statement is not standard for stringology, it makes perfect sense from the practical point of view. Indeed, for most applications it is not important whether a returned substring occurs in T_1 and T_2 with for example 10 or $(1+\frac{1}{5})\cdot 10 = 12$ mismatches. The result is also important from the theoretical point of view as it improves our understanding of the big picture of string comparison.

2 Overview

In this section we give an overview of the main ideas needed to prove Theorem 1. The classic solution to the longest common substring problem is based on two observations. The first observation is that the longest common substring of T_1 and T_2 is in fact the longest common prefix of some suffix of T_1 and some suffix of T_2. The second observation is that the maximal length of the longest common prefix of a fixed suffix S of T_1 and suffixes of T_2 is reached on the two suffixes of T_2 that are closest to S in the lexicographic order. This suggests the following algorithm: First, we build a suffix tree of T_1 and T_2, which contains all suffixes of T_1 and T_2 and orders them lexicographically. Secondly, we compute the longest common prefix of each suffix of T_1 and the two suffixes of T_2 closest to S in the lexicographic order, one from the left and one from the right. The problem of computing the longest common prefix has been extensively studied in the literature and a number of very efficient deterministic and randomised solutions exist [9, 10, 14, 22, 20], for example, one can use a Lowest Common Ancestor (LCA) data structure, which can be constructed in linear time and space and maintains longest common prefix queries in $\mathcal{O}(1)$ time [14, 20].

Our solution to the longest common substring with approximately k mismatches problem is somewhat similar. We will consider $\theta(n^{1+1/(1+\varepsilon)}\log n)$ orderings on the suffixes of T_1 and T_2 and will show that with high probability the length of the longest common substring with approximately k mismatches is the answer to a longest common prefix with approximately k mismatches ($\mathrm{LCP}_{\tilde{k}}$) query for some pair of suffixes that are close to each other in one of the orderings. In an $\mathrm{LCP}_{\tilde{k}}$ query we are given two suffixes S_1, S_2 of T_1 and T_2 and must output any integer in the interval $[\ell_k, \ell_{(1+\varepsilon)\cdot k}]$, where ℓ_k and $\ell_{(1+\varepsilon)\cdot k}$ are the lengths of the longest common prefixes of S_1 and S_2 with k and $(1+\varepsilon)\cdot k$ mismatches respectively. Note that $\mathrm{LCP}_{\tilde{k}}$ queries can be answered deterministically in $\mathcal{O}(k)$ time using the kangaroo method [27, 17], but for the purposes of this paper we give a faster randomised solution.

▶ **Theorem 2.** *After $\mathcal{O}(n\log^3 n)$ time and $\mathcal{O}(n\log^2 n)$ space preprocessing of T_1 and T_2, an $\mathrm{LCP}_{\tilde{k}}$ query can be answered in $\mathcal{O}(\log^2 n)$ time. The answer is correct with probability at least $1-\frac{1}{n^2}$.*

The key idea is to compute sketches for all power-of-two length substrings of T_1 and T_2. The sketches will have logarithmic length (i.e., we will be able to compare them very fast) and the Hamming distance between them will be roughly equal to the Hamming distance between the original substrings. Once the sketches are computed, we can use a simple binary search to answer $\mathrm{LCP}_{\tilde{k}}$ queries in polylogarithmic time.

To define the orderings on suffixes of T_1 and T_2 we will use the locality-sensitive hashing technique, which was initially introduced for the needs of computational geometry [23]. In more detail, we will choose $\theta(n^{1+1/(1+\varepsilon)}\log n)$ hash functions, where each function can be considered as a projection of a string of length n onto a random subset of its positions. By choosing the size of the subset appropriately, we will be able to guarantee that the hash function is locality-sensitive: For any two strings at the Hamming distance at most k, the values of the hash functions on them will be equal with high probability, while the values of the hash functions on any pair of strings at the Hamming distance bigger than $(1+\varepsilon)\cdot k$ will be different with high probability. For each hash function we will sort the suffixes of T_1 and T_2 by the lexicographic order on their hash values. As a corollary of the locality-sensitive property, if two suffixes of T_1 and T_2 have a long common prefix with at most k mismatches, with high probability they will be close to each other in the ordering.

3 Proof of Theorem 2

In this section we show Theorem 2. During the preprocessing stage, we compute sketches [26] of all substrings of the strings T_1 and T_2 of lengths $\ell = 1, 2, \ldots, 2^{\lfloor \log n \rfloor}$, which can be defined in the following way. For a fixed ℓ choose $\lambda = 1.5 \ln n/\gamma^2$ binary vectors r^i_ℓ, where γ is a constant to be defined later and let

$$r^i_\ell[j] = \begin{cases} 1 & \text{with probability } \frac{1}{4k} \\ 0 & \text{with probability } 1 - \frac{1}{4k} \end{cases} \quad \text{for all } i = 1, 2, \ldots, \lambda \text{ and } j = 1, 2, \ldots, \ell$$

For a string x of length $\ell \in \{1, 2, \ldots, 2^{\lfloor \log n \rfloor}\}$ we define a sketch $\mathrm{sk}(x)$ to be a vector of length λ, where $\mathrm{sk}(x)[i] = r^i_\ell \cdot x \pmod 2$. In other words, to obtain $\mathrm{sk}(x)$ we sample each position of x with probability $\frac{1}{4k}$ and then sum the letters in the sampled positions modulo 2. All sketches can be computed in $\mathcal{O}(n \log^3 n)$ time by independently running the Fast Fourier Transform (FFT) algorithm for each of the vectors r^i_ℓ, and occupy $\mathcal{O}(n \log^2 n)$ space [15]. Each substring S can be decomposed uniquely as $x_1 x_2 \ldots x_r$, where $r \in \mathcal{O}(\log n)$ and $|x_1| > |x_2| > \ldots > |x_r|$ are powers of two. We define a sketch $\mathrm{sk}(S) = \sum_q \mathrm{sk}(x_q)$. Let $\delta_1 = \frac{1}{2}(1 - (1 - \frac{1}{4k})^k)$, $\delta_2 = \frac{1}{2}(1 - (1 - \frac{1}{4k})^{(1+\varepsilon)\cdot k})$, and $\Delta = \frac{(\delta_1 + \delta_2)}{2} \cdot \lambda$.

▶ **Lemma 3** ([26]). *For any i if the Hamming distance between S_1 and S_2 is at most k, then $\mathrm{sk}(S_1)[i] \neq \mathrm{sk}(S_2)[i]$ with probability at most δ_1. If the Hamming distance between S_1 and S_2 is at least $(1+\varepsilon)\cdot k$, then $\mathrm{sk}(S_1)[i] \neq \mathrm{sk}(S_2)[i]$ with probability at least δ_2.*

Proof. Let m be the Hamming distance between S_1 and S_2 and let $p_1, p_2, \ldots, p_m$ be the positions of the mismatches between them. If none of the positions $p_1, p_2, \ldots, p_m$ are sampled, then $\mathrm{sk}(S_1)[i] = \mathrm{sk}(S_2)[i]$, and otherwise for each way of sampling $p_1, p_2, \ldots, p_{m-1}$ exactly one of the two choices for p_m will give $\mathrm{sk}(S_1)[i] = \mathrm{sk}(S_2)[i]$. (Recall that the alphabet is binary.) Hence, the probability that $\mathrm{sk}(S_1)[i] \neq \mathrm{sk}(S_2)[i]$ is equal to $\frac{1}{2}(1 - (1 - \frac{1}{4k})^m)$, which is at most δ_1 if the Hamming distance between S_1 and S_2 is at most k, and at least δ_2 if the Hamming distance is greater than $(1+\varepsilon)\cdot k$. ◀

▶ **Lemma 4.** *If the Hamming distance between sketches* $\mathrm{sk}(S_1)$ *and* $\mathrm{sk}(S_2)$ *is bigger than* Δ*, then the Hamming distance between* S_1 *and* S_2 *is bigger than* k*. If the Hamming distance between sketches* $\mathrm{sk}(S_1)$ *and* $\mathrm{sk}(S_2)$ *is at most* Δ*, then the Hamming distance between* S_1 *and* S_2 *is at most* $(1+\varepsilon)\cdot k$*. Both claims are correct with probability at least* $1-\frac{1}{n^3}$*.*

Proof. Let χ_i be an indicator random variable that is equal to one if and only if $\mathrm{sk}(S_1)[i] = \mathrm{sk}(S_2)[i]$. The claim follows immediately from Lemma 3 and the following Chernoff bounds (see [2, Appendix A]). For λ independently and identically distributed binary variables $\chi_1, \chi_2, \ldots, \chi_\lambda$, $\Pr[\sum_i \chi_i \geq (p+\gamma)] \leq e^{-2\lambda\gamma^2}$ and $\Pr[\sum_i \chi_i < (p-\gamma)] \leq e^{-2\lambda\gamma^2}$, where $p = \Pr[\chi_i = 1]$. We put $\gamma = \frac{(\delta_2-\delta_1)}{2}$ and obtain that the error probability is at most $e^{-2\lambda\gamma^2} < \frac{1}{n^3}$. (Note that $\gamma = \Theta(1-e^{\varepsilon/4})$ is a constant depending on ε.) ◀

Suppose we wish to answer an $\mathrm{LCP}_{\tilde{k}}$ query on two suffixes S_1, S_2. It suffices to find the longest prefixes of S_1, S_2 such that the Hamming distance between their sketches is at most Δ. As mentioned above, these prefixes can be represented uniquely as a concatenation of strings of power-of-two lengths $\ell_1 > \ell_2 > \ldots > \ell_r$. To compute ℓ_1, we initialise it with the biggest power of two not exceeding n and compute the Hamming distance between the sketches of corresponding substrings. If it is smaller than Δ, we have found ℓ_1, otherwise we divide ℓ_1 by two and continue. Suppose that we already know $\ell_1, \ell_2, \ldots, \ell_i$ and let h_i be the Hamming distance between the sketches of prefixes of S_1 and S_2 of lengths $\ell_1 + \ell_2 + \ldots + \ell_i$. To compute ℓ_{i+1}, we initialise it with ℓ_i and then divide it by two until the Hamming distance between the corresponding substrings of length ℓ_{i+1} is at most $\Delta - h_i$. From above it follows that the algorithm is correct with probability at least $1-\frac{1}{n^2}$ (we estimate error probability by the union bound) and that the query time is $\mathcal{O}(\log^2 n)$. This completes the proof of Theorem 2.

4 Proof of Theorem 1

Recall that we are given two strings T_1, T_2 of length n, and if ℓ_k is the length of the longest common substring with k mismatches of T_1 and T_2, the objective is to return a substring of length at least ℓ_k that occurs in T_1 and T_2 with at most $(1+\varepsilon)\cdot k$ mismatches.

4.1 Algorithm

We start by preprocessing T_1 and T_2 as described in Theorem 2. The main phase of the algorithm is defined by three parameters t, w, and m to be specified later, and consists of $\theta(t! \log n)$ independent steps.

At each step we choose $\binom{w}{t}$ hash functions, where each hash function can be considered as a t-tuple of projections of strings of length n onto subsets of their positions of size m. Let $\mathcal{H}$ be a set of all projections of strings of length n onto a single position, i.e. the value of the i-th projection on a string of length n is simply its i-th letter. We start by choosing a set of w functions $u_r \in \mathcal{H}^m$, $r = 1, 2, \ldots, w$, uniformly at random. Each hash function h is defined to be a t-tuple of distinct functions u_r. More formally, $h = (u_{r_1}, u_{r_2}, \ldots, u_{r_t}) \in \mathcal{H}^{mt}$, where $1 \leq r_1 < r_2 < \ldots < r_t \leq w$. The fact that the hash functions are constructed from a small set of functions u_j will ensure faster running time for the algorithm.

Consider the set of all suffixes $S_1, S_2, \ldots, S_{2n}$ of T_1 and T_2. We append each suffix in the set with an appropriate number of letters $\$ \notin \Sigma$ so that all suffixes have length n and build a trie on strings $h(S_1), h(S_2), \ldots, h(S_{2n})$.

Algorithm 1 Longest common substring with approximately k mismatches.

```
Preprocess T_1, T_2 for LCP_k̃ queries
for i = 1, 2, ..., θ(t! log n) do
    for r = 1, 2, ..., w do
        Choose a function u_r ∈ H^m uniformly at random
        Preprocess u_r
    end for
    for all h = (u_{r_1}, u_{r_2}, ..., u_{r_t}) do
        Build a trie on h(S_1), h(S_2), ..., h(S_{2n})
        Augment the trie with an LCA data structure
    end for
    for all suffixes S of T_1 do
        Find the largest ℓ s.t. the total size of the ℓ-neighbourhoods of S is ≥ 2·C(w, t)
        Select a set S of 2·C(w, t) suffixes from the ℓ-neighbourhoods of S
        for all suffixes S' ∈ S do
            Compute LCP_k̃(S, S')
            Update the longest common substring with approximately k mismatches
        end for
    end for
end for
```

▶ **Theorem 5.** *After $\mathcal{O}(wn^{4/3}\log^{4/3} n)$-time and $\mathcal{O}(wn)$-space preprocessing of functions u_r, $r = 1, 2, \ldots, w$, for any hash function $h = (u_{r_1}, u_{r_2}, \ldots, u_{r_t})$ a trie on $h(S_1), h(S_2), \ldots, h(S_{2n})$ can be built in $\mathcal{O}(tn \log n)$ time and linear space.*

Let us defer the proof of the theorem until we complete the description of the algorithm and show Theorem 1. The algorithm preprocesses $u_1, u_2, \ldots, u_w$, and for each hash function h builds a trie on $h(S_1), h(S_2), \ldots, h(S_{2n})$. It then augments the trie with an LCA data structure, which can be done in linear time and space [14, 20]. Given two strings $h(S_i), h(S_j)$ the LCA data structure can find the length of their longest common prefix in constant time.

Consider a suffix S of T_1 and let h be a hash function projecting a string of length n onto a subset $\mathcal{P} \subseteq [1, n]$ of its positions. We define $h\big|_{[\ell]}$ to be a projection onto a subset $\mathcal{P} \cap [1, \ell]$ of positions. (In other words, $h\big|_{[\ell]}$ is a function h applied to a prefix of a string of length ℓ.) We further say that the ℓ-neighbourhood of S is the set of all suffixes S' of T_2 such that $h\big|_{[\ell]}(S) = h\big|_{[\ell]}(S')$. Note that for a fixed h and ℓ the size of the ℓ-neighbourhood of S is equal to the number of suffixes S' of T_2 such that the length of the longest common prefix of $h(S)$ and $h(S')$ is at least $|\mathcal{P} \cap [1, \ell]|$. From the properties of the lexicographic order it follows that if S', S'' are two suffixes of T_2 and $h(S'')$ is located between $h(S')$ and $h(S)$ in the trie for h, then the longest common prefix of $h(S')$ and $h(S)$ is no longer than the longest common prefix of $h(S'')$ and $h(S)$. Therefore, the larger ℓ is, the smaller the neighbourhood is. We use a simple binary search and the LCA data structures to find the largest ℓ such that the total size of ℓ-neighbourhoods for all hash functions is at least $2\binom{w}{t}$ in $\mathcal{O}(\binom{w}{t} \cdot \log^2 n)$ time. From the union of the ℓ-neighbourhoods we select a multiset $\mathcal{S}$ of $2\binom{w}{t}$ suffixes ensuring that all suffixes S' such that the longest common prefix of $h(S')$ and $h(S)$ has length at least $|\mathcal{P} \cap [1, \ell]| + 1$ are included. For each suffix $S' \in \mathcal{S}$ we compute the longest common prefix with approximately k mismatches of S' and S by one $\mathrm{LCP}_{\tilde{k}}$ query (Theorem 2). The longest of all retrieved prefixes, over all suffixes S of T_1, is returned as an answer. The algorithm is summarised in the figure above. We will now proceed to its complexity and correctness.

4.2 Complexity and correctness

To ensure the complexity bounds and correctness of the algorithm, we must carefully choose the parameters t, w, and m. Let $p_1 = 1 - k/n$, and $p_2 = 1 - (1+\varepsilon)\cdot k/n$. The intuition behind p_1 and p_2 is that if S_1, S_2 are two suffixes of length n and the Hamming distance between S_1 and S_2 is at most k, then p_1 is a lower bound for the probability of two letters $S_1[i], S_2[i]$ to be equal. On the other hand, p_2 is an upper bound for the probability of two letters $S_1[i], S_2[i]$ to be equal if the Hamming distance between S_1 and S_2 is at least $(1+\varepsilon)\cdot k$. Let $\rho = \log p_1 / \log p_2$, and define

$$t = \left\lceil \sqrt{\frac{\rho \log n}{\ln \log n}} \right\rceil + 1,\ w = \lceil n^{\rho/t} \rceil,\ \text{and } m = \left\lceil \frac{1}{t} \log_{p_2} \frac{1}{n^2} \right\rceil.$$

4.2.1 Complexity

To show the time complexity of the algorithm, we will start from the following simple observation.

▶ **Observation 6.** $\rho \le 1/(1+\varepsilon)$ and $2 \le t \le \sqrt{\log n}$.

Proof. By Bernoulli's inequality $(1-k/n)^{1+\varepsilon} \ge 1 - (1+\varepsilon)\cdot k/n$. Hence, we obtain that $\rho = \frac{\log(1-k/n)}{\log(1-(1+\varepsilon)k/n)} \le \frac{1}{1+\varepsilon}$. The second part of the lemma follows. ◀

▶ **Lemma 7.** *One step of the algorithm takes* $\mathcal{O}(wn^{4/3}\log^{4/3} n + \binom{w}{t}\cdot n\log^2 n)$ *time.*

Proof. By Theorem 5, after $\mathcal{O}(wn^{4/3}\log^{4/3} n)$-time preprocessing we can build a trie and an LCA data structure on strings $h(S_1), h(S_2), \ldots, h(S_{2n})$ for a hash function h in $\mathcal{O}(tn\log n) = \mathcal{O}(n\log^{3/2} n)$ time and there are $\binom{w}{t}$ hash functions in total. For each suffix of T_1 we then run $2\binom{w}{t}$ $\mathrm{LCP}_{\tilde{k}}$ queries, which takes $\mathcal{O}(\binom{w}{t}\cdot n\log^2 n)$ time. ◀

▶ **Corollary 8.** *The running time of the algorithm is* $\mathcal{O}(n^{1+1/(1+\varepsilon)}\log^2 n)$.

Proof. Preprocessing T_1, T_2 for $\mathrm{LCP}_{\tilde{k}}$ queries takes $\mathcal{O}(n\log^3 n/\varepsilon^2)$ time (see Theorem 2). Each step of the algorithm takes $\mathcal{O}(wn^{4/3}\log^{4/3} n + \binom{w}{t}\cdot n\log^2 n)$ time, and there are $\theta(t!\,\log n)$ steps in total. To estimate the total running time of the algorithm we notice that $wt! = \mathcal{O}(e^{\sqrt{\rho\log n \ln\log n}}) = \mathcal{O}(n^{o(1)})$ and $\binom{w}{t}\cdot t! \le \frac{w^t}{t!}t! = n^\rho \le n^{1/(1+\varepsilon)}$. Plugging these inequalities into the time bound for one step and recalling that $0 < \varepsilon < 2$, we obtain the claim. ◀

▶ **Lemma 9.** *The space complexity of the algorithm is* $\mathcal{O}(n^{1+1/(1+\varepsilon)})$.

Proof. The data structure for $\mathrm{LCP}_{\tilde{k}}$ queries requires $\mathcal{O}(n\log^3 n)$ space. At each step of the algorithm, preprocessing functions u_j requires $\mathcal{O}(wn) = \mathcal{O}(n^{1+o(1)})$ space and the tries occupy $\mathcal{O}(\binom{w}{t}\cdot n) = \mathcal{O}(n^{1+1/(1+\varepsilon)})$ space. ◀

4.2.2 Correctness

Let S be a suffix of T_1. Consider a set of the longest common prefixes with k mismatches of S and suffixes of T_2 and let ℓ_k be the maximal length of a prefix in this set achieved on some suffix S' of T_2.

▶ **Lemma 10.** *For each step with probability* $\ge \theta(1/t!)$ *there exists a hash function* h *such that* $h\big|_{[\ell_k]}(S) = h\big|_{[\ell_k]}(S')$.

Proof. Consider strings $S[1,\ell_k]\$^{n-\ell_k}$ and $S'[1,\ell_k]\$^{n-\ell_k}$. The Hamming distance between them is equal to the Hamming distance between $S[1,\ell_k]$ and $S'[1,\ell_k]$, which is k. Moreover, for any hash function h we have that $h(S[1,\ell_k]\$^{n-\ell_k}) = h(S'[1,\ell_k]\$^{n-\ell_k})$ if and only if $h\big|_{[\ell_k]}(S) = h\big|_{[\ell_k]}(S')$. Remember that each hash function is a t-tuple of functions u_j. Consequently, if $h(S[1,\ell_k]\$^{n-\ell_k}) \neq h(S'[1,\ell_k]\$^{n-\ell_k})$ for all hash functions h, the strings collide on at most $t-1$ functions u_j. By [5, Lemma A.1] the probability of this event for two strings at the Hamming distance k is at most $1-\theta(1/t!)$. ◀

As a corollary, we can choose the constant in the number of steps so that with probability $\geq 1-1/n^2$ there will exist a step of algorithm such that for at least one hash function we will have $h\big|_{[\ell_k]}(S) = h\big|_{[\ell_k]}(S')$. The set $\mathcal{S}$ of $2\binom{w}{t}$ suffixes that we sample from the ℓ-neighbourhoods of S might or might not include the suffix S'. If it does, then the $\mathrm{LCP}_{\tilde{k}}$ query for S' and S will return a substring of length $\geq \ell_k$ with high probability. If it does not, then by the definition of neighbourhoods for each suffix $S'' \in \mathcal{S}$ belonging to the neighbourhood for a hash function g we have $g\big|_{[\ell_k]}(S) = g\big|_{[\ell_k]}(S'')$. We will show that only for a small number of such suffixes an $\mathrm{LCP}_{\tilde{k}}$ query can return a substring of length smaller than ℓ_k.

▶ **Lemma 11.** *With probability $\geq 1-2/n^2$ there are at most $\binom{w}{t}$ suffixes S'' such that $g\big|_{[\ell_k]}(S) = g\big|_{[\ell_k]}(S'')$ but the $\mathrm{LCP}_{\tilde{k}}$ query returns a substring shorter than ℓ_k.*

Proof. If the $\mathrm{LCP}_{\tilde{k}}$ query for S and S'' returns a substring shorter than ℓ_k, then with high probability the Hamming distance between $S[1,\ell_k]$ and $S''[1,\ell_k]$ is at least $(1+\varepsilon)\cdot k$. Remember that a hash function g can be considered as a projection onto a random subset of positions of size mt, and therefore we obtain

$$\Pr[g\big|_{[\ell_k]}(S) = g\big|_{[\ell_k]}(S'')] = \Pr[g(S[1,\ell_k]\$^{n-\ell_k}) = g(S'')[1,\ell_k]\$^{n-\ell_k}] \leq (p_2)^{mt} = \frac{1}{n^2}$$

We can consider an indicator random variable that is equal to one if for a suffix S'' such that $g\big|_{[\ell_k]}(S) = g\big|_{[\ell_k]}(S'')$ the $\mathrm{LCP}_{\tilde{k}}$ query returns a substring shorter than ℓ_k, and to zero otherwise. By linearity, the expectation of their sum is at most $\frac{2}{n^2}\cdot\binom{w}{t}$. The claim follows from Markov's inequality. ◀

From Lemmas 10 and 11 and Theorem 2 it follows that the algorithm correctly finds the value of ℓ for the suffix S of T_1 with error probability $\leq 3/n^2$. Applying the union bound, we obtain that the error probability of the algorithm is constant.

4.3 Proof of Theorem 5

Recall that h is a t-tuple of functions u_r, i.e $h = (u_{r_1}, u_{r_2}, \ldots, u_{r_t})$, where $1 \leq r_1 < r_2 < \ldots < r_t \leq w$. Below we will show that after the preprocessing of functions u_r we will be able to compute the longest common prefix of any two strings $u_r(S_i), u_r(S_j)$ in $\mathcal{O}(1)$ time. As a result, we will be able to compute the longest common prefix of $h(S_i), h(S_j)$ in $\mathcal{O}(t)$ time. It also follows that we will be able to compare any two strings $h(S_i), h(S_j)$ in $\mathcal{O}(t)$ time as their order is defined by the letter following the longest common prefix. Therefore, we can sort strings $h(S_1), h(S_2), \ldots, h(S_{2n})$ in $\mathcal{O}(tn\log n)$ time and $\mathcal{O}(n)$ space and then compute the longest common prefix of each two adjacent strings in $\mathcal{O}(tn)$ time. The trie on $h(S_1), h(S_2), \ldots, h(S_{2n})$ can then be built in $\mathcal{O}(n)$ time by imitating its depth-first traverse.

It remains to explain how we preprocess functions u_r, $r = 1, 2, \ldots, w$. For each function u_r it suffices to build a trie on strings $u_r(S_1), u_r(S_2), \ldots, u_r(S_{2n})$ and to augment it with an

LCA data structure [14, 20]. We will consider two different methods for constructing the trie with time dependent on m. No matter what the value of m is, one of these methods will have $\mathcal{O}(n^{4/3}\log^{1/3} n)$ running time. Let u_r be a projection along a subset $\mathcal{P}$ of positions $1 \le p_1 \le p_2 \le \cdots \le p_m \le n$ and denote $T = T_1\$^n T_2\$^n$.

▶ **Lemma 12.** *The trie can be built in $\mathcal{O}(\sqrt{m}n\log n)$ time and $\mathcal{O}(n)$ space correctly with error probability at most $1/n^3$.*

Proof. We partition $\mathcal{P}$ into disjoint subsets $B_1, B_2, \ldots, B_{\sqrt{m}}$, where

$$B_\ell = \{p_{\ell,1}, p_{\ell,2}, \ldots, p_{\ell,\sqrt{m}}\} = \{p_{(\ell-1)\sqrt{m}+q} \mid q \in [1, \sqrt{m}]\}.$$

Now u_r can be represented as a $\sqrt{m}$-tuple of projections $b_1, b_2, \ldots, b_{\sqrt{m}}$ onto the subsets $B_1, B_2, \ldots, B_{\sqrt{m}}$ respectively. We will build the trie by layers to avoid space overhead. Suppose that we have built the trie for a function $(b_1, b_2, \ldots, b_{\ell-1})$ and we want to extend it to the trie for $(b_1, b_2, \ldots, b_{\ell-1}, b_\ell)$.

Let p be a random prime of value at most $n^{\mathcal{O}(1)}$. We create a vector χ of length n, where $\chi[p_{\ell,q}] = 2^{\sqrt{m}-1-q}$ and zero for all positions not in B_ℓ. We then run the FFT algorithm for χ and T in the field $\mathbb{Z}_p$ [15]. The output of the FFT algorithm will contain convolutions of χ and all suffixes $S_1, S_2, \ldots, S_{2n}$. The convolution of χ and a suffix S_i is the Karp-Rabin fingerprint [24] $\varphi_{\ell,i}$ of $b_\ell(S_i)$, where

$$\varphi_{\ell,i} = \sum_{q=1}^{\sqrt{m}} S_i[p_{\ell,q}] \cdot 2^{\sqrt{m}-1-q} \pmod p$$

If the fingerprints of $b_\ell(S_i)$ and $b_\ell(S_j)$ are equal, then $b_\ell(S_i)$ and $b_\ell(S_j)$ are equal with probability at least $1 - \frac{1}{n^4}$, and otherwise they differ. For a fixed leaf of the trie for $(b_1, b_2, \ldots, b_{\ell-1})$ we sort all suffixes that end in it by fingerprints $\varphi_{\ell,i}$, which takes $\mathcal{O}(n\log n)$ time in total. For each two suffixes S_i, S_j that end in the same leaf, adjacent and have $\varphi_{\ell,i} \neq \varphi_{\ell,j}$, we compare $b_\ell(S_i)$ and $b_\ell(S_j)$ letter-by-letter in $\mathcal{O}(\sqrt{m})$ time to find their longest common prefix. Note that this letter-by-letter comparison step will be executed at most n times, and therefore will take $\mathcal{O}(\sqrt{m}n)$ time in total. We then append each leaf with a trie on strings $b_\ell(S_i)$ that can be built by imitating its depth-first traverse, which takes $\mathcal{O}(n)$ time for a layer. ◀

The second method builds the trie in $\mathcal{O}(n^2\log^2 n/m)$ time by the algorithm described in the first paragraph of this section, and we only need to give a method for comparing the longest common prefix of $u_r(S_i)$ and $u_r(S_j)$ (or, equivalently, the first position where $u_r(S_i)$ and $u_r(S_j)$ differ.)

▶ **Lemma 13** ([5])**.** *After $\mathcal{O}(n)$-time and space preprocessing the first position where two strings $u_r(S_i)$ and $u_r(S_j)$ differ can be found in $\mathcal{O}(n\log n/m)$ time correctly with error probability at most $1/n^3$.*

Proof. We start by building the suffix tree for the string T. The suffix tree can be built in $\mathcal{O}(n)$ time and space [32, 13, 19]. Furthermore, we augment the suffix tree with an LCA data structure in $\mathcal{O}(n)$ time [14, 20].

Let $\ell = 3n\log n/m$. We can find the first ℓ positions $q_1 < q_2 < \ldots < q_\ell$ where S_i and S_j differ in $\mathcal{O}(n\log n/m)$ time using the kangaroo method [27, 17]. The idea of the kangaroo method is as follows. We can find q_1 by one query to the LCA data structure in $\mathcal{O}(1)$ time. After removing the first q_1 positions of S_i, S_j, we obtain suffixes S_{i+q_1}, S_{j+q_1} and find q_2

by another query to the LCA data structure, and so on. If at least one of the positions $q_1, q_2, \ldots, q_\ell$ belongs to $\mathcal{P}$, then we return the first such position as an answer, and otherwise we say that $u_r(S_i) = u_r(S_j)$.

Let us show that if p is the first position where $u_r(S_i)$ and $u_r(S_j)$ differ, then p belongs to $\{q_1, q_2, \ldots, q_\ell\}$ with high probability. Because $q_1 < q_2 < \ldots < q_\ell$ are the first ℓ positions where S_i and S_j differ, it suffices to show that at least one of these positions belongs to $\mathcal{P}$. We rely on the fact $\mathcal{P}$ is a random subset of $[1, n]$. We have $Pr[q_1, q_2, \ldots, q_\ell \notin \mathcal{P}] = (1 - \ell/n)^m = (1 - 3\log n/m)^m \leq n^{-3}$. ◀

As a corollary of Lemmas 12 and 13, the trie on strings $u_r(S_1), u_r(S_2), \ldots, u_r(S_{2n})$ can be built in $\mathcal{O}(\min\{\sqrt{m}, n\log n/m\} \cdot n\log n) = \mathcal{O}(n^{4/3}\log^{4/3} n)$ time and $\mathcal{O}(n)$ space correctly with high probability which implies Theorem 5 as explained in the beginning of this section.

References

1. A. Abboud, R. Williams, and H. Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 218–230, 2015.
2. N. Alon and J. Spencer. *The Probabilistic Method*. John Wiley and Sons, 1992.
3. S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.
4. S. Aluru, A. Apostolico, and S. Thankachan. Proceedings of the 19th annual international conference on research in computational molecular biology. pages 1–12, 2015.
5. A. Andoni and P. Indyk. Efficient algorithms for substring near neighbor problem. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1203–1212, 2006.
6. M. Babenko and T. Starikovskaya. Computing longest common substrings via suffix arrays. In *Proceedings of the 3rd International Computer Science Symposium in Russia*, pages 64–75, 2008.
7. M. Babenko and T. Starikovskaya. Computing the longest common substring with one mismatch. *Problems of Information Transmission*, 47(1):28–33, 2011.
8. A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (Unless SETH is false). In *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing (STOC)*, pages 51–58, 2015.
9. P. Bille, I.L. Gørtz, and J. Kristensen. Longest common extensions via fingerprinting. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications*, pages 119–130, 2012.
10. P. Bille, I.L. Gørtz, B. Sach, and H.W. Vildhøj. Time-space trade-offs for longest common extensions. *J. of Discrete Algorithms*, 25:42–50, March 2014.
11. K. Bringmann and M. Kunnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of the 56th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 79–97, 2015.
12. M. Crochemore, G.M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.
13. M. Farach-Colton. Optimal suffix tree construction with large alphabets. In *Procedings of the 38th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.
14. J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Conference on Combinatorial Pattern Matching*, pages 36–48, 2006.

15 M. J. Fischer and M. S. Paterson. String-matching and other products. In *Complexity of Computation*, volume 7, pages 113–125. SIAM AMS, 1974.

16 T. Flouri, E. Giaquinta, K. Kobert, and E. Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6–8):643–647, 2015.

17 Z. Galil and R. Giancarlo. Parallel string matching with k mismatches. *Theoretical Computer Science*, 51(3):341–348, 1987.

18 S. Grabowski. A note on the longest common substring with k-mismatches problem. *Information Processing Letters*, 115(6–8):640–642, 2015.

19 D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

20 D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, May 1984.

21 L.C.K. Hui. Color set size problem with applications to string matching. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture notes in computer science*, pages 230–243, 1992.

22 L. Ilie, G. Navarro, and L. Tinta. The longest common extension problem revisited and applications to approximate string searching. *J. of Discrete Algorithms*, 8(4):418–428, 2010.

23 P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

24 R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development – Mathematics and computing*, 31(2):249–260, 1987.

25 T. Kociumaka, T. Starikovskaya, and Vildhøj H. W. Sublinear space algorithms for the longest common substring problem. In *Proceedings of the 22nd Annual European Symposium on Algorithms*, volume 8737 of *Lecture notes in computer science*, pages 605–617, 2014.

26 E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 614–623, New York, NY, USA, 1998. ACM.

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

28 C.A. Leimeister and B. Morgenstern. kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014.

29 W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

30 T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

31 T. Starikovskaya and H.W. Vildhøj. Time-space trade-offs for the longest common substring problem. In *Proceedings of the 24th Annual Symposium in Combinatorial Pattern Matching*, volume 7922 of *Lecture notes in computer science*, pages 223–234, 2013.

32 P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1–11, 1973.

Fully-online Construction of Suffix Trees for Multiple Texts

Takuya Takagi[1], Shunsuke Inenaga[2], and Hiroki Arimura[3]

1 **Graduate School of IST, Hokkaido University, Japan**
`tkg@ist.hokudai.ac.jp`
2 **Department of Informatics, Kyushu University, Japan**
`inenaga@inf.kyushu-u.ac.jp`
3 **Graduate School of IST, Hokkaido University, Japan**
`arim@ist.hokudai.ac.jp`

Abstract

We consider *fully-online* construction of indexing data structures for multiple texts. Let $\mathcal{T} = \{T_1, \ldots, T_K\}$ be a collection of texts. By fully-online, we mean that a new character can be appended to any text in $\mathcal{T}$ *at any time*. This is a natural generalization of *semi-online* construction of indexing data structures for multiple texts in which, after a new character is appended to the kth text T_k, then its previous texts $T_1, \ldots, T_{k-1}$ will remain static. Our fully-online scenario arises when we maintain dynamic indexes for multi-sensor data. Let N and σ denote the total length of texts in $\mathcal{T}$ and the alphabet size, respectively. We first show that the algorithm by Blumer et al. [Theoretical Computer Science, 40:31-55, 1985] to construct the *directed acyclic word graph* (*DAWG*) for $\mathcal{T}$ can readily be extended to our fully-online setting, retaining $O(N \log \sigma)$-time and $O(N)$-space complexities. Then, we give a sophisticated fully-online algorithm which constructs the *suffix tree* for $\mathcal{T}$ in $O(N \log \sigma)$ time and $O(N)$ space. A key idea of this algorithm is synchronized maintenance of the DAWG and the suffix tree.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases suffix trees, DAWGs, multiple texts, online algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.22

1 Introduction

Text indexing is a fundamental problem in computer science, which plays important roles in many applications including text retrieval, molecular biology, signal processing, and sensor data analysis. In this paper, we focus on indexing a collection of multiple texts, so that subsequent pattern matching queries can be answered quickly. In particular, we study online indexing for a collection $\mathcal{T}$ of multiple texts, where a new character can be appended to each text at *any* time. Such fully-online indexing for multiple growing texts has potential applications to continuous processing of data streams, where a number of symbolic events or data items are produced from multiple, rapid, time-varying, and unbounded data streams [2, 11]. For example, motif mining system tries to discover characteristic or interesting collective behaviors, such as frequent path or anomalies, from data streams generated by a collection of moving objects or sensors [14, 11].

It is known that suffix trees [13] and DAWGs [4] can be efficiently constructed for a collection of growing texts in the *semi-online* setting, where only the last inserted text can be grown. However, these existing semi-online algorithms to maintain a suffix tree or a DAWG for multiple texts are not sufficient to construct indexing structures for multiple data streams which grow in a fully-online manner.

We propose how the DAWG and the suffix tree can be incrementally constructed for a fully-online text collection. First, we observe that Blumer et al.'s construction [4] for DAWGs and Weiner's right-to-left construction [15] for suffix trees can readily be adapted to solve this problem. Hence, at any moment during the fully-online growth of the texts, we can find all *occ* occurrences of a given pattern of length M in the current text collection in $O(M \log \sigma + occ)$ time.

Our next goal is to extend Ukkonen's construction [13] to fully-online left-to-right construction of suffix trees for multiple texts. A motivation of this goal is that a growing suffix tree can be enhanced with powerful semi-dynamic tree data structures such as those for *nearest marked ancestor (NMA) queries* [16], *lowest common ancestor (LCA) queries* [7], and *level ancestor (LA) queries* [1]. Note that these data structures cannot be applied to DAWGs, and that the same query results cannot be obtained on the suffix tree maintained in a Weiner-like right-to-left online manner since the suffix tree obtained in this manner inherently indexes the *reversed* texts in the collection. However, it turns out that this goal is a big algorithmic challenge, because: (A) In Ukkonen's algorithm, a pointer called the *active point* keeps track of the insertion points of suffixes in decreasing order of length. The efficiency of Ukkonen's algorithm is due to the monotonicity of the tracking path of the active point. However, unfortunately this monotonicity does not hold in the fully-online setting for multiple texts. (B) Due to the non-monotonicity mentioned above, Ukkonen's technique to amortize the cost to track the suffix insertion points does not work in our case. (C) Ukkonen's "open edge" technique to maintain the leaves does not work in our case, either. In Section 5 we will explain in more details why and how these problems arise in our fully-online setting. In this paper, we present a number of new novel techniques to overcome all the difficulties above. As a final result, we propose the first *optimal* $O(N \log \sigma)$-time $O(N)$-space fully-online left-to-right construction algorithm for a suffix tree of multiple texts over a general ordered alphabet of size σ, where N is the final total length of the texts.

1.1 Related work

We note that we can obtain fully-online text index for multiple texts using existing more general dynamic text indices as follows. To use the index of Ferragina and Grossi [8] which permits character-wise updates, we build a text $\$_1 \cdots \$_K$ which initially consists only of K delimiters. Then, appending a character a to the kth text in the collection reduces to prepending a to the kth delimiter $\$_k$. Using this approach, the index of Ferragina and Grossi [8] takes $O(N \log N)$ total time to be constructed, requires $O(N \log N)$ space, and allows pattern matching in $O(M + \log N + N \log M + occ)$ time. Using the compressed index for a dynamic text collection of Chan et al. [6], we can append a new character a to the kth text T_k by removing T_k and then adding $T_k a$ in $O(|T_k|)$ time. This yields a fully-online index with $O(N^2 \log N)$ construction time and $O(N)$ bits of space (or $O(N/\log N)$ words of space assuming $\Theta(\log N)$-bit machine word), supporting pattern matching in $O(M \log N + occ \log^2 N)$ time.

2 Preliminaries

2.1 Strings

Let Σ be a general ordered alphabet. Any element of Σ^* is called a *string*. For any string T, let $|T|$ denote its length. Let ε be the empty string, namely, $|\varepsilon| = 0$. If $T = XYZ$, then X, Y, and Z are called a *prefix*, a *substring*, and a *suffix* of T, respectively. For any

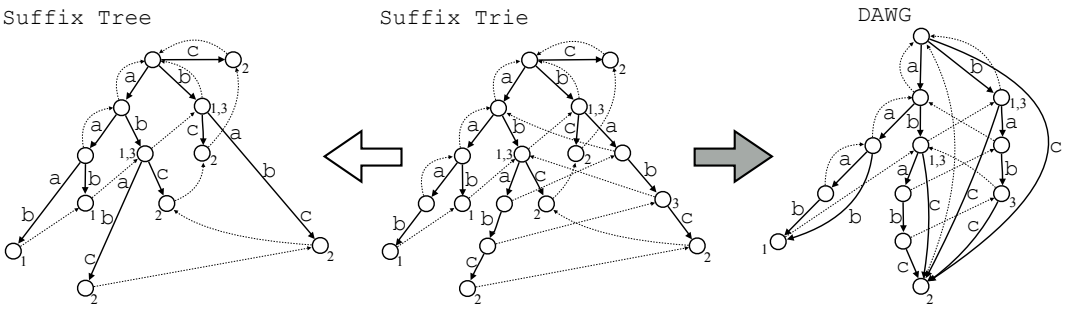


Figure 1 Illustration for $STrie(\mathcal{T})$, $STree(\mathcal{T})$, and $DAWG(\mathcal{T})$ with $\mathcal{T} = \{T_1 = \texttt{aaab}, T_2 = \texttt{ababc}, T_3 = \texttt{bab}\}$. The solid arrows and broken arrows represent the edges and the suffix links of each data structure, respectively. The number k ($k = 1, 2, 3$) beside each node indicates that the node represents a suffix of T_k. The nodes $[\texttt{ab}]_{\mathcal{T}}$ and $[\texttt{b}]_{\mathcal{T}}$ are separated in $DAWG(\mathcal{T})$ since the node `bab` in $STrie(\mathcal{T})$ is represents a suffix of T_3, while the node `abab` does not (see also the subtrees rooted at nodes `ab` and `b` in $STrie(\mathcal{T})$).

$1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. For any $1 \leq i \leq |T|$, let $T[i]$ denote the ith character of T. For any string T, let $Suffix(T)$ denote the set of suffixes of T, and for any set $\mathcal{T}$ of strings, let $Suffix(\mathcal{T})$ denote the set of suffixes of all strings in $\mathcal{T}$. Namely, $Suffix(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} Suffix(T)$. For any string T, let $\overline{T}$ denote the reversed string of T, i.e., $\overline{T} = T[|T|] \cdots T[1]$.

Let $\mathcal{T} = \{T_1, \ldots, T_K\}$ be a collection of K texts. For any $1 \leq k \leq K$, let $lrs_{\mathcal{T}}(T_k)$ be the longest repeating suffix of T_k that occurs at least twice in $\mathcal{T}$.

2.2 Suffix trees and DAWGs for multiple texts

The suffix trie for a text collection $\mathcal{T} = \{T_1, \ldots, T_K\}$, denoted $STrie(\mathcal{T})$, is a trie which represents $Suffix(\mathcal{T})$. The size of $STrie(\mathcal{T})$ is $O(N^2)$, where N is the total length of texts in $\mathcal{T}$. We identify each node v of $STrie(\mathcal{T})$ with the string that v represents. A substring x of a text in $\mathcal{T}$ is said to be *branching* in $\mathcal{T}$, if there exist two distinct characters $a, b \in \Sigma$ such that both xa and xb are substrings of some texts in $\mathcal{T}$. Clearly, node x of $STrie(\mathcal{T})$ is branching iff x is branching in $\mathcal{T}$. For each node av of $STrie(\mathcal{T})$ with $a \in \Sigma$ and $v \in \Sigma^*$, let $slink(av) = v$. This auxiliary edge $slink(av) = v$ from av to v is called a *suffix link*.

The *suffix tree* [15] for a text collection $\mathcal{T}$, denoted $STree(\mathcal{T})$, is a "compacted trie" which represents $Suffix(\mathcal{T})$. $STree(\mathcal{T})$ is obtained by compacting every path of $STrie(\mathcal{T})$ which consists of non-branching internal nodes (see Fig. 1). Since every internal node of $STree(\mathcal{T})$ is branching, and since there are at most N leaves in $STree(\mathcal{T})$, the numbers of edges and nodes are $O(N)$. The edge labels of $STree(\mathcal{T})$ are non-empty substrings of some text in $\mathcal{T}$. By representing each edge label x with a triple $\langle k, i, j \rangle$ of integers s.t. $x = T_k[i..j]$, $STree(\mathcal{T})$ can be stored with $O(N)$ space. We say that any branching (resp. non-branching) substring of $\mathcal{T}$ is an *explicit node* (resp. *implicit node*) of $STree(\mathcal{T})$. An implicit node x is represented by a triple (v, a, ℓ), called a *reference* to x, such that v is an explicit ancestor of x, a is the first character of the path from v to x, and ℓ is the length of the path from v to x. A reference (v, a, ℓ) to node x is called *canonical* if v is the lowest explicit ancestor of x. For each node av of $STree(\mathcal{T})$ with $a \in \Sigma$ and $v \in \Sigma^*$, let $slink(av) = v$.

The *directed acyclic word graph* [3, 4] of a text collection $\mathcal{T}$, denoted $DAWG(\mathcal{T})$, is a smallest DAG which represents $Suffix(\mathcal{T})$. $DAWG(\mathcal{T})$ is obtained by merging identical subtrees of $STrie(\mathcal{T})$ connected by the suffix links (see Fig. 1). Hence, the label of every edge of $DAWG(\mathcal{T})$ is a single character. The numbers of nodes and edges of $DAWG(\mathcal{T})$ are

$O(N)$ [4], and hence $DAWG(\mathcal{T})$ can be stored with $O(N)$ space. $DAWG(\mathcal{T})$ can be defined formally as follows: For any string x, let $Epos_{\mathcal{T}}(x)$ be the set of ending positions of x in the texts in $\mathcal{T}$, i.e., $Epos_{\mathcal{T}}(x) = \{(k, j) \mid x = T_k[j - |x| + 1..j], 1 \leq j \leq |T_k|, 1 \leq k \leq K\}$. Consider an equivalence relation $\equiv_{\mathcal{T}}$ on substrings x, y of texts in $\mathcal{T}$ such that $x \equiv_{\mathcal{T}} y$ iff $Epos_{\mathcal{T}}(x) = Epos_{\mathcal{T}}(y)$. For any substring x of texts of $\mathcal{T}$, let $[x]_{\mathcal{T}}$ denote the equivalence class w.r.t. $\equiv_{\mathcal{T}}$. There is a one-to-one correspondence between each node v of $DAWG(\mathcal{T})$ and each equivalence class $[x]_{\mathcal{T}}$, and hence we will identify each node v of $DAWG(\mathcal{T})$ with its corresponding equivalence class $[x]_{\mathcal{T}}$. Let $long([x]_{\mathcal{T}})$ denote the longest member of $[x]_{\mathcal{T}}$. By the definition of equivalence classes, $long([x]_{\mathcal{T}})$ is unique for each $[x]_{\mathcal{T}}$ and every member of $[x]_{\mathcal{T}}$ is a suffix of $long([x]_{\mathcal{T}})$. If x, xa are substrings of some text in $\mathcal{T}$ with $x \in \Sigma^*$ and $a \in \Sigma$, then there exists an edge labeled with character $a \in \Sigma$ from node $[x]_{\mathcal{T}}$ to node $[xa]_{\mathcal{T}}$. This edge is called *primary* if $|long([x]_{\mathcal{T}})| + 1 = |long([xa]_{\mathcal{T}})|$, and is called *secondary* otherwise. For each node $[x]_{\mathcal{T}}$ of $DAWG(\mathcal{T})$ with $|x| \geq 1$, let $slink([x]_{\mathcal{T}}) = y$, where y is the longest suffix of $long([x]_{\mathcal{T}})$ which does not belong to $[x]_{\mathcal{T}}$. In the example of Fig. 1, $[\mathtt{aaab}]_{\mathcal{T}} = \{\mathtt{aaab}, \mathtt{aab}\}$. The edge labeled with $\mathtt{b}$ from node $[\mathtt{aaa}]_{\mathcal{T}}$ to node $[\mathtt{aaab}]_{\mathcal{T}}$ is primary, while the edge labeled with $\mathtt{b}$ from $[\mathtt{aa}]_{\mathcal{T}}$ to node $[\mathtt{aaab}]_{\mathcal{T}}$ is secondary. $slink([\mathtt{aaab}]_{\mathcal{T}}) = [\mathtt{ab}]_{\mathcal{T}}$.

The following fact follows from the definition of branching substrings:

▶ **Fact 1.** *For any substring x of texts in $\mathcal{T}$, node x is branching (explicit) in $STree(\mathcal{T})$ iff node $[x]_{\mathcal{T}}$ is branching in $DAWG(\mathcal{T})$.*

2.3 Fully-online text collection

We consider a collection $\{T_1, \ldots, T_K\}$ of K growing texts, where each text T_k $(1 \leq k \leq K)$ is initially the empty string ε. Given a pair (k, a) of a text id k and a character $a \in \Sigma$ which we call an *update operator*, the character a is appended to the k-th text of the collection. For a sequence U of update operators, let $U[1..i]$ denote the sequence of the first i update operators in U with $0 \leq i \leq |U|$. Also, for $0 \leq i \leq |U|$ let $\mathcal{T}_{U[1..i]}$ denote the collection of texts which have been updated according to the first i update operators of U. For instance, consider a text collection of three texts which grow according to the following sequence $U = (1, \mathtt{a}), (2, \mathtt{b}), (2, \mathtt{a}), (3, \mathtt{a}), (1, \mathtt{a}), (3, \mathtt{c}), (3, \mathtt{b}), (2, \mathtt{b}), (1, \mathtt{a}), (1, \mathtt{b}), (3, \mathtt{c}), (3, \mathtt{b}), (1, \mathtt{c}), (3, \mathtt{b}), (2, \mathtt{c})$ of 15 update operators. Then,

$$\mathcal{T}_{U[1..0]} = \left\{ \begin{array}{l} \varepsilon \\ \varepsilon \\ \varepsilon \end{array} \right\}, \ \ldots, \ \mathcal{T}_{U[1..14]} = \left\{ \begin{array}{llllll} {\scriptstyle 1} & {\scriptstyle 5} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 13} & \\ \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \\ {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 8} & & & \\ \mathtt{b} & \mathtt{a} & \mathtt{b} & & & \\ {\scriptstyle 4} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 14} \\ \mathtt{a} & \mathtt{c} & \mathtt{b} & \mathtt{c} & \mathtt{b} & \mathtt{b} \end{array} \right\}, \ \mathcal{T}_{U[1..15]} = \left\{ \begin{array}{llllll} {\scriptstyle 1} & {\scriptstyle 5} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 13} & \\ \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \\ {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 8} & {\scriptstyle 15} & & \\ \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{c} & & \\ {\scriptstyle 4} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 14} \\ \mathtt{a} & \mathtt{c} & \mathtt{b} & \mathtt{c} & \mathtt{b} & \mathtt{b} \end{array} \right\}$$

where the superscript i over each character a in the k-th text implies that $U[i] = (k, a)$. For instance, $U[15] = (2, \mathtt{c})$ and hence $\mathtt{c}$ was appended to the 2nd text $T_2 = \mathtt{bab}$ in $\mathcal{T}_{U[1..14]}$, yielding $T_2 = \mathtt{babc}$ in $\mathcal{T}_{U[1..15]}$.

If there is no restriction on U like the one in the example above, then U is called *fully-online*. If there is a restriction on U such that once a new character is appended to the k-th text, then no characters will be appended to its previous $k - 1$ texts, then U is called *semi-online*. Hence, any semi-online sequence of update operators is of form $(1, T_1[1]), \ldots, (1, T_1[|T_1|]), \ldots, (K, T_K[1]), \ldots, (K, T_K[|T_K|])$.

Section 3 reviews previous algorithms which incrementally construct the DAWG and the suffix tree for a growing text collection in the semi-online setting. Sections 4 and 5 propose our new algorithms which incrementally construct the DAWG and the suffix tree for a text collection in the fully-online setting, respectively.

3 Semi-online construction algorithms

3.1 Blumer et al.'s semi-online DAWG construction algorithm

We recall Blumer et al.'s algorithm [4] which incrementally builds $DAWG(\mathcal{T}_U)$ for a given semi-online sequence U of update operators of length N. Since U is semi-online, at each step i ($0 \leq i \leq N$) of the semi-online update, there exists a unique k ($1 \leq k < K$) such that $T_1, \ldots, T_{k-1}$ will be static for all the following i'th steps ($i \leq i' \leq N$), T_k is now growing from left to right, and $T_{k+1}, \ldots, T_K$ are still the empty strings. Assume that $U[i] = (k, a)$, and hence a new character a is appended to the kth text in the collection at the ith step. For ease of notation, let $\mathcal{T}' = \mathcal{T}_{U[1..i-1]}$ and $\mathcal{T} = \mathcal{T}_{U[1..i]}$. Also, assume that $DAWG(\mathcal{T}')$ has already been constructed. In updating $DAWG(\mathcal{T}')$ to $DAWG(\mathcal{T})$, we have to assure that all suffixes of the extended text $T_k a$ will be represented by $DAWG(\mathcal{T})$. These suffixes are categorized to three different types:

Type-1 The suffixes of $T_k a$ that are longer than $lrs_{\mathcal{T}'}(T_k)a$.

Type-2 The suffixes of $T_k a$ that are not longer than $lrs_{\mathcal{T}'}(T_k)a$ and are longer than $lrs_{\mathcal{T}}(T_k a)$.

Type-3 The suffixes of $T_k a$ that are not longer than $lrs_{\mathcal{T}}(T_k a)$.

Blumer et al.'s algorithm inserts the suffixes of $T_k a$ in decreasing order of length, from the Type-1 ones to the Type-2 ones. By definition, the Type-3 ones are already represented by $DAWG(\mathcal{T}')$, and hence, we need not insert them explicitly.

Their algorithm maintains an invariant v which indicates node $[T_k]_{\mathcal{T}'}$, called the *active point*, from which the update starts. There are two cases to happen:

1. If there is an out-going edge labeled with a from v, then $T_k a = lrs_{\mathcal{T}}(T_k a)$, which implies all suffixes of $T_k a$ are of Type-3. There are two subcases:
   - **a.** If the edge labeled with a is primary, then no updates to the graph topology are needed. The new active point for the next step is on $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$.
   - **b.** If the edge labeled with a is secondary, then the graph topology needs to be updated. Since the edge is secondary, every member Xa of $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$ that is longer than $T_k a$ is not a suffix of $T_k a$, while every member Ya of $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$ that is not longer than $T_k a$ is a Type-3 suffix of $T_k a$. This implies that $Epos_{\mathcal{T}}(lrs_{\mathcal{T}}(T_k a)) \supset Epos_{\mathcal{T}}(Xa)$. By the definition of the nodes of DAWGs (recall Section 2.2), the node u is split into two nodes $z = [Xa]_{\mathcal{T}}$ and $w = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$: First, a new node w is created. All secondary in-coming edges of u corresponding to Type-3 suffixes Ya are redirected to w. This can be done by traversing the chain of the suffix links starting from v. All the out-going edges of u are copied to w. Now, node w is complete, and the node u with its remaining in-coming edges is the other new node z. The suffix link of u is inherited by w, and the suffix link of z is set to w. The new active point for the next step is on node w.
2. If there is no out-going edge labeled with a from the active point v, then a new sink s is created. The Type-1 suffixes are inserted by making a new edge labeled by a from $v = [T_k]_{\mathcal{T}'}$ to s. To insert the Type-2 suffixes, the active point v moves by updating $v \leftarrow slink(v)$. Then the following procedure is repeated until an out-going edge labeled with a from the active point is found: (i) A new edge labeled with a from v to s is created. (ii) The active point v moves by updating $v \leftarrow slink(v)$. The node u where the above procedure ends is $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$, and the new sink s is exactly $[T_k a]_{\mathcal{T}}$ which represent all Type-1 and Type-2 suffixes of $T_k a$. There are two cases:

a. If the edge labeled with a from the last locus v of the active point to u is primary, then $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$. Thus no updates to the graph topology are needed. The suffix link of the new sink $s = [T_k a]_{\mathcal{T}}$ is set to u.

b. If the edge labeled with a from the last locus v of the active point to u is secondary, then as in Case 1b, u is split into two nodes w and z where w represents the members of u that are longer than the longest repeating suffix $lrs_{\mathcal{T}}(T_k a)$ (none of these members is a suffix of $T_k a$), and z represents the members of u which are Type-3 suffixes of $T_k a$. The suffix link of the new sink s is set to z.

In both subcases above, the new active point is on the new sink $s = [T_k a]_{\mathcal{T}}$.

It is not difficult to see that if the total number of new nodes, edges, and suffix links is q, then the above update takes $O(q \log \sigma)$ time, where the $\log \sigma$ term is due to searching for an out-going edge labeled by a. Since no existing nodes, edges, or suffix links are deleted during the updates, and since the size of $DAWG(\mathcal{T}_U)$ is $O(N)$, the amortized time for the update is $O(\log \sigma)$. Hence, $DAWG(\mathcal{T}_U)$ can be constructed in $O(N \log \sigma)$ time and $O(N)$ space in the semi-online setting.

3.2 Ukkonen's semi-online suffix tree construction algorithm

Ukkonen [13] proposed an algorithm to incrementally construct the suffix tree of a single text. His algorithm can easily be extended to incrementally construct the suffix tree for multiple texts in the semi-online setting.

Let U be a semi-online sequence of N update operators such that the last update operator for each k $(1 \leq k \leq K)$ is $(k, \$_k)$, where $\$_k$ is a special end-marker for the kth text in the collection. For ease of notation, $\mathcal{T}' = \mathcal{T}_{u[1..i-1]}$ and $\mathcal{T} = \mathcal{T}_{u[1..i]}$, Also, assume that we have already constructed $STree(\mathcal{T}')$ and that the next update operator is $U[i] = (k, a)$. Thus a new character a is appended to the kth text T_k of $\mathcal{T}'$, and the kth text of $\mathcal{T}$ becomes $T_k a$.

As in the case of semi-online DAWG construction, the suffixes of $T_k a$ are inserted in decreasing order of length. The Type-1 suffixes are maintained as follows. Let s be any suffix of T_k which is represented by a leaf of $STree(\mathcal{T}')$. Since s is a non-repeating suffix of T_k in $\mathcal{T}'$, sa is a non-repeating suffix of $T_k a$ in $\mathcal{T}$, which implies that sa will also be a leaf of $STree(\mathcal{T})$. Based on this observation, the label of the in-coming edge of s is represented by a triple $\langle k, b, \infty \rangle$ called an *open edge*, where b is the beginning position of the label of the in-coming edge in the kth text. This way, every existing leaf will then be automatically extended. Hence, updating $STree(\mathcal{T}')$ to $STree(\mathcal{T})$ reduces to inserting the Type-2 suffixes of $T_k a$. For this sake, the algorithm maintains an invariant which indicates the locus of $x = lrs_{\mathcal{T}'}(T_k)$ on $STree(\mathcal{T}')$ called the *active point*. Since x can be an implicit node, the algorithm maintains the canonical reference (v, c, ℓ) to x. For convenience, if x is an explicit node, then let its canonical reference be $(x, \varepsilon, 0)$. The update starts from the current active point x represented by its canonical reference pair, and the Type-2 suffixes of $T_k a$ are inserted in decreasing order of length, by using the chain of (virtual) suffix links. There are two cases:

I. If it is possible to go down from x with character a, then no updates to the tree topology are needed. The new active point is xa, and the reference to xa is made canonical if necessary. The update ends.

II. If it is impossible to go down from x with character a, then we create a new leaf. Let j be the beginning position of the suffix of $T_k a$ which corresponds to this new leaf. The following procedure is repeated until Case I happens.

1. If the active point x is on an explicit node, then a new leaf node s is created as a new child of x, with its incoming edge labeled by $\langle k, b, \infty \rangle$, where $b = |T_k a| - |x| + 1$. The active point x is updated to $\mathit{slink}(x)$.
2. If the active point x is on an implicit node, then x becomes explicit in this step. A new leaf node s is created as a new child of x with its incoming edge labeled by $\langle k, b, \infty \rangle$. Since the suffix link of the new explicit node x does not yet exist, we simulate the suffix link traversal as follows: Let (v_j, c_j, ℓ_j) be the canonical reference to x. First, we follow the suffix link $\mathit{slink}(v_j)$ of v_j, and then go down along the path of length ℓ_j from $\mathit{slink}(v_j)$ starting with character c_j. Let this locus be x'. Let v_{j+1} be the longest explicit node in this path.
   - **(i)** If $|v_{j+1}| = |x'|$, then we firstly create the new suffix link $\mathit{slink}(x) = v_{j+1}$ for the new explicit node x. The active point x is updated to x' and is represented by canonical reference $(v_{j+1}, \varepsilon, 0)$.
   - **(ii)** If $|v_{j+1}| < |x'|$, then the next active point is implicit. The active point x is updated to x' and is represented by canonical reference $(v_{j+1}, c_{j+1}, \ell_{j+1})$. The suffix link of x will be set to x' when x' becomes explicit in the next step.

The most expensive case is II-b-(ii). Since the path from v_{j+1} to x' contains at most $\ell_j - \ell_{j+1}$ explicit nodes, it takes $O((\ell_j - \ell_{j+1} + 1) \log \sigma)$ time to locate the next active point x' (note $\ell_j - \ell_{j+1} \geq 0$ holds). All the other operations take $O(\log \sigma)$ time. Hence, the total cost to insert all leaves (suffixes) for the kth text is $O(\sum_{j=1}^{N_k} (\ell_j - \ell_{j+1} + 1) \log \sigma) = O(N_k \log \sigma)$, where N_k is the final length of the kth text. Thus the amortized time cost for each leaf (suffix) for the kth text is $O(\log \sigma)$. Overall, it takes a total of $O(N \log \sigma)$ time to construct $\mathit{STree}(\mathcal{T}_U)$ for a semi-online sequence U of update operators. The space requirement is $O(N)$.

4 Fully-online DAWG construction algorithm

We can easily extend Blumer et al.'s semi-online DAWG construction algorithm to the fully-online setting. Let U be a fully-online sequence of N update operators. Our fully-online algorithm maintains the active point v_k for *every* growing text T_k in the collection, at any step of the algorithm. Now, assume that we have already constructed $\mathit{DAWG}(\mathcal{T}')$, where $\mathcal{T}' = \mathcal{T}_{U[1..i-1]}$ for $1 \leq i \leq N$. Let $U[i] = (k, a)$, and we are updating $\mathit{DAWG}(\mathcal{T}')$ to $\mathit{DAWG}(\mathcal{T})$, where $\mathcal{T} = \mathcal{T}_{U[1..i]}$. The update starts from the active point $v_k = [T_k]_{\mathcal{T}'}$, exactly in the same way as was described in Section 3. The total cost to update $\mathit{DAWG}(\mathcal{T}')$ to $\mathit{DAWG}(\mathcal{T})$ is again $O(q \log \sigma)$, where q is the total number of nodes, edges, and suffix links which were introduced in this update. Since the total size of $\mathit{DAWG}(\mathcal{T})$ is $O(N)$, the amortized cost for this update is again $O(\log \sigma)$. By the above arguments, we obtain the following theorem.

▶ **Theorem 1.** *Given a fully-online sequence U of N update operators for a collection of K texts, we can update $\mathit{DAWG}(\mathcal{T}_{U[1..i]})$ for $i = 1, \ldots, N$ in a total of $O(N \log \sigma)$ time and $O(N)$ space.*

Assume for now that each text T_k in a collection $\mathcal{T}$ begins with a special character $\#_k$ which does not appear elsewhere in $\mathcal{T}$. Then, the tree of the (reversed) suffix links of $\mathit{DAWG}(\mathcal{T})$ forms the suffix tree $\mathit{STree}(\overline{\mathcal{T}})$ for the collection $\overline{\mathcal{T}} = \{\overline{T_1}, \ldots, \overline{T_K}\}$ of the reversed texts of $\mathcal{T}$ [4]. Hence, the next corollary follows from Theorem 1, which gives *right-to-left* fully-online suffix tree construction.

▶ **Corollary 2.** *Given a fully-online sequence U of N update operators for a collection of K texts, we can update $STree(\overline{\mathcal{T}_{U[1..i]}})$ for $i = 1, \ldots, N$ in a total of $O(N \log \sigma)$ time and $O(N)$ space.*

5 Fully-online suffix tree construction algorithm

5.1 Difficulties in fully-online construction of suffix trees

Unlike the case with DAWGs, it is not easy to extend Ukkonen's semi-online suffix tree construction algorithm to our left-to-right fully-online setting, because:

(A) Let $U[i] = (k, a)$ which updates the current kth text T_k to $T_k a$, and assume that we have just constructed $STree(\mathcal{T}_{U[1..i]})$. Recall that we defined the initial locus of the active point for $T_k a$ on $STree(\mathcal{T}_{U[1..i]})$ to be the longest repeating suffix of $T_k a$ in $\mathcal{T}_{U[1..i]}$. However, since U is fully-online, any other text T_h $(h \neq k)$ in the collection would be updated by following update operators $U[r]$ with $r > i$. Then, the longest repeating suffix of $T_k a$ in $\mathcal{T}_{U[1..r]}$ can be much longer than that of $T_k a$ in $\mathcal{T}_{U[1..i]}$. In other words, some Type-1 suffixes of $T_k a$ in $\mathcal{T}_{U[1..i]}$ can become of Type-2 in $\mathcal{T}_{U[1..r]}$. What is worse, updating T_h can affect the longest repeating suffix of any other text in the collection as well. If we maintain all these active points naïvely, it takes $O(KN \log \sigma)$ time.

(B) Even if we somehow manage to efficiently maintain the active point for each text in the collection, there remains another difficulty. Let j be the beginning position of the longest repeating suffix of $T_k a$ in $\mathcal{T}_{U[1..i]}$, and let (v_j, c_j, ℓ_j) be the canonical reference to this suffix. Let $U[i'] = (k, a')$ be the first update operator in U which updates the kth text after $U[i] = (k, a)$. Let (v'_j, c'_j, ℓ'_j) be the canonical reference to the longest repeating suffix of $T_k a$ in $\mathcal{T}_{U[1..i']}$, which is the "real" initial active point where insertion of the Type-2 suffixes should start at this i'th step. By the property of suffix trees $\ell'_j \geq \ell_j$ holds, and what is worse, this length ℓ'_j is unbounded by the number of Type-2 suffixes inserted at this i'th step. Thus, the amortization technique we used for the semi-online construction does not work in the fully-online setting.

(C) The phenomenon mentioned in Difficulty A also causes a problem of how to represent the labels of the in-coming edges to the leaves. Assume that we created a new leaf w.r.t. an update operator (k, a), and let $\langle k, b_k, \infty \rangle$ be the triple representing the label of the in-coming edge to the leaf, where b_k is the beginning position of the edge label in the kth text. It corresponds to a Type-1 suffix of the kth text, but the leaf can later be extended by another growing text T_h. Then, the triple $\langle k, b_k, \infty \rangle$ has to be updated to $\langle h, b_h, \infty \rangle$, where b_h is the beginning position of the edge label in the hth text. Notice that this update may happen repeatedly.

5.2 Constructing suffix trees with the aid of DAWGs

We utilize DAWGs to overcome Difficulties A, B and C in fully-online construction of suffix trees. Namely, we construct $STree(\mathcal{T})$ in tandem with $DAWG(\mathcal{T})$.

A high-level description of our algorithm is as follows. We insert the Type-2 suffixes of $T_k a$ in *increasing order* of length, starting from the locus of the longest Type-3 suffix of $T_k a$. The idea of inserting the Type-2 suffixes in increasing order of length was also used by Breslauer and Italiano [5], for quasi real-time left-to-right construction of the suffix tree for a single text. To efficiently find the locus where the next longer Type-2 suffix should be inserted in the tree from the locus where the last Type-2 suffix was inserted, we introduce a simpler amortized variant of the *suffix tree oracle* of Fischer and Gawrychowski [10, 9].

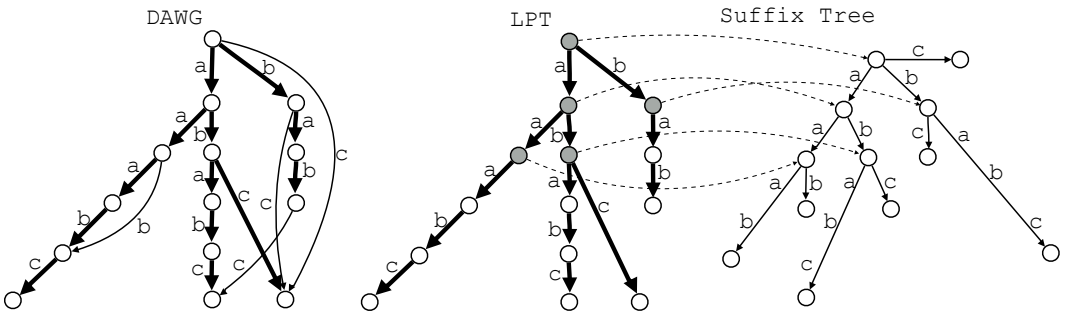


Figure 2 Illustration for $DAWG(\mathcal{T})$, $LPT(\mathcal{T})$, and $STree(\mathcal{T}')$, where $\mathcal{T}' = \{T_1 = \texttt{aaab}, T_2 = \texttt{ababc}, T_3 = \texttt{bab}\}$ and $\mathcal{T} = \{T_1\texttt{c}, T_2, T_3\}$. The bold solid arrows represent the primary edges of $DAWG(\mathcal{T})$, the gray nodes are the marked nodes of $LPT(\mathcal{T})$, and the dashed arrows represent the links between the marked nodes of $LPT(\mathcal{T})$ and the corresponding branching nodes of $STree(\mathcal{T}')$. $lrs_{\mathcal{T}}(T_1\texttt{c}) = \texttt{abc}$, and hence we perform an NMA query from node `abc` on $LPT(\mathcal{T})$, obtaining node `ab`. We then access the suffix tree node `ab` using the pointer from $LPT(\mathcal{T})$, and obtain the locus of `abc` on $STree(\mathcal{T}')$.

These will overcome Difficulties A and B. To overcome Difficulty C, we introduce new *lazy representation* of the labels of edges leading to the leaves. The next is a key lemma.

▶ **Lemma 3.** *We can compute, in amortized $O(\log \sigma)$ time, the canonical reference to the longest Type-3 suffix $lrs_{\mathcal{T}}(T_k a)$ of $T_k a$ on $STree(\mathcal{T}')$, using a data structure which requires space linear in the total length of the texts in $\mathcal{T}$.*

Proof. We introduce the *longest path tree* of $\mathcal{T}'$, denoted $LPT(\mathcal{T}')$, which is the spanning tree of $DAWG(\mathcal{T}')$ consisting only of the primary edges of $DAWG(\mathcal{T}')$. Every node of $LPT(\mathcal{T}')$ is marked iff its corresponding node on $DAWG(\mathcal{T}')$ is branching. Every marked node of $LPT(\mathcal{T})$ is linked to its corresponding node of $STree(\mathcal{T}')$ which is also branching by Fact 1 (see also Fig. 2). $LPT(\mathcal{T}')$ is enhanced with the *nearest marked ancestor* (*NMA*) data structure of Westbrook [16], which supports the following operations in amortized $O(1)$ time using linear space: 1) find the NMA of any node; 2) insert an unmarked node; 3) mark an unmarked node.

When $DAWG(\mathcal{T}')$ is updated to $DAWG(\mathcal{T})$, at most two new primary edges are introduced to $DAWG(\mathcal{T})$, one for the new sink and one for the split node. We insert these new edges to $LPT(\mathcal{T}')$ and obtain $LPT(\mathcal{T})$. Because of these new edges, at most two non-branching nodes of $DAWG(\mathcal{T}')$ can become branching in $DAWG(\mathcal{T})$. We mark their corresponding nodes in $LPT(\mathcal{T})$, and link them to the corresponding suffix tree nodes after we have constructed $STree(\mathcal{T})$. This is because the corresponding nodes of $STree(\mathcal{T}')$ are still non-branching.

We use $LPT(\mathcal{T})$ to quickly move from the DAWG to the suffix tree (see Fig. 2 for a concrete example). Since $lrs_{\mathcal{T}}(T_k a)$ is the longest in $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$, there always exists a node y of $LPT(\mathcal{T})$ which represents $lrs_{\mathcal{T}}(T_k a)$. We conduct an NMA query from y on $LPT(\mathcal{T})$, and let v be the NMA of y. Let $\ell = |y| - |v|$, and let c be the label of the first edge in the path from v to y. We move from v to its corresponding node x in $STree(\mathcal{T}')$. Then, (x, c, ℓ) is a reference to $lrs_{\mathcal{T}}(T_k a)$ in $STree(\mathcal{T}')$. Since v is the NMA of y in $LPT(\mathcal{T})$, and since updating T_k to $T_k a$ does not explicitly insert any suffix shorter than $lrs_{\mathcal{T}}(T_k a)$, this reference is canonical by Fact 1.

Clearly the total size of the above data structures is linear in the total length of the texts in $\mathcal{T}$. We analyze the time complexity. Recall Case 2 when updating $DAWG(\mathcal{T}')$ to

$DAWG(\mathcal{T})$. At the end of the update, we find (or create) in amortized $O(\log \sigma)$ time the node of $DAWG(\mathcal{T})$ which represents $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$. Hence we can find node $y = lrs_{\mathcal{T}}(T_k a)$ in amortized $O(\log \sigma)$ time. Updating $LPT(\mathcal{T}')$ to $LPT(\mathcal{T})$ takes $O(\log \sigma)$ time. Inserting a new node and querying an NMA from a given node takes amortized $O(1)$ time. We can link a new marked node of $LPT(\mathcal{T})$ to the corresponding new branching node of $STree(\mathcal{T})$ in $O(1)$ time, since we can remember this new branching node when updating $STree(\mathcal{T}')$ to $STree(\mathcal{T})$. Hence, the amortized bound is $O(\log \sigma)$. ◀

To find the insertion point of the shortest Type-2 suffix from the longest Type-3 suffix $lrs_{\mathcal{T}}(T_k a)$, and to insert the Type-2 suffixes of $T_k a$ in increasing order of length, we maintain the labeled reversed suffix links for each explicit node of the suffix tree. Namely, if $slink(bv) = v$ for two nodes bv, v with $v \in \Sigma^*$ and $b \in \Sigma$, let $rslink_b(v) = bv$. We leave $rslink_b(v)$ undefined if bv is not a substring of any text in the collection, or if node bv is implicit in the suffix tree.

A *suffix tree oracle* for a suffix tree S is a data structure which efficiently answers the following query: given a pair (v, b) of a node v of S and a character $b \in \Sigma$, return the nearest ancestor u of v for which $rslink_b(u)$ is defined. The state-of-the-art suffix tree oracle by Fischer and Gawrychowski [10, 9] answers queries and supports updates in worst-case $O(\log \log n + (\log \log \sigma)^2 / \log \log \log \sigma)$ time each, using $O(n)$ space, where n is the number of leaves in S. We present a simpler suffix tree oracle with amortized $O(\log \sigma)$ bound.

▶ **Lemma 4.** *For a suffix tree with n leaves, there is a suffix tree oracle of size $O(n)$ which answers each query in amortized $O(\log \sigma)$ time. It takes amortized $O(\log \sigma)$ time to update this suffix tree oracle, per insertion of a new leaf or a new suffix link to the suffix tree.*

Proof. (Sketch) We follow the approach by Fischer and Gawrychowski [10, 9]. The $\log \log n$ term in the running time of their suffix tree oracle is due to the fringe nearest marked ancestor data structure by Breslauer and Italiano [5], which answers each NMA query in a special case in worst case $O(\log \log n)$ time. It is possible to replace the fringe nearest marked ancestor data structures with the NMA data structures of Westbrook [16], so the time cost for each NMA query is amortized to $O(1)$. The other $(\log \log \sigma)^2 / \log \log \log \sigma$ term is due to fast predecessor data structures for integer alphabets. Since our alphabet is more general, we use balanced search trees with $O(\log \sigma)$-time operations. Hence our bound is $O(\log \sigma)$ amortized. A complete proof can be found in a full version of this paper [12]. ◀

To overcome Difficulty C, we employ lazy maintenance for leaves, namely, we maintain only the *first character* of the label of every edge leading to a leaf. On the other hand, we eagerly maintain the whole label of every edge leading to an internal explicit node.

▶ **Lemma 5.** *The lazy representation of the in-coming edges of leaves allows for updating the suffix tree in amortized $O(\log \sigma)$ time per insertion of a new leaf.*

Proof. Let $U[i] = (k, a)$ and $\mathcal{T} = \mathcal{T}_{U[1..i]}$ as previously. Let xa be a Type-2 suffix of the extended text $T_k a$ that will be inserted to the suffix tree. Using the suffix tree oracle of Lemma 4, we obtain the canonical reference (v, c, ℓ) to x from which a new leaf for the suffix xa is to be inserted.

The difficult case is when x is on the edge e from v to a leaf and $\ell \geq 2$, since we only know the first character c of the label of e. We create a new internal node x on e, and create a new leaf as a child of x and its in-coming edge labeled with the first character a. We can determine the label of the in-coming edge of the new internal explicit node x as follows. Let y be the node of $LPT(\mathcal{T})$ which corresponds to the node $[v]_{\mathcal{T}}$ of $DAWG(\mathcal{T})$,

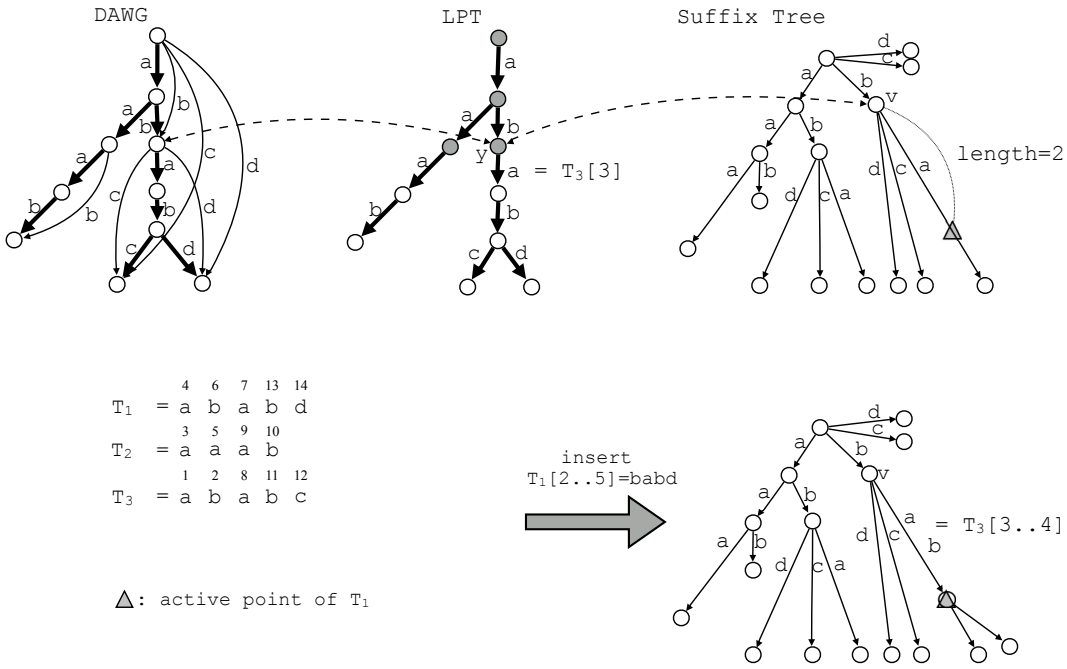


Figure 3 Illustration of how to determine the label of the in-coming edge of a new internal explicit node which is created on an edge leading to an existing leaf. Let $\mathcal{T}' = \{T_1 = \texttt{abab}, T_2 = \texttt{aaab}, T_3 = \texttt{ababc}\}$, and $\mathcal{T} = \{T_1\texttt{d}, T_2, T_3\}$. Now we are inserting a new leaf w.r.t. Type-2 suffix $\texttt{babd}$ of $T_1\texttt{d}$. The canonical reference to the insertion point of this suffix is $(\texttt{b}, \texttt{a}, 2)$, and hence we create a new internal node on the middle of the out-going edge of node $\texttt{b}$ whose edge label begins with $\texttt{a}$. Now, since $\mathit{long}([\texttt{b}]_{\mathcal{T}}) = \texttt{ab}$, we access the LPT node $y = \texttt{ab}$. Since the label $\texttt{a}$ of the out-going edge of y in $\mathit{LPT}(\mathcal{T})$ is now represented by pair $\langle 3, 3\rangle$, we can label the new suffix tree edge leading to the new internal node by $\langle 3, 3, 3 + 2 - 1\rangle = \langle 3, 3, 4\rangle$.

namely $y = \mathit{long}([v]_{\mathcal{T}})$. We represent the label of each edge of $\mathit{LPT}(\mathcal{T})$ by a pair of the text id and the position of the character in the text of that id. Let $\langle h, j\rangle$ be the label of the out-going edge of node y of $\mathit{LPT}(\mathcal{T})$ such that $T_h[j] = c$. Since we insert the Type-2 suffixes of $T_k a$ in increasing order of length, the path in $\mathit{LPT}(\mathcal{T})$ of length ℓ starting with this edge from y is non-branching. Thus, we can label the in-coming edge of the suffix tree by triple $\langle h, j, j + \ell - 1\rangle$. See also Fig. 3.

While updating $\mathit{DAWG}(\mathcal{T}')$ to $\mathit{DAWG}(\mathcal{T})$, we have visited the node $[x]_{\mathcal{T}}$. We can obtain node y on $\mathit{LPT}(\mathcal{T})$ by an NMA query from node $\mathit{long}([x]_{\mathcal{T}})$, and associate to y each Type-2 suffix xa of $T_k a$ whose length is in range $[s + 1, l + 1]$, where s and l are the lengths of the shortest and longest members of $[x]_{\mathcal{T}}$, respectively. As we insert the Type-2 suffixes of $T_k a$ to the suffix tree in increasing order of length, for each Type-2 suffix xa the time cost to access its corresponding node y on $\mathit{LPT}(\mathcal{T})$ is $O(\log \sigma)$ amortized. It takes amortized $O(\log \sigma)$ time to query the suffix tree oracle by Lemma 4. All the other operations take $O(1)$ time each. ◀

Assume we are searching a growing text collection $\mathcal{T}$ for a given pattern P. If we stuck on the parent node u of a leaf in $\mathit{STree}(\mathcal{T})$ due to our lazy leaf representation, then we can move to the DAWG node which corresponds to the parent node u via $\mathit{LPT}(\mathcal{T})$, and continue searching for P on $\mathit{DAWG}(\mathcal{T})$. This way we can find the locus of P on $\mathit{STree}(\mathcal{T})$ in optimal $O(M \log \sigma)$ time, where $M = |P|$. Also, since the tree topology is correctly maintained with

our lazy leaf representation, semi-dynamic NMA [16], LCA [7], and LA [1] queries can be correctly supported in $O(1)$ time on our suffix tree representation.

Finally, we obtain the main result of this section.

▶ **Theorem 6.** *Given a fully-online sequence U of N update operators for a collection of K texts, we can update $STree(\mathcal{T}_{U[1..i]})$ for $i = 1, \ldots, N$ in a total of $O(N \log \sigma)$ time and $O(N)$ space.*

After the whole U has been processed, we determine the triples representing the entire labels of the in-coming edges of all leaves of $STree(\mathcal{T}_U)$ in a total of $O(N)$ time. We can then discard $DAWG(\mathcal{T}_U)$ and $LPT(\mathcal{T}_U)$.

6 Conclusions and open problems

The main contribution of this paper is an $O(N \log \sigma)$-time algorithm to maintain the suffix tree for a text collection in the left-to-right fully-online setting, where N and σ are the total text length and the alphabet size, respectively. The key was a non-trivial use of the DAWG.

There are interesting open problems for the left-to-right fully-online suffix tree construction:

1. Is it possible to efficiently maintain *complete* labels of the edges leading to the leaves?
2. Our bound is amortized, namely, for each new character our algorithm takes $O(\log \sigma)$ amortized time. Is it possible to de-amortize it, e.g. by using techniques in [5, 9, 10]?

References

1 Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *ICALP 2000*, pages 73–84, 2000.

2 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS 2002*, pages 1–16, 2002.

3 Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel Seiferas. The smallest automaton recognizing the subwords of a text. *TCS*, 40:31–55, 1985.

4 Anselm Blumer, Janet Blumer, David Haussler, Ross Mcconnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.

5 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.

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

7 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. In *SODA 1999*, pages 235–244, 1999.

8 Paolo Ferragina and Roberto Grossi. Improved dynamic text indexing. *J. Algorithms*, 31(2):291–319, 1999.

9 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. *CoRR*, abs/1302.3347, 2013.

10 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *CPM 2015*, pages 160–171, 2015.

11 Eamonn J. Keogh, Stefano Lonardi, and Bill Yuan-chi Chiu. Finding surprising patterns in a time series database in linear time and space. In *KDD 2002*, pages 550–556, 2002.

12 Takuya Takagi, Shunsuke Inenaga, and Hiroki Arimura. Fully-online construction of suffix trees and dawgs for multiple texts. *CoRR*, abs/1507.07622, 2015.

13 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

14 Yilun Wang, Yu Zheng, and Yexiang Xue. Travel time estimation of a path using sparse trajectories. In *KDD 2014*, pages 25–34, 2014.

15 Peter Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.

16 Jeffery Westbrook. Fast incremental planarity testing. In *ICALP 1992*, pages 342–353, 1992.

Linear-time Suffix Sorting – A New Approach for Suffix Array Construction

Uwe Baier

Institute of Theoretical Computer Science, Ulm University
D-89069 Ulm, Germany
uwe.baier@uni-ulm.de

Abstract

This paper presents a new approach for linear-time suffix sorting. It introduces a new sorting principle that can be used to build the first non-recursive linear-time suffix array construction algorithm named GSACA. Although GSACA cannot keep up with the performance of state of the art suffix array construction algorithms, the algorithm introduces a couple of new ideas for suffix array construction, and therefore can be seen as an 'idea collection' for further suffix array construction improvements.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Suffix array, sorting algorithm, linear-time

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.23

1 Introduction

The suffix array is an elementary data structure used in string processing as well as in data compression. Introduced by Manber and Myers in 1990 [11], the suffix array nowadays finds application in dozens of different areas. Constructing a suffix array from a given string unfortunately turns out to be a computationally hard task; despite the existence of linear-time algorithms for suffix array construction, some super-linear algorithms still achieve better results in practice.

As data grows bigger and bigger, 'optimal' suffix array construction algorithms (SACAs) nowadays still stay an area of great interest. According to a survey paper of Puglisi et al. [19], an 'optimal' SACA fulfils three requirements: First, an algorithm should run in asymptotic minimal worst-case-time, linear-time in an optimal way. Second, an algorithm should run fast in practice, too. Finally, the algorithm should consume as less extra space in addition to the text and the suffix array as possible, a constant amount optimally.

Presently, no SACA is able to meet all of those requirements in an optimal way. Our contribution towards this goal will be the presentation of a new design principle for suffix array construction, resulting in the first non-recursive linear-time suffix array construction algorithm. Although the new algorithm is not able to fulfil all requirements of optimal suffix array construction, it presents a new approach for suffix array construction, and therefore is interesting from a theoretical point of view.

Overview This paper will be organised as follows: Section 2 contains a short introduction to suffix arrays and basic definitions. Section 3 presents the new sorting principle along with an introductory example, before Section 4 lists the new algorithm with explanations of technical details. Section 5 contains performance analyses of the new algorithm, before Section 6 summarises the results and gives an outline for future work.

Related Work The suffix array first was described in 1990 by Manber and Myers [11] as a space-saving alternative to suffix trees [21].

Then, in 2003, four linear-time[1] SACAs were contemporary introduced by Kim et al. [8], Kärkkäinen and Sanders [7], Ko and Aluru [10] and Hon et al. [6], before Joong Chae Na introduced another linear-time SACA in 2005 [15]. Two algorithms stood out: the *Skew Algorithm* by Kärkkäinen and Sanders [7] because of its elegance, as well as the algorithm by Ko and Aluru [10] because of its good performance in practice.

Later on, in 2009, Nong et al. presented two new algorithms using the induced sorting principle [17, 18] as an improvement to the algorithm by Ko and Aluru. One of those algorithms, called *SA-IS* [17], was able to outperform most of other existing SACAs [14] while guaranteeing asymptotic linear runtime and almost optimal space requirements. In the meantime, performance of *SA-IS* was further improved while decreasing the required workspace to an only alphabet-dependent linear term [16]. Consequently, variants of the *SA-IS* algorithm serve as best linear-time SACAs known at the moment.

2 Preliminaries

Let Σ be a totally ordered set (alphabet) of elements (characters). A string S of length n over alphabet Σ is a finite sequence of n characters originating from Σ. The empty string with length 0 is denoted by ε.

Let i and j be two integers in range $[1, n]$. We denote by

- $S[i]$ the i-th character of S.
- $S[i..j]$ the substring of S starting at the i-th and ending at the j-th position.

  We state $S[i..j] = \varepsilon$ if $i > j$, and define $S[i..j+1) = S[i..j]$.
- S_i the suffix of S starting at the i-th position, i.e. $S_i = S[i..n]$.

Furthermore, we call S a *nullterminated* string if $\$ \in \Sigma$, $\$ < c$ for all $c \in \Sigma \setminus \{\$\}$, and $\$$ occurs exactly once in S, at the end of the string. First, a definition of the suffix array shall be presented. Additionally, next lexicographically smaller suffixes are required.

▶ **Definition 1.** Let Σ be an alphabet, S be a string of length n over alphabet Σ and T be a string of length m over alphabet Σ. We write $S <_{\mathsf{lex}} T$ and say that S is *lexicographically smaller* than T, if one of the following conditions holds:

- There exists an i $(1 \leq i \leq \min\{n, m\})$ with $S[i] < T[i]$ and $S[1..i) = T[1..i)$.
- S is a *proper prefix* of T, i.e. $n < m$ and $S[1..n] = T[1..n]$.

▶ **Definition 2.** Let S be a nullterminated string of length n. The *suffix array* SA of S is a permutation of integers in range $[1, n]$ satisfying $S_{\mathsf{SA}[1]} <_{\mathsf{lex}} S_{\mathsf{SA}[2]} <_{\mathsf{lex}} \cdots <_{\mathsf{lex}} S_{\mathsf{SA}[n]}$. The *inverse suffix array* ISA is the inverse permutation of SA.

▶ **Definition 3.** Let S be a nullterminated string of length n, and let i be an integer in range $[1, n)$. Then, by $\widehat{i}$ we denote the position of the next lexicographically smaller suffix of S_i, i.e. $\widehat{i} := \min\{\ j \in [i \ldots n] \mid S_j <_{\mathsf{lex}} S_i\ \}$. Also, we define $\widehat{n} := n + 1$ for the last suffix of S.[2]

An example of these definitions can be found in Table 1.

[1] Super-linear-time SACAs are not object of interest here; we refer to the survey paper of Puglisi et al. [19] for more information about them.

[2] One can think of this as follows: if we define an imaginary empty last suffix $S_{n+1} := \varepsilon$, then S_{n+1} is a proper prefix of S_n, so S_{n+1} is the next smaller suffix of S_n.

Table 1 Suffix array and next lexicographically smaller suffixes of $S =$ `graindraining$`.

i	$\mathsf{SA}[i]$	$\widehat{\mathsf{SA}[i]}$	$S_{\mathsf{SA}[i]}$	$S[\mathsf{SA}[i]..\widehat{\mathsf{SA}[i]})$
1	14	15	`$`	`$`
2	3	14	`aindraining$`	`aindraining`
3	8	14	`aining$`	`aining`
4	6	8	`draining$`	`dr`
5	13	14	`g$`	`g`
6	1	3	`graindraining$`	`gr`
7	4	6	`indraining$`	`in`
8	11	13	`ing$`	`in`
9	9	11	`ining$`	`in`
10	5	6	`ndraining$`	`n`
11	12	13	`ng$`	`n`
12	10	11	`ning$`	`n`
13	2	3	`raindraining$`	`r`
14	7	8	`raining$`	`r`

3 Algorithmic Idea

Within this Section, the algorithmic idea of the new algorithm will be presented. The main idea is to split the suffix array construction in two phases.

In a first phase, suffixes are divided into suffix groups as if each suffix S_i consists only of the string $S[i..\widehat{i})$: If $S[i..\widehat{i}) = S[j..\widehat{j})$ holds for two suffixes S_i and S_j, then they belong to the same group, otherwise to different groups. For any group $\mathcal{G}$ containing a suffix S_i, we denote the string $S[i..\widehat{i})$ as the *group context* of $\mathcal{G}$. In addition to the division of suffixes, the groups itself also will be ordered by comparing their group contexts. When comparing suffix groups by their contexts, the terms 'lower group' and 'higher group' will be used rather than the terms 'smaller' or 'larger', because groups are sets, and the latter both terms usually refer to set sizes, not to lexicographic comparison.

Afterwards, in a second phase, this group structure can be used to compute the suffix array. By iterating over the suffix array in ascending lexicographic order and completing the contexts of suffixes such that only groups with a single suffix remain, the desired order of suffixes can be obtained. A sketch of the principle can be found in Algorithm 1.

First, let's clarify the correctness of the principle by some argumentation. Assume that before the i-th iteration of the outer loop in Phase 2 (lines 4 to 8) all entries $\mathsf{SA}[1] \cdots \mathsf{SA}[i]$ were computed correctly. Then, within the i-th iteration, each further computed SA-entry is correct: Let j be any index with $\widehat{j} = \mathsf{SA}[i]$. Assume that an index k from the same group exists such that $S_k <_{\mathsf{lex}} S_j$. Because $\text{group}(k) = \text{group}(j)$, by the sorting in Phase 1, $S[j..\widehat{j}) = S[k..\widehat{k})$ holds, so $S_{\widehat{k}} <_{\mathsf{lex}} S_{\widehat{j}}$ must hold. Because of the ascending iteration order of the outer loop in Phase 2, $\widehat{k}$ must have been processed in one of the previous $i-1$ iterations. Within this iteration, the index k was processed in the inner loop of Phase 2, and thus has been removed from its group in line 8, $\text{group}(k) \neq \text{group}(j)$, contradiction. For the same reason, and because of the group order computed in Phase 1 (line 2), exactly those suffixes S_k with $\text{group}(k) < \text{group}(j)$ must be lexicographically smaller than S_j, so j is correctly placed into the suffix array in line 7.

Now we know that all entries are placed correctly to SA, but it remains to show that the suffix array is filled entirely. Therefore, consider the point in time after the i-th iteration of the outer loop in Phase 2, and let S_j be the lexicographically $i+1$-th smallest suffix.

Algorithm 1 Suffix array construction for a given nullterminated string S of length n.

Phase 1: divide suffixes into groups

order all suffixes of S into groups: Let S_i and S_j be two suffixes.
Then, group(i) = group(j) if and only if $S[i..\widehat{i}) = S[j..\widehat{j})$.

order the suffix groups by their contexts: Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two groups, $i \in \mathcal{G}_1$, $j \in \mathcal{G}_2$. Then, $\mathcal{G}_1 < \mathcal{G}_2$ if and only if $S[i..\widehat{i}) <_{\mathsf{lex}} S[j..\widehat{j})$.

Phase 2: construct suffix array from groups

$\mathsf{SA}[1] \leftarrow n$

for $i = 1$ **up to** n **do**

  for all suffixes S_j with $\widehat{j} = \mathsf{SA}[i]$ **do**

    let sr be the number of suffixes placed in lower groups,
    i.e. $sr := |\{\ s \in [1 \dots n] \mid \text{group}(s) < \text{group}(j)\ \}|$.

    $\mathsf{SA}[sr + 1] \leftarrow j$

    remove j from its current group and put it in a new group
    placed as immediate predecessor of j's old group.

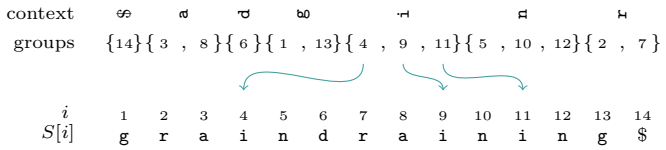


Figure 1 Initial group division for the suffixes of $S =$ `graindraining`\$, where links from the group with context `i` to the text are shown. Groups are ordered by their context from left to right.

Because $S_{\widehat{j}} <_{\mathsf{lex}} S_j$ holds by the definition of next lexicographically smaller suffixes, the index $\widehat{j}$ must have been processed by the outer loop of Phase 2 already, and thus, the index j must have been placed to the suffix array correctly, $\mathsf{SA}[i + 1] = j$ holds.

The argumentation shows that the principle works correctly, but there are still a lot of issues remaining. But instead of presenting a more detailed algorithm directly, an introductory example will be presented, to bridge the gap between the sorting principle and the final algorithm.

3.1 Example: Phase 1

Within Phase 1, suffixes have to be divided into groups. More specifically, all suffixes S_i sharing the same prefix $S[i..\widehat{i})$ must belong to the same group, while the groups itself must be sorted by their contexts, see Algorithm 1. To accomplish this task, in an initial step, suffixes are split into groups by their first character. Also, the groups are sorted by their initial context, see Figure 1 for an example.

To obtain the requested group order, all groups are processed in descending order (i.e. from highest to lowest group), repeating the following steps for each group $\mathcal{G}$:

1. For each index $i \in \mathcal{G}$ compute its *prev pointer* prev(i), the previous index placed in a lower group, i.e. prev$(i) := \max\{\ j \in [1..i] \mid \text{group}(j) < \text{group}(i)\ \}$.
2. Split the set $\mathcal{P} := \{\ \text{prev}(i) \mid i \in \mathcal{G}\ \}$ into subsets $\mathcal{P}_1, \dots, \mathcal{P}_k$ such that $i, j \in \mathcal{P}_q \Leftrightarrow i, j \in \mathcal{P}$ and group(i) = group(j) for any subset $\mathcal{P}_q$.
3. For each subset $\mathcal{P}_q$, remove the indices of $\mathcal{P}_q$ from their old group and put them to a new group, placed as immediate successor of their old group.

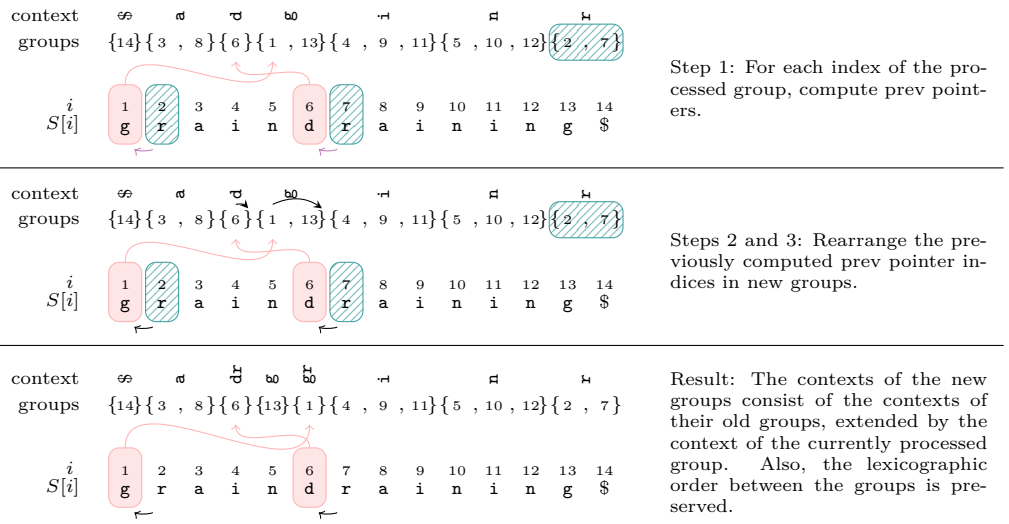


Figure 2 First iteration step of Phase 1 applied to the string $S = \texttt{graindraining\$}$.

Such processing causes an effect quite similar to the *prefix doubling* technique: Each time when indices of a group are removed and collected in a new group (step 3), the context of the new group consists of the context of the old group, extended by the context of the currently processed group, see Figure 2 for an example.

To clarify why context extensions take place, let i be an index and i_c be the first index following i such that i is not reachable using the prev pointer chain starting at i_c, i.e. $i_\mathsf{c} := \min\{\, j \in [i+1..n+1] \mid i \notin \{j, \mathrm{prev}(j), \mathrm{prev}(\mathrm{prev}(j)), \ldots\} \,\}$.[3] As one can show (see [2]), during the processing of groups in Phase 1, $\mathrm{group}(i) = \mathrm{group}(j) \Leftrightarrow S[i..i_\mathsf{c}) = S[j..j_\mathsf{c})$ holds for two indices i and j, so the string $S[i..i_\mathsf{c})$ meets our imagination of group contexts. However, coming back to the above mentioned context extensions, we'll take a closer look onto the steps performed when processing a group. In Step 1, prev pointers are computed. Let i be an index of the processed group, and let $p := \mathrm{prev}(i)$ be its prev pointer. By the definition of a prev pointer (see Step 1), all indices j between p and i ($p < j < i$) are placed in higher groups than p and i.[4] Since groups are processed in decreasing order, for each such index a prev pointer must have been computed already. As p belongs to a lower group than all of those indices, $p \leq \mathrm{prev}(j)$ must hold for all $p < j < i$. Consequently, p is reachable from the prev pointer chains starting at all indices j with $p < j < i$, but as index i had no prev pointer before the current step, $p_\mathsf{c} = i$ must hold. Now, after the computation of the prev pointer, p is reachable from all indices up to $i_\mathsf{c} - 1$, so the new context of p is $S[p..i_\mathsf{c})$. This shows that p's old context was extended by the context of the currently processed group. Consequently, p must be placed into a new group, as performed in Steps 2 and 3.

Another property of the processing is a consistent group order: For any groups $\mathcal{G}_1$ and $\mathcal{G}_2$, $\mathcal{G}_1$ is lower ordered than $\mathcal{G}_2$ if and only if the context of $\mathcal{G}_1$ is lexicographically smaller than the context of $\mathcal{G}_2$. Whenever a new group is created, its context is extended by a lexicographically larger context, so the new group must be placed higher than the old one. Also, since the context of the old group is lexicographically smaller than that of the next

[3] After the initial step $i_\mathsf{c} = i + 1$ holds for all indices, because no prev pointers were computed yet.
[4] The special case that groups of indices between p and i are equal to $\mathrm{group}(i)$ will be handled later.

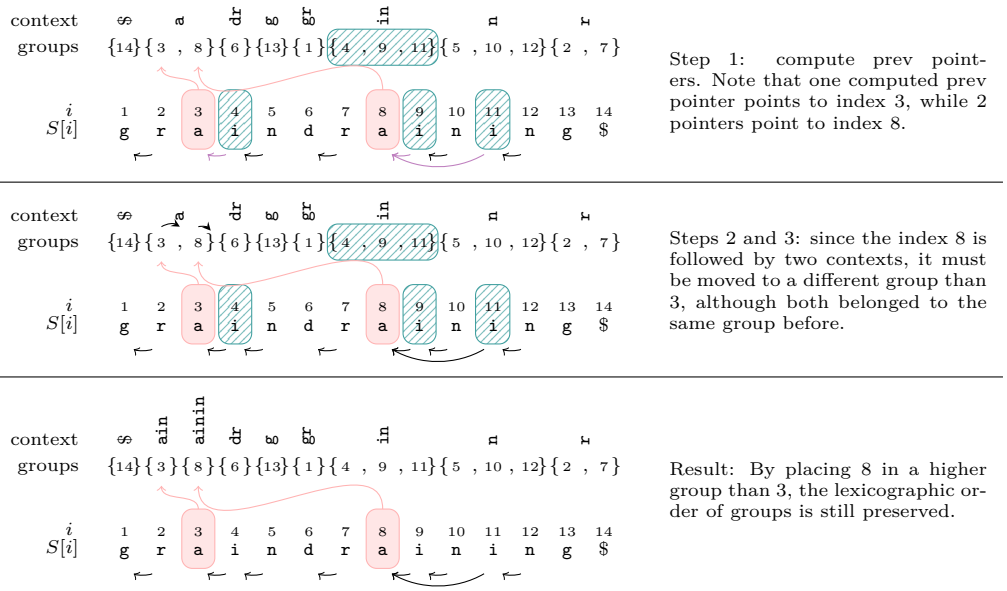


Figure 3 Third iteration step of Phase 1 applied to the string $S =$ `graindraining`$.

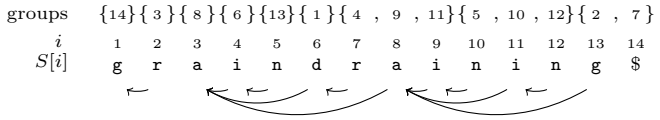


Figure 4 Groups and prev pointers from the string $S =$ `graindraining`$ after Phase 1.

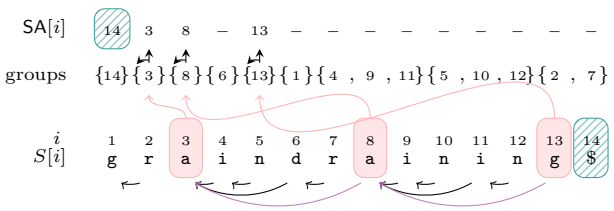
higher group $\widetilde{\mathcal{G}}$, the extended context of the new group is lexicographically smaller than that of $\widetilde{\mathcal{G}}$, so the placement of the new groups in Step 3 preserves the lexicographic order.

Now knowing that context extensions take place, one needs to be aware of one special case to preserve a consistent group order: Think about two indices i and j of the same group such that one prev pointer from an index of the currently processed group points to i, and two prev pointers from the currently processed group point to j. Since context extensions take place, i's context is extended one time, while j's context is extended by two contexts of the currently processed group. Since i and j belong to the same group, the new context of i is lexicographically smaller than that of j. As a consequence, after the extensions, i and j cannot belong to the same group, and must be handled separately as shown in Figure 3. Note that the example considers only two indices with different pointer counts; in general terms, an arbitrary number of indices and pointers must be taken into account.

The result of Phase 1 for our running example can be found in Figure 4. Summarising, the *greedy* group processing from highest to lowest group in conjunction with aspects of *implicit dynamic programming* lead to the desired group division after Phase 1. A formal proof for correctness must be omitted here, but can be found in [2]. Next, we'll take a look at the implementation of the missing part: Phase 2.

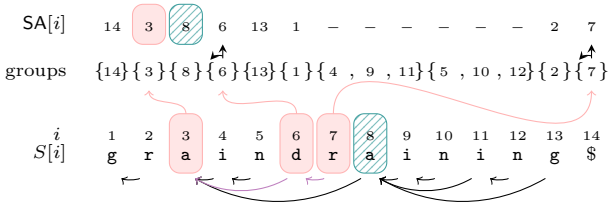
3.2 Example: Phase 2

After dividing suffixes into groups in Phase 1, the purpose of Phase 2 is to compute the suffix array using the group division. During Phase 2, the suffix array is processed in ascending



None of the elements in the prev pointer chain of index $\mathsf{SA}[1]-1 = 13$ is placed in the suffix array already, so $\widehat{j} = \mathsf{SA}[1]$ holds for each such index. Each index is removed from its current group and placed into a new group as immediate predecessor of its old group. Also, each index is placed into SA, at the position that equals the number of suffixes placed in lower groups.

Figure 5 First iteration step of Phase 2 applied to the string $S = \texttt{graindraining\$}$.



Index 3 is already contained in the suffix array, so only the suffixes S_6 and S_7 are placed into SA, since they are part of the prev pointer chain starting at index $\mathsf{SA}[3]-1 = 7$.

Figure 6 Third iteration step of Phase 2 applied to the string $S = \texttt{graindraining\$}$.

order. Within the i-th iteration, all indices j with $\widehat{j} = \mathsf{SA}[i]$ are computed. Each such index is removed from its current group, placed into a new group as immediate predecessor of its old group, and stored in the suffix array, see Algorithm 1.

The main issue in implementing this method is to compute indices j with $\widehat{j} = \mathsf{SA}[i]$. As we will see, prev pointers computed in Phase 1 will be very useful for this computation: starting at $j \coloneqq \mathsf{SA}[i]-1$, we follow the prev pointer chain $\mathrm{prev}(j), \mathrm{prev}(\mathrm{prev}(j)), \ldots$ until either no more prev pointer exists, or the index under consideration is already contained in the suffix array. The set $\{\, j \in [1\ldots n] \mid \widehat{j} = \mathsf{SA}[i] \,\}$ then consists of exactly those indices visited in the prev pointer chain of $\mathsf{SA}[i]-1$. Examples can be found in Figures 5 and 6, the next purpose is to ensure correctness of this statement.

The first index under consideration is $j \coloneqq \mathsf{SA}[i]-1$: if j is not contained in the suffix array already, then by the ascending iteration order of Phase 2, $S_{\mathsf{SA}[i]} <_{\mathsf{lex}} S_j$ must hold. Since S_j is the preceding suffix of $S_{\mathsf{SA}[i]}$, $S_{\mathsf{SA}[i]}$ clearly must be the next lexicographically smaller suffix of S_j. Now, given a suffix S_j with $\widehat{j} = \mathsf{SA}[i]$, the next index k with $\widehat{k} = \mathsf{SA}[i]$ (if existing) can be found by following j's prev pointer, i.e. $k = \mathrm{prev}(j)$. If k is not contained in the suffix array already, $S_{\mathsf{SA}[i]} <_{\mathsf{lex}} S_k$ must hold. Also, since $\mathrm{group}(k) < \mathrm{group}(l)$ holds for all $k < l \leq j$ by the definition of prev pointers, $S_k <_{\mathsf{lex}} S_l$ holds for all $k < l \leq j$ because of the group order of Phase 1. This indeed means that $\widehat{k} \geq \widehat{j}$. Combined with $S_{\mathsf{SA}[i]} <_{\mathsf{lex}} S_k$, $S_{\mathsf{SA}[i]}$ clearly must be the next lexicographically smaller suffix of S_k.

For any index k between j and $\mathrm{prev}(j)$ ($\mathrm{prev}(j) < k < j$) $\mathrm{group}(k) \geq \mathrm{group}(j)$ must hold by the definition of prev pointers. If $\mathrm{group}(k) > \mathrm{group}(j)$, by sorting in Phase 1, $S_k >_{\mathsf{lex}} S_j$ must hold. Because $k < j$, $\widehat{k} \leq j \neq \mathsf{SA}[i]$ holds, so those indices can be skipped. In the special case that $\mathrm{group}(k) = \mathrm{group}(j)$, by Phase 1, $S[k..\widehat{k}) = S[j..\widehat{j})$ holds. Since $k < j$ and the contexts are the same, $\widehat{k} < \widehat{j}$ holds, so clearly $\widehat{k} \neq \mathsf{SA}[i]$ must be fulfilled and those indices can be skipped, too.

If an index j is reached that is already contained in the suffix array, we know that it must have been placed into the suffix array in an earlier step. This indeed means that $S_{\widehat{j}} <_{\mathsf{lex}} S_{\mathsf{SA}[i]}$, so j can be skipped. For any further index k in the prev pointer chain of j, an argumentation as above clearly shows that $S_{\widehat{k}} <_{\mathsf{lex}} S_{\mathsf{SA}[i]}$, so those indices can be

Algorithm 2 Suffix array construction of a given nullterminated string S of length n.

Phase 1: divide suffixes into groups

order all suffixes of S into groups according to their first character:
Let S_i and S_j be two suffixes. Then, $\text{group}(i) = \text{group}(j) \Leftrightarrow S[i] = S[j]$.
order the suffix groups: Let $\mathcal{G}_1$ be a suffix group with group context character u, $\mathcal{G}_2$ be a suffix group with group context character v. Then, $\mathcal{G}_1 < \mathcal{G}_2$ if $u < v$.
for each group $\mathcal{G}$ in descending group order **do**
    for each $i \in \mathcal{G}$ **do**
        $\text{prev}(i) \leftarrow \max(\{\ j \in [1 \dots i] \mid \text{group}(j) < \text{group}(i)\ \} \cup \{0\})$
    let $\mathcal{P}$ be the set of previous suffixes from $\mathcal{G}$,
    $\mathcal{P} := \{\ j \in [1 \dots n] \mid \text{prev}(i) = j \text{ for any } i \in \mathcal{G}\ \}$.
    split $\mathcal{P}$ into k subsets $\mathcal{P}_1, \dots, \mathcal{P}_k$ such that a subset $\mathcal{P}_l$ contains
    suffixes whose number of prev pointers from $\mathcal{G}$ pointing to them
    is equal to l, i.e. $i \in \mathcal{P}_l \Leftrightarrow |\{\ j \in \mathcal{G} \mid \text{prev}(j) = i\ \}| = l$.
    for $l = k$ **down to** 1 **do**
        split $\mathcal{P}_l$ into m subsets $\mathcal{P}_{l_1}, \dots, \mathcal{P}_{l_m}$ such that suffixes
        of same group are gathered in the same subset.
        for $q = 1$ **up to** m **do**
            remove suffixes of $\mathcal{P}_{l_q}$ from their group and put them into a new
            group placed as immediate successor of their old group.

Phase 2: construct suffix array from groups

$\mathsf{SA}[1] \leftarrow n$
for $i = 1$ **up to** n **do**
    $j \leftarrow \mathsf{SA}[i] - 1$
    while $j \neq 0$ **do**
        let sr be the number of suffixes placed in lower groups,
        i.e. $sr := |\{\ s \in [1 \dots n] \mid \text{group}(s) < \text{group}(j)\ \}|$.
        if $\mathsf{SA}[sr + 1] \neq$ `nil` **then**
            break
        $\mathsf{SA}[sr + 1] \leftarrow j$
        remove j from its current group and put it in a new group
        placed as immediate predecessor of j's old group.
        $j \leftarrow \text{prev}(j)$

skipped, too. For the remaining indices between this prev pointer chain, we can also use the argumentation above and forget about these indices, too.

We refer to [2] for a formal proof, it must be omitted here for reasons of space. So far, we've seen a running example along with some argumentations for correctness. The missing part is an algorithm along with its runtime analysis, which will be addressed in the next section.

4 Algorithm

The new suffix array construction algorithm including all special cases discussed in the previous section can be found in Algorithm 2.

Now, to verify that the algorithm can be implemented in asymptotic linear time, some technical details about the algorithm will be discussed. First thing that has to be done is to explain a set of needed data structures. Six arrays of size n will be used:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$S[i]$	g	r	a	i	n	d	r	a	i	n	i	n	g	$
GSIZE[i]	1	2	0	1	2	0	3	0	0	3	0	0	2	0
SA[i]	14	3	8	6	1	13	4	9	11	5	10	12	2	7
GLINK[i]	5	13	2	7	10	4	13	2	7	10	7	10	5	1
ISA[i]	5	13	2	7	10	4	14	3	8	11	9	12	6	1

Figure 7 Initial data structure setup after line 2 of Phase 1, applied to the string $S =$ `graindraining`$. Prev pointers are not listed since all entries initially are set to `nil`.

- SA contains suffix starting positions, ordered according to the current group order.
- ISA is the inverse permutation of SA, to be able to detect the position of a suffix in SA.
- GSIZE contains the sizes of all groups. Group sizes are ordered according to the group order, so GSIZE has the same order as SA. GSIZE contains the size of each group once at the beginning of the group, followed by zeros until the beginning of the next group.
- GLINK stores pointers from suffixes to their groups. All entries point to the beginning of a group, at the same position where GSIZE contains the size of the group.
- PREV is used to store prev pointers. All entries initially are set to `nil`.
- PC is used to count prev pointers pointing from $\mathcal{G}$ to $\mathcal{P}$. PC initially is set to zero.

The initial setup of those structures (lines 1 and 2 of Algorithm 2) can be performed in $O(n)$ time using a technique called *bucket sort*. Refer to Figure 7 for an example.

The first problem to be solved is the processing of groups in descending group order, line 3. Therefore, if two variables gs and ge contain the bounds of the current group $\mathcal{G}$ in SA, we get to the preceding group by setting $ge \leftarrow gs - 1$ and $gs \leftarrow \mathsf{GLINK}[\mathsf{SA}[gs - 1]]$, and so trivially need $O(n)$ time to iterate over all groups.

For the prev pointer computation in line 5, we observe the following: Each index j between an index i and $\mathrm{prev}(i)$ belongs to a higher or equal group. If j belongs to a higher group, its prev pointer is already computed, and each index between j and $\mathrm{prev}(j)$ belongs to a higher group than that of i. So, to compute the prev pointer of an index i, we start at index $i-1$ and follow prev pointers until an index j belongs to the same or a lower group[5]. If j belongs to a lower group, the prev pointer of i is found; otherwise, if j belongs to the same group and itself has no prev pointer yet, we collect j in a list and repeat the same procedure, thus setting prev pointers of a whole list of indices. This technique is called *pointer jumping* and is well known to require $O(n)$ work totally, since each pointer is used only once for pointer jumping, and overall n pointers are computed. The extra amount of work for the list collection is $O(|\mathcal{G}|)$, and therefore sums up to $O(n)$ in total for Phase 1, since each group is processed only once.

For the computation of the set $\mathcal{P}$ and subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$[6], (lines 6 to 7) we use the PC-array. After prev pointer computation, for each $i \in \mathcal{G}$, we increment $\mathsf{PC}[\mathsf{PREV}[i]]$. After this loop, $\mathsf{PC}[p]$ contains the count of prev pointers pointing from $\mathcal{G}$ to p. Also note that the set $\mathcal{P}$ easily can be computed during the loop, by adding the index $\mathrm{prev}(i)$ to set $\mathcal{P}$ if $\mathsf{PC}[\mathrm{prev}(i)]$ was zero before the incrementation. Now, while the set $\mathcal{P}$ is not empty, do the following: In the l-th iteration, for each $p \in \mathcal{P}$, decrement $\mathsf{PC}[p]$. If $\mathsf{PC}[p]$ is zero, remove p from $\mathcal{P}$ and add it to set $\mathcal{P}_l$. This way, all sets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are computed, and all entries of the array PC are set to zero, so it can be reused again. Time results in $O(|\mathcal{G}|)$ per group $\mathcal{G}$, because the

[5] This can be done by comparing $\mathsf{GLINK}[j]$ with gs from the actual group.
[6] The set $\mathcal{P}$ and subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ can be implemented as list and list of lists respectively.

Table 2 SACA performance results. Speed[a)] and cache misses[b)] are composed of the arithmetic mean of 10 runs per file for each text corpus.

Text Corpus		divsufsort[12]	SA-IS[13]	KA[9]	DC3[20]	GSACA[1]
Silesia [3]	speed[a)]	15.9 MB/s	17.2 MB/s	8.1 MB/s	2.9 MB/s	4.5 MB/s
(files < 40 MB)	cache misses[b)]	26.5 %	32.7 %	24.2 %	52.0 %	61.2 %
Pizza & Chili [4]	speed[a)]	9.2 MB/s	8.1 MB/s	3.5 MB/s	1.1 MB/s	3.0 MB/s
(files with 200 MB)	cache misses[b)]	49.5 %	74.8 %	55.2 %	86.1 %	79.0 %
Repetitive [5]	speed[a)]	12.5 MB/s	14.2 MB/s	5.3 MB/s	1.7 MB/s	3.5 MB/s
(files > 45 MB)	cache misses[b)]	41.9 %	68.6 %	49.7 %	78.0 %	76.9 %

a) Construction speed: $^{\text{size of input}}/_{\text{time to construct SA}}$, in $^{\text{MB}}/_{\text{s}}$.
b) Cache miss rate: $^{\text{number of cache misses}}/_{\text{number of cache accesses}}$ of last–level cache, in %.

number of decrementations in the array PC is identical to the number of additions in the preceding stage, and therefore again, computation requires $O(n)$ work during Phase 1.

The suffix rearrangements from lines 9 to 11 can be performed like the following:

1. For all $p \in \mathcal{P}_l$, decrement $\mathsf{GSIZE}[\mathsf{GLINK}[p]]$ and exchange p with the index placed at $\mathsf{GLINK}[p] + \mathsf{GSIZE}[\mathsf{GLINK}[p]]$ using SA and ISA. This way, p is moved to the back of its group and 'virtually' removed from it.[7]
2. For all $p \in \mathcal{P}_l$, set $\mathsf{GLINK}[p]$ to $\mathsf{GLINK}[p] + \mathsf{GSIZE}[\mathsf{GLINK}[p]]$, so GLINK correctly points to the beginning of the new groups again.
3. For all $p \in \mathcal{P}_l$, increment $\mathsf{GSIZE}[\mathsf{GLINK}[p]]$, so the sizes of the new groups are correct.

Total work again results in $O(n)$ for Phase 1, for the same reasons as above.

After the processing of a group $\mathcal{G}$ is finished, we also set $\mathsf{SA}[ge] \leftarrow gs$ and $\mathsf{ISA}[i] = ge$ for all indices $i \in \mathcal{G}$: this serves as a preparation for Phase 2. In Phase 2, to detect if an index j is contained in SA already (line 17), we check if $\mathsf{ISA}[j] = 0$ holds; otherwise, sr, the number of suffixes placed in lower groups (line 16), can be computed using ISA and SA. As mentioned above, in Phase 2, ISA entries point to the end of a group. The last index of a group then contains a pointer to the start of the group. If we set $sr \leftarrow \mathsf{SA}[\mathsf{ISA}[j]]$, increment $\mathsf{SA}[\mathsf{ISA}[j]]$ and afterwards set $\mathsf{SA}[sr] \leftarrow j$ and $\mathsf{ISA}[j] \leftarrow 0$, j 'virtually' gets removed from its group, while the group counter points to the next SA - entry.

Now, summing up all work performed, we get $O(n)$ work for Phase 1 as well as for Phase 2, since the inner loop of Phase 2 is executed $n - 1$ times totally, as each suffix has exactly one next lexicographically smaller suffix. There might be smarter ways to implement the algorithm; refer to [2] for other suggestions; however, the point of interest here is that Algorithm 2 can be implemented in a non-recursive way, running in asymptotic linear time.

5 Performance Analyses

All experiments were conducted on a 64 bit Ubuntu 14.04.3 LTS system equipped with two ten-core Intel Xeon processors E5-2680v2 with 2.8 GHz and 128 GB of RAM.

The algorithm described in this paper was named *GSACA* because of its greedy and grouping behaviour. It was compared against common linear-time and state of the art SACAs on text selections of different text corpuses. The benchmark itself is available online [1], results can be found in Table 2.

[7] Note that the additional split of $\mathcal{P}_l$ from line 9 of Algorithm 2 implicitly is performed within this step.

The results clearly show that GSACA cannot keep up with current state of the art SACAs; construction speeds of divsufsort or SA-IS are about 3 to 4 times faster than those of GSACA. Limited performance mainly is owed to cache-unfriendly operations like pointer jumping or suffix rearrangements, causing high cache miss rates and slow construction.

6 Conclusion

We presented the first non-recursive linear-time suffix array construction algorithm. Unfortunately, by comparing its performance with other linear–time SACAs, GSACA must be seen as a late child of the 2003 'epoch of suffix array construction' rather than a state of the art SACA. Nonetheless, the results are quite promising: the algorithm deals a lot with previous smaller and next smaller values, what normally hints to an alternative stack-based approach. This could result in better cache miss rates and speed, but this remains an open problem for the moment. Compared to developmental histories of other SACAs, GSACA is in its infancy, and therefore offers a lot of room for future improvements.

References

1 Uwe Baier. GSACA. `https://github.com/waYne1337/gsaca`. last visited January 2016.
2 Uwe Baier. Linear-time Suffix Sorting-A new approach for suffix array construction. Master's thesis, Ulm University, 2015.
3 Sebastian Deorowicz. Silesia Corpus. `http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia`. last visited January 2016.
4 Paolo Ferragina and Gonzalo Navarro. Pizza & Chili Corpus. `http://pizzachili.dcc.uchile.cl/texts.html`. last visited January 2016.
5 Paolo Ferragina and Gonzalo Navarro. Repetitive Corpus. `http://pizzachili.dcc.uchile.cl/repcorpus.html`. last visited January 2016.
6 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, pages 251–260, 2003.
7 Juha Kärkkäinen and Peter Sanders. Simple Linear Work Suffix Array Construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP '03, pages 943–955, 2003.
8 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time Construction of Suffix Arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM '03, pages 186–199, 2003.
9 Pang Ko. Ko–Aluru Algorithm. `https://sites.google.com/site/yuta256/KA.tar.bz2`. last visited January 2016.
10 Pang Ko and Srinivas Aluru. Space Efficient Linear Time Construction of Suffix Arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM '03, pages 200–210, 2003.
11 Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-line String Searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, 1990.
12 Yuta Mori. libdivsufsort. `https://github.com/y-256/libdivsufsort`. last visited January 2016.
13 Yuta Mori. sais–lite–2.4.1. `https://sites.google.com/site/yuta256/sais`. last visited January 2016.
14 Yuta Mori. Suffix Array Construction Benchmark. `https://github.com/y-256/libdivsufsort/blob/wiki/SACA_Benchmarks.md`. last visited January 2016.

15 Joong Chae Na. Linear-Time Construction of Compressed Suffix Arrays Using O(N Log N)-bit Working Space for Large Alphabets. In *Proceedings of the 16th Annual Conference on Combinatorial Pattern Matching*, CPM '05, pages 57–67, 2005.

16 Ge Nong. Practical Linear-time O(1)-workspace Suffix Sorting for Constant Alphabets. *ACM Transactions on Information Systems*, 31(3):15:1–15:15, 2013.

17 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In *Proceedings of the 2009 Data Compression Conference*, DCC '09, pages 193–202, 2009.

18 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Time Suffix Array Construction Using D-Critical Substrings. In *Proceedings of the 20th Annual Conference on Combinatorial Pattern Matching*, CPM '09, pages 54–67, 2009.

19 Simon J Puglisi, William F Smyth, and Andrew H Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computational Survey*, 39(2), 2007.

20 Peter Sanders. DC3 Algorithm. `http://people.mpi-inf.mpg.de/~sanders/programs/suffix/`. last visited January 2016.

21 Peter Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, SWAT '73, pages 1–11, 1973.

Color-Distance Oracles and Snippets

Tsvi Kopelowitz[*1] **and Robert Krauthgamer**[†2]

1 **University of Michigan, Ann Arbor, Michigan, USA**
kopelot@gmail.com
2 **Weizmann Institute of Science, Rehovot, Israel**
robert.krauthgamer@weizmann.ac.il

Abstract

In the snippets problem we are interested in preprocessing a text T so that given two pattern queries P_1 and P_2, one can quickly locate the occurrences of the patterns in T that are the closest to each other. A closely related problem is that of constructing a color-distance oracle, where the goal is to preprocess a set of points from some metric space, in which every point is associated with a set of colors, so that given two colors one can quickly locate two points associated with those colors, that are as close as possible to each other.

We introduce efficient data structures for both color-distance oracles and the snippets problem. Moreover, we prove conditional lower bounds for these problems from both the 3SUM conjecture and the Combinatorial Boolean Matrix Multiplication conjecture.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Snippets, Text Indexing, Distance Oracles, Near Neighbor Search

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.24

1 Introduction

We introduce and study the following problem: preprocess a text T so that given two pattern queries, P_1 and P_2, one can quickly locate the occurrences of the two patterns in T that are closest to each other, or report the distance between these occurrences. This natural task arises in many common indexing applications, for example, when searching a corpus of documents for two query keywords, the relevance of a document may be measured by the two keywords' proximity inside the document. Web search engines often use this notion of relevance by providing with each result a *snippet* — a subtext from the corresponding webpage in which the two keywords appear close to each other, which is very useful to assess the relevance of that result. This problem, which we call the *snippets* problem, turns out to be a special case of a more general problem, which we define next, and deals with colored points in a metric space.

Colored Points.

Let M be a metric space with distance function $d(\cdot,\cdot)$. Each point $p \in M$ may have an associated color $c_p \in [\ell]$, in which case we say that the point is *colored*. Let $S \subset M$ be a set of N points. We call S a *colored set* if every point $p \in S$ is colored. For a color $c \in [\ell]$, let $P(c)$ denote the set of points in S which have color c. The distance between a point $p \in M$ and a color $c \in [l]$ is defined as $d(p,c) := \min\{d(p,q) : \ q \in P(c)\}$. The distance between

* Work supported in part by NSF grants CCF-1217338, CNS-1318294, and CCF-1514383.
† Work supported in part by an Israel Science Foundation grant #897/13.

two colors, called the *color-distance*, of $c, c' \in [\ell]$ is defined as $d(c, c') := \min\{d(q, q') : q \in P(c), q' \in P(c')\}$.

One application for computing the distance of two colors arises in navigational tools. For example, consider a user who is interested in visiting both a postoffice and a pharmacy. One can color, in advance, all of the pharmacies with one color and all of the postoffices with another color. The distance between the two colors corresponds to the closest pair of a postoffice and a pharmacy. This leads to the following problem.

▶ Problem 1.1 (Color-Distance Oracle). The color-distance oracle problem asks to preprocess a colored set S, so that given a query of two colors $c, c' \in [\ell]$, one can quickly report $d(c, c')$.

Multi-colored points.

A natural generalization of color-distance oracles is to let each point have several colors. For example, a single location in a map could be both a postoffice and a pharmacy. A point $p \in M$ is said to be *multi-colored* if p has an associated nonempty set of colors $C(p) \subseteq [\ell]$. A set of points $S \subset M$ is a *multi-colored set* if each point $p \in S$ is multi-colored.

▶ Problem 1.2 (Multi-Color-Distance Oracle). The multi-color-distance oracle problem asks to preprocess a multi-colored set S, so that given a query of two colors $c, c' \in [\ell]$, one can quickly report $d(c, c')$.

One straightforward way for solving the multi-color-distance oracle problem is to create $|C(p)|$ copies of each point, one for each color, and apply a solution for the color-distance oracle problem (such as Theorem 2). The size of the instance of the newly created instance is $\sum_{p \in S} |C(p)|$, which could be much larger than $N = |S|$. Notice that this quantity is actually the size of the input for the multi-color-distance oracle problem, since generally speaking, each point may need to have its colors listed explicitly. Nevertheless, there are interesting cases in which the list of colors for each point need not be given explicitly. One such example is in the snippets problem, which falls under the notion of a color hierarchy, described next.

Color hierarchies.

We say that a multi-colored set S admits a *color hierarchy* if for every two colors $c, c' \in [l]$, either one of the sets $P(c)$ and $P(c')$ contains the other, or the two sets are disjoint (a formal terminology is that $\{P(c)\}_{c \in [l]}$ is a laminar family). It is easy to see that a color hierarchy can be represented by a rooted forest (i.e., each tree has a root) T_S of size $O(\ell)$, where each color c is associated with a vertex u_c in this forest, such that the descendants of u_c are exactly all the vertices $u_{c'}$ whose color c' satisfies $P(c') \subseteq P(c)$. We convert the forest T_S to a rooted tree by adding a dummy root vertex and making it the parent of all of the roots of the trees in the forest. With the aid of T_S, a multi-colored set S that admits a color hierarchy can be represented using only $O(N + \ell)$ machine words, because it suffices to describe the tree and associate with each point just one color (the color with the lowest corresponding vertex in T_S); the other colors of this point are implicit from T_S (the colors on the path to the root of T_S). This leads us to the following problem.

▶ Problem 1.3 (Multi-Color-distance Oracle with a Color Hierarchy). The multi-color-distance oracles with a color hierarchy problem asks to preprocess a multi-colored set S that admits a color hierarchy, so that given a query of two colors $c, c' \in [\ell]$, one can quickly report $d(c, c')$.

1.1 Our Results

Our main result is a data structure for the snippets problem, summarized as follows.

▶ **Theorem 1.** *For every fixed $\epsilon > 0$, there is a data structure for the snippets problem with preprocessing time $O(N^{1.5}\log^{\epsilon} N)$, query time $O(|P_1|+|P_2|+\sqrt{N}\log^{\epsilon} N))$, and space usage $O(N)$ words.*

To prove Theorem 1 we solve the more general problems of colored and multi-color-distance oracles. These data structures use (in a black-box manner) an algorithm (data structure) for nearest neighbor search.

Nearest Neighbor Search (NNS).

In the Nearest Neighbor Search (NNS) problem, the goal is to preprocess a set of points $S \subset M$ (recall M is a metric space), so that given a point $p \in M$ one can quickly report $\text{argmin}_{q\in S}\{d(p,q)\}$. Given an NNS algorithm for $k = |S|$ points, we denote its preprocessing time by $t_{\text{NNS}}(k)$, its query time by $q_{\text{NNS}}(k)$, and its space usage by $s_{\text{NNS}}(k)$. Once the nearest neighbor is found, one can evaluate the distance between p and its nearest neighbor by invoking the function d (we assume such evaluation takes $O(1)$ time, for simplicity), thereby obtaining $d(p,S)$.

With the aid of an NNS data structure for point sets in a metric M, we prove the following theorem in Section 2.

▶ **Theorem 2** (Color-distance Oracle)**.** *Assume there is a data structure that supports NNS queries on a set of k points from M using preprocessing time $t_{\text{NNS}}(k)$, query time $q_{\text{NNS}}(k)$, and space usage $s_{\text{NNS}}(k)$ words. Then for every $0 < \tau \le N$, there exists a color-distance oracle for N-point sets in M, that has preprocessing time $O(t_{\text{NNS}}(N) + N\cdot\tau\cdot q_{\text{NNS}}(N))$, query time $O(\tau\cdot q_{\text{NNS}}(N))$, and space usage $O(s_{\text{NNS}}(N) + (\frac{N}{\tau})^2)$ words.*

Our solution for the multi-color-distance oracles with a color hierarchy problem is based on the notion of range NNS, which is defined as follows.

▶ Problem 1.4 (Range NNS). In the range NNS problem the goal is to preprocess an array A of N points from a metric M, so that given two indices $1 \le i \le j \le N$ and a point $p \in M$, one can quickly find the NNS of p in the set $\{A[i], A[i+1], \ldots, A[j]\}$.

We prove the following theorem in Section 3.

▶ **Theorem 3** (Multi-Color-distance Oracle with a Color Hierarchy)**.** *Assume there is a range NNS algorithm for k-point sets in a metric M that uses preprocessing time $t_{\text{RNNS}}(k)$, query time $q_{\text{RNNS}}(k)$, and space usage $s_{\text{RNNS}}(k)$ words. Then for every $0 < \tau \le N$, there exists a multi-color-distance oracle for N-point sets in M that admit a color hierarchy, (specifically, the multi-coloring of the input S is given implicitly via a tree T_S), that has preprocessing time $O(t_{\text{RNNS}}(N) + \frac{N^2}{\tau}\cdot(q_{\text{RNNS}}(N) + \log\log\log\frac{N}{\tau}))$, query time $O(\tau\cdot q_{\text{RNNS}}(N))$, and space usage $O(s_{\text{RNNS}}(N) + (\frac{N}{\tau})^2)$ words.*

Conditional Lower Bounds.

Solving the multi-color-distance oracle problem with poly-logarithmic query time and non-trivial preprocessing time seems to be extremely difficult, leading to the question of finding a polynomial time lower bound. Polynomial (unconditional) lower bounds for data structure problems are considered beyond the reach of current techniques. Thus, it is common to

prove conditional lower bounds (CLBs) based on the *conjectured* hardness of some "basic" problem. One of the most popular conjectures for CLBs is that the 3SUM problem (given n integers determine if any three sum to zero) cannot be solved in truly subquadratic time, where truly subquadratic time is $O(n^{2-\Omega(1)})$ time. This conjecture is reasonable even if the algorithm is allowed to use randomization, see e.g. [30, 1, 23, 15].

Another popular conjecture is the *combinatorial Boolean matrix multiplication* (BMM) conjecture. In the BMM problem we are given two $n \times n$ Boolean matrices A and B and the task is to compute the Boolean product of the two matrices. The combinatorial BMM conjecture states that combinatorial algorithms for computing this Boolean product require runtime $\Omega(n^{3-o(1)})$, see [1].

In Section 5 we prove CLBs for the color-distance oracle problem based on the 3SUM and combinatorial BMM conjectures. Moreover, these CLBs hold also for approximate versions of the color-distance oracle problem, where the answer to a color-distance query between two colors c and c' is required to be between $d(c, c')$ and $\alpha \cdot d(c, c')$, where $\alpha \geq 1$ is a *stretch* parameter. The CLBs are summarized as follows.

▶ **Theorem 4.** *Assume the 3SUM conjecture holds. Then for every fixed $0 < \gamma < 1$ and fixed $\alpha \geq 1$, every data structure for the color-distance oracle problem with stretch α for points on the line, that has preprocessing time t_{CDO} and query time q_{CDO}, must satisfy*

$$t_{CDO} + N^{\frac{1+\gamma}{2-\gamma}} q_{CDO} = \Omega\left(N^{\frac{2}{2-\gamma}-o(1)}\right).$$

Notice that by taking γ arbitrarily close to 0, a linear preprocessing time implies an $\Omega(N^{0.5-o(1)})$ query time. This is line with other conditional lower bounds based on the 3SUM conjecture [23].

▶ **Theorem 5.** *Assume the combinatorial BMM conjecture holds. Then every combinatorial data structure for the color-distance oracle problem with constant stretch $\alpha \geq 1$ for points on the line, that has preprocessing time t_{CDO} and query time q_{CDO}, must satisfy*

$$t_{CDO} + N \cdot q_{CDO} = \Omega\left(N^{1.5-o(1)}\right).$$

Comparing Theorem 5 with Theorem 1 and assuming the combinatorial BMM conjecture holds, it is impossible to obtain a polynomial speedup in both the preprocessing and query time of Theorem 1 via combinatorial algorithms. However, it might be possible to obtain a polynomial speedup in one of them. We emphasize that the proofs of Theorems 4 and 5 are for the one dimensional case, and thus the conditional lower bounds apply to the special case of the snippets problem.

1.2 Related Work

Perhaps the most related problem color-distance oracles is the *(approximate) vertex-labeled distance oracles for graphs problem*, where we are interested in preprocessing a colored graph G so that given a query of a vertex q and a color c we can return $d(q, c)$ (or some approximation thereof). Hermelin, Levy, Weimann and Yuster [16] introduced this problem and provided, amongst other results, a data structure using $O(kn^{1+1/k})$ expected space with stretch $(4k-5)$ and $O(k)$ query time. In another result they showed how to reduce the space usage to $O(kN\ell^{1/k})$ at the expense of an exponential stretch $(2^k - 21)$. Chechik [8] showed how to reduce this stretch back down to $(4k - 5)$.

Two pattern document retrieval problems.

Another related body of work are document retrieval problems on two patterns. In the *Document Retrieval problem* [28] we are interested in preprocessing a collection of documents $X = \{D_1, \cdots, D_k\}$ where $N = \sum_{D \in X} |D|$, so that given a pattern P we can quickly report all of the documents that contain P. Typically, we are interested in run time that depends on the number of documents that contain P and not in the total number of occurrences of P in the entire collection of documents. In the *Two Patterns Document Retrieval problem* we are given two patterns P_1 and P_2 during query time, and wish to report all of the documents that contain both P_1 and P_2. In the *Forbidden Pattern Document Retrieval problem* [14] we are also interested in preprocessing the collection of documents but this time given a query P^+ and P^- we are interested in reporting all of the documents that contain P^+ and do not contain P^-.

All known solutions for the Two Patterns Document Retrieval problem or the Forbidden Pattern Document Retrieval problem with non trivial preprocessing use at least $\Omega(\sqrt{N})$ time per query [28, 11, 17, 18, 14]. In a recent paper, Larsen, Munro, Nielsen, and Thankachan [25] show lower bounds for these problems conditioned on the hardness of BMM. More recently some CLBs for both problems were shown from the 3SUM conjecture as well [23].

Nearest Neighbor Search.

The NNS problem has been studied intensively for many metric spaces M, due to its numerous applications. The literature on both theoretical and practical aspects is very extensive, and we provide below only a brief overview of leading theoretical approaches.

In the classical setting where M is a D-dimensional Euclidean space, the standard algorithm is to preprocess the point set by computing a Voronoi diagram, which has a fast query time. However, the Voronoi diagram requires $O(n^{\lceil D/2 \rceil})$ time and space, which is prohibitive unless D is rather small. Several algorithms are known to achieve $(1+\epsilon)$-approximate NNS in R^D (often under any ℓ_p norm) by employing various space partitions. Specifically, Arya, Mount, Netanyahu, Silverman, and Wu [5] achieve preprocessing time that is linear in the number of points k (but exponential in D), which is quite effective when the dimension D is not too large. Locality Sensitive Hashing (LSH), which was introduced by Indyk and Motwani [19] and further refined later, see e.g. [3, 4], is an alternative approach that is often preferred for high dimension D, because its performance is polynomial in D (although its query time is typically polynomial, and not logarithmic, in k).

In general metric spaces (i.e., not of the form R^D), NNS is considered a very difficult problem. But under certain "bounded growth" conditions on the data, one can obtain performance that is similar to, or even better than, the low-dimensional Euclidean case, see e.g. [10, 20, 24, 12], and the survey [9].

2 Color-distance Oracle

Proof of Theorem 2. We begin by preprocessing each set $P(c)$ with a NNS data structure. This takes a total of $O(s_{\text{NNS}}(N))$ words of space and $O(t_{\text{NNS}}(N))$ time. A color c is said to be *light* if $|P(c)| < \tau$. If color c is not light then it is *heavy*. Notice that there can be at most N/τ heavy colors. For each pair of heavy colors we precompute and store their distances in a lookup table using $O((\frac{N}{\tau})^2)$ words. The computation of the entries for this table is done directly using $O(N\tau)$ NNS queries. To answer a color-distance query on two colors c and c', if both colors are heavy then we use the lookup table for their precomputed distance.

Otherwise, assume without loss of generality that c is light. We then execute $|P(c)| < \tau$ NNS queries on $P(c')$, one for each of the points in $P(c)$, and the distance between c and c' is the smallest distance found by any of these NNS queries.

To summarize, the preprocessing time is $O(t_{\text{NNS}}(N) + N \cdot \tau \cdot q_{\text{NNS}}(N))$, the query time is $O(\tau \cdot q_{\text{NNS}}(N))$, and the space usage is $O(s_{\text{NNS}}(N) + (\frac{N}{\tau})^2)$ words. ◀

Since we may assume that $s_{\text{NNS}}(k) = \Omega(k)$ (as we need to store all of the points in S), picking $\tau = \sqrt{N}$ the preprocessing time becomes $O(t_{\text{NNS}}(N) + N^{1.5} q_{\text{NNS}}(N))$, the space usage becomes $O(s_{\text{NNS}}(N))$ and the query time is $O(\sqrt{N} \cdot q_{\text{NNS}}(N))$.

3 Multi-color-distance Oracle

Proof of Theorem 3. Assume without loss of generality that T_S is an ordinal tree (the children of each vertex are ordered). For every point $p \in S$ we create a new vertex u_p and add it to T_S as a child of the single vertex representing the color set $C(p)$. This process adds N leaves to T_S and now each leaf is associated with a unique point. With the aid of T_S we embed the set S in an array A, where the order is determined by the order in which the leaves (corresponding to points in S) are encountered during a pre-order traversal of T_S. After the construction of A, by the properties of ordered traversals on trees, each color $c \in [\ell]$ is associated with a range in A. We preprocess A using a RNNS data structure.

Next, we partition A into blocks of size τ. For each pair of blocks we precompute and store the two closest points, one from each block, together with their distances in a $\frac{N}{\tau} \times \frac{N}{\tau}$ matrix B. The entry $B[i][j]$ corresponds to the two closest points between the ith and jth blocks. It is straightforward to compute each entry in $O(\tau \cdot q_{\text{RNNS}}(N))$ time, for a total of $O(\frac{N^2}{\tau} \cdot q_{\text{RNNS}}(N))$ time to precompute the B. Next, we preprocess B using a 2D Range Minimum Query (2DRMQ) data structure [2]. Such data structures preprocess a matrix of values (in our case these are the distances that are stored in B) so that given a rectangle in the matrix, defined by its corners, we can quickly return the entry with the smallest value. The 2DRMQ data structure uses $O((\frac{N}{\tau})^2)$ space and $O((\frac{N}{\tau})^2 \log\log\log \frac{N}{\tau})$ preprocessing times, and the query time is constant.

Answering a query.

To answer a multi-color-distance oracle query between two colors c and c', let $[x_c, y_c]$ and $[x_{c'}, y_{c'}]$ be the ranges in A that are associated with c and c' respectively. Each interval can be partitioned into three parts, based on the block partitioning. The first (last) part is a suffix (prefix) of some block that starts from the left (ends at the right) endpoint of the interval. The middle part is everything else, which completely spans some consecutive blocks. Notice that the first and last part have size at most τ. For the two middle parts (one for each color) we find the two closest points by invoking the 2DRMQ data structure on B, since the two contiguous ranges define a natural rectangle in B. This covers all of the possible combinations of two points from the two middle parts and takes constant time. Now to the remaining parts. Each of the $O(\tau)$ points in the first and last parts of $[x_c, y_c]$ is queried against the entire range $[x_{c'}, y_{c'}]$ with the range NNS data structure. Similarly, each of the $O(\tau)$ points in the first and last parts of $[x_{c'}, y_{c'}]$ is queried against the entire range $[x_c, y_c]$ with the range NNS data structure. This costs $O(\tau \cdot q_{\text{RNNS}}(N))$ time. Finally, we take the minimum over all of the answers to the queries. Since the queries together cover all of the possible pairs of points, the minimum over all queries is the distance between the colors. ◀

4 Back to Snippets

A multi-color-distance oracle with a color hierarchy can be used to solve the snippets problem as follows. During the preprocessing phase we construct the suffix tree for the text T. Every internal node in the suffix tree defines a unique color. Location i in T is colored with all of the colors defined by nodes on the path from the root of the suffix tree to the leaf corresponding to the suffix at location i. Thus, the suffix tree represents the color hierarchy of the set of colors.

Given two query patterns, P_1 and P_2, we first locate their corresponding vertices in the suffix tree. This takes $O(|P_1| + |P_2|)$ time. These two nodes define the two colors that we give as input to the multi-color-distance oracle query. It is straightforward to see that the answer to this color-distance oracle query is exactly the distance of the two patterns in T.

In order to use Theorem 3 we need to specify a range NNS data structure that works in a metric defined by locations of an array. For this we can use the range predecessor data structures [13, 29, 26, 31, 27, 6, 7, 22, 21]. In these data structures the goal is to preprocess an array A of n elements from integer universe $[u]$ so that given a query $range\_pred(x, y, p)$ where $1 \leq x \leq y \leq n$ and $p \in [u]$ the data structure quickly returns $\text{argmax}_{q \in \{A[x], A[x+1], \ldots, A[y]\}}\{q < p\}$. Using, for example, the data structure of [29] the preprocessing time is $O(n \log n)$, the query time is $O(\log^{\epsilon} n)$ and the space usage is $O(n)$ words, where $\epsilon > 0$ is an arbitrarily small constant. Plugging these runtimes into Theorem 3 and setting $\tau = \sqrt{N}$ completes the proof of Theorem 1.

Notice that if the multi-color-distance oracle would return the two points that are closest, and not just their distance, then we could also report the two occurrences of the patterns that are closest to each other. Our implementations of multi-color-distance oracle do in fact allow for this information to be returned.

5 Conditional Lower Bounds

Offline SetDisjointness.

Both the 3SUM problem and the combinatorial BMM problem can be reduced to the SetDisjointness data structure problem. In this problem we wish to preprocess a family of sets F, all from universe U, with total size $N = \sum_{S \in F} |S|$ so that given a query of pointers to two sets $S, S' \in F$, one can quickly determine if $S \cap S' = \emptyset$. For a SetDisjointness data structure let t_p denote the preprocessing time and let t_q denote query time. The following theorems summarize the best known CLBs for the SetDisjointness data structure problem from the 3SUM and combinatorial BMM conjectures.

▶ **Theorem 6** ([23]). *Assume the 3SUM conjecture holds. For every fixed $0 < \gamma < 1$, any data structure for SetDisjointness has $t_p + N^{\frac{1+\gamma}{2-\gamma}} t_q = \Omega\left(N^{\frac{2}{2-\gamma} - o(1)}\right)$.*

▶ **Theorem 7** (Folklore). *Assume the combinatorial BMM conjecture holds. Any combinatorial data structure for SetDisjointness has $t_p + N \cdot t_q = \Omega\left(N^{1.5-o(1)}\right)$.*

SetDisjointness via color-distance oracles

We prove next that the SetDisjointness data structure problem can be reduced to the color-distance oracle problem. Combining this reduction with Theorems 6 and Theorem 7 we obtain CLBs for the the color-distance oracle problem from both the 3SUM conjecture and the combinatorial BMM conjecture. Moreover, this reduction also holds for approximate

versions of the color-distance oracle problem. In these versions the answer to a color-distance query between two colors c and c' can be as large as $\alpha \cdot d(c, c')$, where $\alpha \geq 1$ is a constant stretch parameter.

▶ **Theorem 8.** *If there exists a color-distance oracle with constant stretch $\alpha \geq 1$ for points on the line with t_{CDO} preprocessing time and q_{CDO} query time, then there exists a data structure for online* SetDisjointness *where $t_p = O(t_{CDO} \log k)$ and $t_q = O(q_{CDO} \log k)$.*

Proof. We reduce the SetDisjointness problem to the color-distance problem as follows. Let $F = \{S_1, \cdots, S_k\}$. For each S_i we define a unique color c_i. For an element $e \in U$ let $|e|$ denote the number of subsets containing e. Since each element in U appears in at most $O(k)$ subsets, we partition U into $\Theta(\log k)$ parts where the i^{th} part P_i contains all of the elements $e \in U$ such that $2^{i-1} < |e| \leq 2^i$. An array X_i is constructed from $P_i = \{e_1, \cdots e_{|P_i|}\}$ by assigning an interval $I_j = [f_j, \ell_j]$ in X_i to each $e_j \in P_i$ such that no two intervals overlap. Every interval I_j contains a list of all of the colors of sets in F that contain e_j. This implies that $|I_j| = |e_j| \leq 2^i$. Furthermore, for each e_j and e_{j+1} we separate I_j from I_{j+1} with a dummy color d listed $2^i + 1$ times at locations $[\ell_j + 1, f_{j+1} - 1]$. Finally, we pad each X_i so that its size is exactly N. This is always possible since $\sum_{e \in U} |e| = N$ (so the array is never of size more than N).

We now simulate a SetDisjointness query on subsets $(S_i, S_j) \in F$ by performing a color-distance query on colors c_i and c_j in each of the $\Theta(\log k)$ arrays. There exists a P_i for which the two points returned from the query are at distance strictly less than $2^i + 1$ if and only if there is an element in U that is contained in both S_i and S_j. Thus, using $O(\log k)$ color-distance queries we solve the SetDisjointness query.

Finally, notice that the reduction also holds for the approximate case, as for any constant α the reduction can overcome the α approximation by separating intervals using $2^i\alpha + 1$ instances of the dummy color d. ◀

Acknowledgments. We thank Sharma Thankachan for suggesting to consider range predecessor data structures, thereby significantly simplifying our earlier, more complicated, solution.

References

1 A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 434–443, 2014. doi:10.1109/FOCS.2014.53.

2 A. Amir, J. Fischer, and M. Lewenstein. Two-dimensional range minimum queries. In *Combinatorial Pattern Matching, 18th Annual Symposium, CPM*, pages 286–294, 2007. doi:10.1007/978-3-540-73437-6_29.

3 A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *47th Annual IEEE Symposium on Foundations of Computer Science*, pages 459–468. IEEE, 2006. doi:10.1109/FOCS.2006.49.

4 A. Andoni and I. Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *47th Annual ACM on Symposium on Theory of Computing*, STOC'15, pages 793–801. ACM, 2015. doi:10.1145/2746539.2746553.

5 S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998. doi:10.1145/293347.293348.

6 M. A. Babenko, P. Gawrychowski, T. Kociumaka, and T. A. Starikovskaya. Wavelet trees meet suffix trees. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 572–591, 2015. doi:10.1137/1.9781611973730.39.

7 D. Belazzougui and S. J. Puglisi. Range predecessor and lempel-ziv parsing. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2053–2071, 2016. doi:10.1137/1.9781611974331.ch143.

8 S. Chechik. Improved distance oracles and spanners for vertex-labeled graphs. In *ESA*, pages 325–336, 2012. doi:10.1007/978-3-642-33090-2_29.

9 K. Clarkson. Nearest-neighbor searching and metric space dimensions. In G. Shakhnarovich, T. Darrell, and P. Indyk, editors, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59. MIT Press, 2006. URL: http://kenclarkson.org/nn_survey/p.pdf.

10 K. L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Comput. Geom.*, 22(1):63–93, 1999.

11 H. Cohen and E. Porat. Fast set intersection and two-patterns matching. *Theor. Comput. Sci.*, 411(40-42):3795–3800, 2010.

12 R. Cole and L.-A. Gottlieb. Searching dynamic point sets in spaces with bounded doubling dimension. In *38th annual ACM symposium on Theory of computing*, pages 574–583. ACM, 2006. doi:10.1145/1132516.1132599.

13 M. Crochemore, C. S. Iliopoulos, M. Kubica, M. S. Rahman, G. Tischler, and T. Walen. Improved algorithms for the range next value problem and applications. *Theor. Comput. Sci.*, 434:23–34, 2012.

14 J. Fischer, T. Gagie, T. Kopelowitz, M. Lewenstein, V. Mäkinen, L. Salmela, and N. Välimäki. Forbidden patterns. In *LATIN*, 2012. doi:10.1007/978-3-642-29344-3_28.

15 A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 621–630, 2014.

16 D. Hermelin, A. Levy, O. Weimann, and R. Yuster. Distance oracles for vertex-labeled graphs. In *Automata, Languages, and Programming - 38th International Colloquium, ICALP (2)*, 2011. doi:10.1007/978-3-642-22012-8_39.

17 W. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. String retrieval for multi-pattern queries. In *SPIRE*, 2010.

18 W. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. Document listing for queries with excluded pattern. In *CPM*, 2012.

19 P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *30th Annual ACM Symposium on Theory of Computing*, pages 604–613, May 1998.

20 D. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *34th Annual ACM Symposium on the Theory of Computing*, pages 63–66, 2002.

21 O. Keller, T. Kopelowitz, S. Landau Feibish, and M. Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.

22 O. Keller, T. Kopelowitz, and M. Lewenstein. Range non-overlapping indexing and successive list indexing. In *Algorithms and Data Structures, 10th International Workshop, WADS*, pages 625–636, 2007. doi:10.1007/978-3-540-73951-7_54.

23 T. Kopelowitz, S. Pettie, and E. Porat. Higher lower bounds from the 3SUM conjecture. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1272–1287, 2016. doi:10.1137/1.9781611974331.ch89.

24 R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 791–801, January 2004.

25 K. Green Larsen, J. I. Munro, J. Sindahl Nielsen, and S. V. Thankachan. On hardness of several string indexing problems. In *CPM*, 2014.

26 M. Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 267–302, 2013.

27 J. I. Munro, Y. Nekrich, and J. Scott Vitter. Fast construction of wavelet trees. In *String Processing and Information Retrieval - 21st International Symposium, SPIRE*, pages 101–110, 2014.

28 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666. ACM/SIAM, 2002.

29 Y. Nekrich and G. Navarro. Sorted range reporting. In *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops*, pages 271–282, 2012.

30 M. Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *STOC*, pages 603–610. ACM, 2010. doi:10.1145/1806689.1806772.

31 C. Yu, W. Hon, and B. Wang. Improved data structures for the orthogonal range successor problem. *Comput. Geom.*, 44(3):148–159, 2011.

The Nearest Colored Node in a Tree

Paweł Gawrychowski[*1], Gad M. Landau[†2], Shay Mozes[‡3], and Oren Weimann[‡4]

1 **University of Warsaw,** gawry@mimuw.edu.pl
2 **University of Haifa,** landau@cs.haifa.ac.il
3 **IDC Herzliya,** smozes@idc.ac.il
4 **University of Haifa,** oren@cs.haifa.ac.il

Abstract

We start a systematic study of data structures for the nearest colored node problem on trees. Given a tree with colored nodes and weighted edges, we want to answer queries (v, c) asking for the nearest node to node v that has color c. This is a natural generalization of the well-known nearest marked ancestor problem. We give an $O(n)$-space $O(\log\log n)$-query solution and show that this is optimal. We also consider the dynamic case where updates can change a node's color and show that in $O(n)$ space we can support both updates and queries in $O(\log n)$ time. We complement this by showing that $O(\text{polylog}\, n)$ update time implies $\Omega(\frac{\log n}{\log\log n})$ query time. Finally, we consider the case where updates can change the edges of the tree (link-cut operations). There is a known (top-tree based) solution that requires update time that is roughly linear in the number of colors. We show that this solution is probably optimal by showing that a strictly sublinear update time implies a strictly subcubic time algorithm for the classical all pairs shortest paths problem on a general graph. We also consider versions where the tree is rooted, and the query asks for the nearest ancestor/descendant of node v that has color c, and present efficient data structures for both variants in the static and the dynamic setting.

1998 ACM Subject Classification E.1 Data Structures – Trees. F.2 Analysis of Algorithms and Problem Complexity. F.2.2 Nonnumerical Algorithms and Problems – Pattern Matching.

Keywords and phrases Marked ancestor, Vertex-label distance oracles, Nearest colored descendant, Top-trees

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.25

1 Introduction

We consider a number of problems on trees with colored nodes. Each of these problems can be either static, meaning the color of every node of a tree T on n nodes is fixed, or dynamic, meaning that an update can change a node's color (but the tree itself does not change). The edges of T may have arbitrary nonnegative lengths and $\mathsf{dist}(u, v)$ denotes the total length of the unique path connecting u and v. Depending on the version of the problem, given a node u and a color c we are interested in:

The nearest colored ancestor: the first node v on the u-to-root path that has color c.

[*] Currently holding a post-doctoral position at Warsaw Center of Mathematics and Computer Science.
[†] Partially supported by the National Science Foundation Award 0904246, Israel Science Foundation grant 571/14, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.
[‡] Partially supported by Israel Science Foundation grant 794/13.

The nearest colored descendant: the node v of color c such that the v-to-root path goes through u and the distance from u to v is as small as possible.

The nearest colored node: the node v of color c such that the distance from u to v is as small as possible.

In the static case, if the number of colors is constant, there is a trivial (and optimal) solution for all three problems with $O(n)$-space and $O(1)$-query. In fact, this can be achieved even for a logarithmic number of colors [7]. For an arbitrary number of colors, a lower bound of $\Omega(\log\log n)$-query for any $O(n \operatorname{polylog} n)$-space solution to each of these problems (in fact, even on strings) follows from a simple reduction from the well known predecessor problem. We present tight $O(n)$-space $O(\log\log n)$-query solutions to all three problems. To achieve this, for every color c we construct a separate tree $T(c)$. If there are total s nodes of color c then $T(c)$ is only of size $O(s)$ but (after augmenting it with appropriate additional data) it captures for all n nodes of the original tree their nearest node of color c.

In the dynamic case, the nearest colored ancestor problem has been studied by Alstrup-Husfeldt-Rauhe [3] who gave a solution with $O(n)$-space, $O(\frac{\log n}{\log\log n})$-query, and $O(\log\log n)$-update. They also gave a lower bound stating that $O(\operatorname{polylog} n)$-update requires $\Omega(\frac{\log n}{\log\log n})$-query. This holds even when the number of colors is only two (then a node is either marked or unmarked and the problem is known as the marked ancestor problem). We show that this lower bound (with the same statement and only two colors) extends to both the nearest colored node and the nearest colored descendant. For upper bounds, we show that the nearest colored node problem can be solved with $O(n)$-space, $O(\log n)$-update, and $O(\log n)$-query. Our solution can be seen as a variant of the centroid decomposition tweaked to guarantee some properties of top-trees.

The original top-trees of Alstrup-Holm-de Lichtenberg-Thorup [2] were designed for only two colors (i.e., for the nearest marked node problem). They achieve $O(\log n)$ query and update and also support updates that insert and delete edges (i.e., maintain a forest under link-cut operations). The straightforward generalization of top-trees from two to k colors increases the space dramatically to $O(nk)$. We believe it is possible to improve this to $O(n)$ using similar ideas to those we present here. However, because we do not allow link-cut operations, compared to top-trees our solution is simpler. Moreover, our query time can be improved to (optimal) $O(\frac{\log n}{\log\log n})$ at the cost of increasing the update time by a $\log^{\epsilon} n$ factor and the space by a $\log^{1+\epsilon} n$ factor. Whether such an improvement is possible with top trees remains open. We note that in both the $O(nk)$ and the $O(n)$ space solutions with top-trees, while queries and color-changes require $O(\log n)$ time the time for link/cut is $O(k \cdot \log n)$. This can be significant since k can be as large as n (we emphasise that our solution does not support link/cut at all). We show that $\tilde{O}(k)$ is probably optimal by showing that $O(k^{1-\varepsilon})$ query and update time implies an $O(n^{3-\varepsilon})$ solution for the classical all pairs shortest paths problem on a general graph with n vertices. The non existence of such an algorithm has recently been widely used as an assumption with various consequences [19].

Finally, for the nearest colored descendant problem, we give a solution with $O(\frac{\log n}{\log\log n})$-query and $O(\log^{2/3+\epsilon} n)$-update by reducing the problem to 3-sided emptiness queries on points in the plane. We then show that the $O(\operatorname{polylog} n)$-update $\Omega(\frac{\log n}{\log\log n})$-query lower bound of [3] also applies to the nearest colored descendant problem by giving a reduction from nearest colored ancestor to nearest colored descendant.

Related work. The approximate version of the nearest colored node problem (where we settle for approximate distances) has recently been studied (as the vertex-to-label distance

query problem) in general graphs [8, 12, 20] and in planar graphs [1, 13, 14]. In fact, the query-time in [14] is dominated by a $O(\log\log n)$ nearest colored node query on a string (which we now know is optimal).

Preliminaries. A *predecessor* structure is a data structure that stores a set of n integers $S \subseteq [0, U]$, so that given $x \in [0, U]$ we can determine the largest $y \in S$ such that $y \leq x$. It is known [15] that for $U = n^2$ any predecessor structure of $O(n \operatorname{polylog} n)$-space requires $\Omega(\log\log n)$-query, and that linear-size structures with such query-time exist [16, 18].

A *Range Minimum Query* (RMQ) structure on an array $A[1, \ldots, n]$ is a data structure for answering queries $\min\{A[i], \ldots, A[j]\}$. When the array A is static, RMQ can be optimally solved in $O(n)$-space and $O(1)$ query [5, 6, 11]. In the dynamic case, we allow updates that change the value of array elements. When the query range is restricted to be a suffix $A[i, \ldots, n]$ we refer to the problem as the Suffix Minimum Query (SMQ) problem.

A *Lowest Common Ancestor* (LCA) structure on a rooted tree T is a data structure for finding the common ancestor of two nodes u, v with the largest distance from the root. For static trees, LCA is equivalent to RMQ and thus can be solved in $O(n)$-space and $O(1)$-query.

A *perfect hash* structure stores a collection of n integers S. Given x we can determine if $x \in S$ and return its associated data. There exists $O(n)$-space, $O(1)$-query perfect hash structure [10], which can be made dynamic with $O(1)$-update (expected amortized) [9].

2 Static Upper Bounds

We root the tree at node 1 and assign pre- and post-order number $\mathsf{pre}(u), \mathsf{post}(u) \in [1, 2n]$ to every node u. All these numbers are distinct, $[\mathsf{pre}(u), \mathsf{post}(u)]$ is a laminar family of intervals, and u is an ancestor of v if and only if $\mathsf{pre}(v) \in (\mathsf{pre}(u), \mathsf{post}(u))$. We order edges outgoing from every node according to the preorder numbers of the corresponding nodes.

We assume the colors are represented by integers in $[1, n]$. We will construct a separate additional structure for every possible color c. The size of the additional structure will be always proportional to the number of nodes of color c, which sums up to $O(n)$ over all colors c. Below we describe the details of the additional structure for every version of the problem.

Nearest colored descendant. Let $v_1, v_2, \ldots, v_s$ be all nodes of color c sorted so that $\mathsf{pre}(v_1) < \mathsf{pre}(v_2) < \ldots < \mathsf{pre}(v_s)$. We insert the preorder numbers of all these nodes into a predecessor structure, so that given an interval $[x, y]$ we can determine the range $v_i, v_{i+1}, \ldots, v_j$ consisting of all nodes with preorder numbers from $[x, y]$ in $O(\log\log n)$ time. Additionally, we construct an array $D[1..s]$, where $D[i] = \mathsf{dist}(1, v_i)$. The array is augmented with an RMQ structure. To answer a query, we use the predecessor structure to locate the range consisting of nodes v such that $\mathsf{pre}(v) \in [\mathsf{pre}(u), \mathsf{post}(u)]$. Then, if the range is nonempty, a range minimum query allows us to retrieve the nearest descendant of u with of color c. The total query time is hence $O(\log\log n)$.

Nearest colored ancestor. Let $v_1, v_2, \ldots, v_s$ be all nodes of color c. We insert all their pre- and postorder numbers into a predecessor structure. Additionally, for every i we store (in an array) the nearest ancestor with the same color for the node v_i (or null if such ancestor does not exist). To answer a query, we use the predecessor structure to locate the predecessor of $\mathsf{pre}(u)$. There are two cases:

1. The predecessor is $\mathsf{pre}(v_i)$ for some i. Because $[\mathsf{pre}(v), \mathsf{post}(v)]$ create a laminar family, either $\mathsf{pre}(u) \in [\mathsf{pre}(v_i), \mathsf{post}(v_i)]$ and v_i is the answer, or u has no ancestor of color c.

2. The predecessor is $\mathsf{post}(v_i)$ for some i. Consider an ancestor u' of u with the same color. Then $\mathsf{pre}(u') < \mathsf{post}(v_i)$, so u' is also an ancestor of v_i. Similarly, consider an ancestor v' of v_i with the same color, then $\mathsf{post}(v') > \mathsf{pre}(u)$ so v' is also an ancestor of u. Therefore, the nearest ancestor of color c is the same for u and v_i, hence we can return the answer stored for v_i.

The query time is hence again $O(\log \log n)$.

Nearest colored node. We define the subtree induced by color c, denoted $T(c)$, as follows. Let $v_1, v_2, \ldots, v_s$ be all nodes of color c. $T(c)$ consists of all nodes v_i together with the lowest common ancestor of every pair of nodes v_i and v_j. The parent of $u \in T(c)$ is defined as the nearest ancestor v of $u \in T$ such that $v \in T(c)$ as well; if there is no such node, u is the root of $T(c)$ (there is at most one such node). Thus, an edge $(u, v) \in T(c)$ corresponds to a path from u to v in T.

▶ **Lemma 1.** *$T(c)$ consists of at most $2s - 1$ nodes and can be constructed in $O(s)$ time assuming that we are given a list of all nodes of color c sorted according to their preorder numbers and a constant time LCA built for T.*

Proof. Let $v_1, v_2, \ldots, v_s$ be the given list of nodes of color c. By assumption, $\mathsf{pre}(v_1) < \mathsf{pre}(v_2) < \ldots < \mathsf{pre}(v_s)$. We claim that $T(c)$ consists of all nodes v_i and the lowest common ancestor of every v_i and v_{i+1}. To prove this, consider two nodes v_i and v_j such that $i < j$ such that their lowest common ancestor u is different than v_i and v_j. Then, u is a proper ancestor of v_i and v_j, and furthermore v_i is a descendant of u_a and v_j a descendant of u_b, where $a < b$ and $u_1, u_2, \ldots, u_\ell$ is an ordered list of the children of u. v_i can be replaced by the node $v_{i'}$ of color c with the largest preorder number in the subtree rooted at u_a. Then the lowest common ancestor of $v_{i'}$ and $v_{i'+1}$ is still u, so it is indeed enough to include only the lowest common ancestor of such pairs of nodes and the bound of $2s - 1$ follows.

To construct $T(c)$ we need to determine its set of nodes and edges. Determining the nodes is easy by the above reasoning. To determine the edges, we use a method similar to constructing the Cartesian tree of a sequence: we scan $v_1, v_2, \ldots, v_s$ from the left to right while maintaining the subtree induced by $v_1, v_2, \ldots, v_i$. We keep the rightmost path of the current subtree on a stack, with the bottommost edge on the top. To process the next v_{i+1}, we first calculate its lowest common ancestor with v_i, denoted x. Then, we pop from the stack all edges (u, v) such that u and v are both below (or equal to) x in T. Finally, we possibly split the edge on the top of the stack into two and push a new edge onto the stack. The amortized complexity of every step is constant, so the total time is $O(s)$. ◀

The first part of the additional structure is the nearest node of color c stored for every node of $T(c)$. Given a node u, we need to determine its nearest ancestor u' such that $u' \in T(c)$ or u' lies strictly inside some path corresponding to an edge of $T(c)$. In the latter case, we want to retrieve the endpoints of that edge. This is enough to find the answer, as any path from u to a node of color c must necessarily go through u' (because u' is the lowest ancestor of u such that the subtree rooted at u' contains at least one node of color c, and a simple path from u to a node of color c must go up as long as the subtree rooted at the current node does not contain any node of color c), and then either $u' \in T(c)$ and we have the answer for u' or the path continues towards one of the endpoints of the edge of $T(c)$ strictly containing u' (because the subtrees hanging off the inside of a path corresponding to an edge of $T(c)$ do not contain any nodes of color c). Hence after having determined u' we need only constant time to return the answer.

To determine u', we use the structure for the nearest colored ancestor constructed for a subset of $O(s)$ marked nodes of T. These marked nodes are all nodes of T corresponding to the nodes of $T(c)$, and additionally, for every path $u_1 \to u_2 \to \ldots \to u_\ell$ corresponding to an edge of $T(c)$, the node u_2 (where u_1 is closer to the root than u_ℓ and $\ell \geq 2$). For every marked node of the second type we store the endpoints (u_1, u_ℓ) of its corresponding edge of $T(c)$. Then, locating the nearest marked ancestor of u allows us to determine that the sought nearest ancestor u' is a node of $T(c)$, or find the edge of $T(c)$ strictly containing u'. By plugging in the aforementioned structure for the nearest colored ancestor, we obtain the answer in $O(\log\log n)$ time with a structure of size $O(s)$.

This concludes the description of our static solution. Before moving on to the dynamic case, we note that the above solution can be easily extended to the case where every node $v \in T$ has an associated set of colors $\mathsf{C}(v)$ and instead of looking for a node of color c we look for a node v such that $c \in \mathsf{C}(v)$.

3 Dynamic Upper Bounds

In the dynamic setting, we allow updates to change a node's color. To be even more general, we assume that every node $v \in T$ has an associated (dynamically changing) set of colors $\mathsf{C}(v)$, and an update can either insert or remove a color c from the current set $\mathsf{C}(v)$.

Nearest colored descendant. As in the static case, we construct a separate structure for every possible color c. We also maintain a mapping from the set of colors to their corresponding structures. Let $v_1, v_2, \ldots, v_s$ be all nodes of color c. We create a set of points of the form $(\mathsf{pre}(v_i), \mathsf{dist}(1, v_i))$. Then, a nearest colored descendant query can be answered by locating the point with the smallest y-coordinate in the slab $[\mathsf{pre}(u), \mathsf{post}(u)] \times (-\infty, \infty)$. We store the points in a fully dynamic 3-sided emptiness structure of Wilkinson [17]. The structure answers a 3-sided emptiness query by locating the point with the smallest y-coordinate in a slab $[x_1, x_2] \times (-\infty, \infty)$ in $O(\frac{\log n}{\log\log n})$ time and can be updated by inserting and removing points in $O(\log^{2/3+\epsilon} n)$ time, with both the update and the query time being amortized. Consequently, we obtain the same bounds for the nearest colored descendant.

Nearest colored ancestor. This has been considered by Alstrup-Husfeldt-Rauhe [3]. The query time is $O(\frac{\log n}{\log\log n})$ and the update time $O(\log\log n)$. While not explicitly stated in the paper, the total space is linear.

Nearest colored node. Our data structure is based on a variant of the centroid decomposition. That is, we recursively decompose the tree into smaller and smaller pieces by successively removing nodes. The difference compared to the standard centroid decomposition is that each of the obtained smaller trees has up to two appropriately defined boundary nodes (similarly to the decomposition used in top-trees).[1]

We assume the degree of every node is at most 3. This can be achieved by standard ternerization with zero length edges. The basis of our recursive decomposition is the following well-known fact.

▶ Fact 1. In any tree T on n nodes there exists a node $c \in T$ such that $T \setminus \{c\}$ is a collection of trees of size at most $\frac{n}{2}$ each.

[1] Using standard centroid decomposition leads to update time of $O(\log^2 n)$, compared to $O(\log n)$ when controlling the number of boundary nodes.

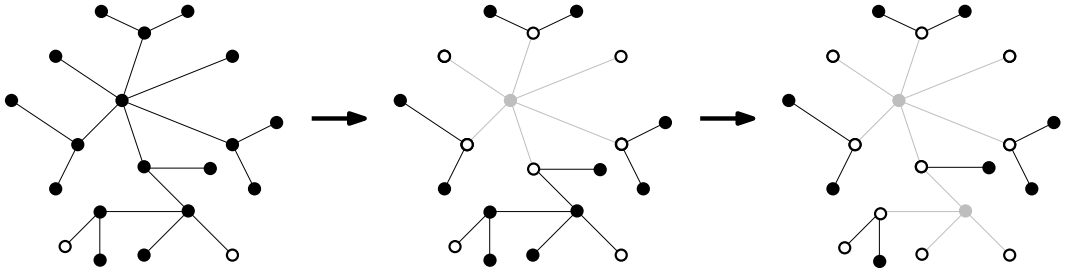


Figure 1 Schematic depiction of a single step of our centroid decomposition. After removing the grayed out node and its adjacent edges we obtain 6 pieces. One of them contains three boundary nodes and hence needs to be further partitioned into 4 smaller pieces.

We apply it recursively. The input to a single step of the recursion is a tree T on n nodes with at most two distinguished *boundary* nodes. We use Fact 1 to find node $c_1 \in T$ such that $T \setminus \{c_1\}$ is a collection of smaller trees $T_1, T_2, \ldots$. Each neighbor of c_1 in the original tree becomes a boundary node in its smaller tree T_i. A boundary node $u \in T$ such that $u \neq c_1$ is also a boundary node in its smaller tree T_i. Because T contains at most two boundary nodes, at most one smaller tree T_i contains three boundary nodes, while all other smaller trees contain at most two boundary nodes. If such T_i containing three boundary nodes u_1, u_2, u_3 exists, we further partition it into even smaller trees $T'_1, T'_2, \ldots$. This is done by finding a node $c_2 \in T_i$ which, informally speaking, separates all u_1, u_2, u_3 from each other. Formally speaking, we take c_2 to be any node belonging to all three paths $u_1 - u_2, u_1 - u_3, u_2 - u_3$ (intersection of such three paths is always nonempty). Then, $T_i \setminus \{c_2\}$ is a collection of trees $T'_1, T'_2, \ldots$ such that each T'_j contains at most one of the nodes u_1, u_2, u_3. Finally, each neighbor of c_2 in T_i becomes a boundary node in its smaller tree T'_j; see Figure 1.

▶ **Lemma 2.** *Given a tree T on n nodes with at most two boundary nodes b_1, b_2, we can find two nodes $c_1, c_2 \in T$, called the centroids of T, such that $T \setminus \{c_1, c_2\}$ is a collection of trees $T_1, T_2, \ldots$ with the property that each T_i consists of at most $\frac{n}{2}$ nodes and contains at most two boundary nodes, which are defined as nodes corresponding to the original boundary nodes of T or nodes adjacent to c_1 or c_2 in T.*

Let T_0 denote the original input tree. We apply Lemma 2 to T_0 recursively until the tree is empty. The resulting recursive decomposition of T_0 can be described by a decomposition tree $\mathcal{T}$ as followed. Each node of $\mathcal{T}$ corresponds to a subtree of T_0. The root r of $\mathcal{T}$ corresponds to T_0. The children of a node $u \in \mathcal{T}$, whose corresponding subtree of T_0 is T, are the recursively defined decomposition trees of the smaller trees T_i obtained by removing the centroid nodes from T with Lemma 2. For a node $u \in \mathcal{T}$ whose corresponding subtree is T we define $\mathsf{C}(u)$ to be $\mathsf{C}(c_1) \cup \mathsf{C}(c_2)$, where c_1 and c_2 are the centroids of T. Because the size of the tree decreases by a factor of two in every step, the depth of $\mathcal{T}$ is at most $\log n$. We will sometimes abuse notation and say that a tree T in the decomposition is the parent of T' if the node of $\mathcal{T}$ whose corresponding tree is T is the parent of the node of $\mathcal{T}$ whose corresponding tree is T'. This concludes the description of our recursive decomposition.

We now describe the information maintained in order to implement dynamic nearest colored node queries. For every tree T in the decomposition, every boundary node b of T, and every color c such that $c \in \mathsf{C}(v)$ for some $v \in T$, we store the node of T with color c that is nearest to b. Observe that, since the degree is bounded, this information can be used to compute in constant time the nearest node with color c to each centroid c_i of T, by considering the nearest nodes with color c to each of the adjacent (to c_1 or to c_2) boundary

nodes of the children T_i of T in $\mathcal{T}$.

For every node $v \in T_0$ we store a pointer to the unique node of $\mathcal{T}$ in which v is a centroid. We also preprocess the original tree T_0 so that the distance between any two nodes can be calculated in constant time: we root the tree at node 1, construct an LCA structure, and store $\mathsf{dist}(1, v)$ for every $v \in T_0$. Such preprocessing actually allows us to compute the distance between any two nodes in any of the smaller trees in the decomposition.

Queries. Given a tree T in the decomposition, a node $v \in T$ and a color c, we need to find the node $u \in T$ with color c that is nearest to v.

Let c_i $(i = 1, 2)$ be the centroids of T. Either some c_i lies on the v-to-u path in T, or v and u belong to the same child T_i of T. In the former case u is the closest node to c_i in T with color c. Note that this information is already stored. In the latter case, the query is reduced to a query in T_i.

It follows that, in order to find the closest node to v with color c in T_0, it suffices to consider the closest nodes with color c to each of the centroids of each of the trees on the path in $\mathcal{T}$ from the node of $\mathcal{T}$ in which v is a centroid to the root of $\mathcal{T}$. There are $O(\log n)$ such centroids, and each of them can be checked in constant time using the stored information.

Updates. Consider adding or removing color c from $\mathsf{C}(v)$. We implement the updates in a bottom-up fashion along the same path used for the query. Subtrees on this path are the only ones in the decomposition containing v, so only their information should be updated.

Repairing the information for the boundary nodes of a subtree T along the path in $\mathcal{T}$ is done in a similar manner to that of the query. For each boundary node b_i of T $(i = 1, 2)$, we need to find the nearest node $u \in T$ with color c. Let c_j $(j = 1, 2)$ be the centroids of T. Let T_i denote the child of T in $\mathcal{T}$ that contains b_i. Either b_i and u both belong to T_i, or $u = c_j$ for some j, or u is in some other child T_ℓ of T and some c_j lies on the b_i-to-u path in T. In all cases we can use the information stored at the children of T to correctly determine the information stored at T. If $b_i, u \in T_i$ then b_i is a boundary node of T_i, so we use the information stored for T_i. If $u = c_j$ then we verify that $c \in \mathsf{C}(c_j)$. Finally, in the last case, the closest node to b_i with color c in T_ℓ is also the closest node to the boundary node of T_ℓ adjacent (in T) to c_j, so we use the information stored for T_ℓ.

Summary. To summarize, both the query and the update time is $O(\log n)$. The space is $O(\log n \cdot \sum_{v \in T_0} |\mathsf{C}(v)|)$, because every $c \in C(v)$ contributes constant space at every level.

Decreasing the space. The space can be reduced to $O(n + \sum_{v \in T_0} |\mathsf{C}(v)|)$. Let T be a tree in the decomposition. Recall that for each boundary node $u \in T$ and color c such that $c \in \mathsf{C}(v)$ for some $v \in T$ we maintain the nearest node of T with color c. Hence, every $c \in \mathsf{C}(v)$ might contribute constant space at every tree T such that $v \in T$. Now we describe how this can be avoided by maintaining, for every color c, a separate structure of size proportional to the number of nodes with color c.

Recall that we extend the colors of nodes in the original tree T_0 to color sets of nodes of the decomposition tree $\mathcal{T}$. For a node $u \in \mathcal{T}$ that is associated with subtree T of T_0 we define u's color set to be the union of the color sets of the centroids of T. For every color c we maintain the subtree of $\mathcal{T}$ induced by color c (cf. Section 2 for definition of induced), denoted $\mathcal{T}(c)$. Before we describe how these subtrees can be efficiently maintained, we describe how to use $\mathcal{T}(c)$ instead of $\mathcal{T}$ to perform queries and updates.

Consider a query (v, c) and let u be the node of $\mathcal{T}$ in which v is a centroid. The query traverses the ancestors of u. At each such ancestor $u' \in \mathcal{T}$, we iterate through the centroids c_i $(i = 1, 2)$ and consider their nearest node with color c as candidate for the answer. The nearest node is either the centroid itself, or the nearest node with color c to a boundary node of a child $u'' \in \mathcal{T}$ of u'. In the former case, $u' \in \mathcal{T}(c)$. In the latter case, $u' \notin \mathcal{T}(c)$. If also $u'' \notin \mathcal{T}(c)$ then, by definition of $\mathcal{T}(c)$, $c \notin \mathsf{C}(u'')$ and u'' has at most one child $u''' \in \mathcal{T}$ containing nodes with color c in its corresponding subtree of T. Hence instead of iterating through the boundary nodes of u'' we can iterate through the boundary nodes of u'''. By repeating this reasoning, u'' can be replaced by its highest descendant belonging to $\mathcal{T}(c)$ (such highest descendant is uniquely determined, unless the subtree of T corresponding to u'' has no nodes with color c). Consequently, the queries can be modified to operate on $\mathcal{T}(c)$ instead of $\mathcal{T}$: we locate the first ancestor $u' \in \mathcal{T}$ of u such that $u' \in \mathcal{T}(c)$ (if there is none, we take the root of $\mathcal{T}(c)$ as u'), and then iterate through all ancestors of u' in $\mathcal{T}(c)$. For each such ancestor u'', we consider as candidates for the answer its centroid nodes c_i $(i = 1, 2)$ and also the nearest node with color c to every boundary node of each child of u'' in $\mathcal{T}(c)$. The same reasoning allows us to recalculate, upon an update, the information stored at $u \in \mathcal{T}(c)$ using the information stored at all of its children in $\mathcal{T}(c)$.

By Lemma 1, the size of the subtree induced by color c is at most $2|\{v \in T : c \in \mathsf{C}(v)\}| - 1$. Summing over all colors we obtain that the total size of all induced subtrees is $2\sum_{v \in T_0} |\mathsf{C}(v)|$. We still need to show how to maintain them and also how to efficiently locate the first ancestor $u' \in \mathcal{T}$ of u such that $u' \in \mathcal{T}(c)$. The latter is implemented with a nearest colored ancestor structure. We only describe how to update $\mathcal{T}(c)$ after adding c to some $\mathsf{C}(v)$, where $v \in T_0$, and do not change $\mathcal{T}(c)$ after removing c (so our trees will be in fact larger than necessary). Whenever the total size of all maintained subtrees exceeds $4\sum_{v \in T_0} |\mathsf{C}(v)|$, we rebuild the whole structure. This does not increase the amortized complexity of an update and can be deamortized using the standard approach of maintaining two copies of the structure.

After adding c to some $\mathsf{C}(v)$, where $v \in T_0$, we might also need to include c in $\mathsf{C}(u)$ for some $u \in \mathcal{T}$, thus changing $\mathcal{T}(c)$. Inspecting the proof of Lemma 1 we see that the change consists of two parts: we need to include u in $\mathcal{T}(c)$, and in particular insert it onto the sorted list of nodes of $\mathcal{T}$ of color c. Then, we might also need to include the lowest common ancestor of u and its predecessor on the list, and also the lowest common ancestor of u and its successor there. We implement the list with a balanced search tree, so that all these new nodes can be generated in $O(\log n)$ time. We also need to generate new edges (or, more precisely, split some existing edges into two and possibly attach a new edge to the new middle node). This is easy to do if we are able to efficiently find the edge of $\mathcal{T}(c)$ corresponding to a path containing a given node $u \in \mathcal{T}$. To this end, we also maintain a list of all nodes of $\mathcal{T}(c)$ sorted according to their preorder numbers in $\mathcal{T}$. Then binary searching over the list gives us the highest descendant of u belonging to $\mathcal{T}(c)$. By implementing the list with a balanced search tree we can hence find such an edge in $O(\log n)$ time. Thus, the update and the query time is still $O(\log n)$ and the space linear.

Decreasing the query time. The query time can be decreased to $O(\frac{\log n}{\log \log n})$, which is optimal, at the cost of increasing the update time to $O(\log^{1+\epsilon} n)$ and the space to $O(\log^{1+\epsilon} n \sum_{v \in T} |\mathsf{C}(v)|)$.

For trees of constant degree, Lemma 2 decomposes T into a constant number of trees, each of size $\frac{n}{2}$, by removing at most two nodes. By iterating the lemma $\epsilon \log \log n$ times we obtain the following.

▶ **Lemma 3.** *Given a tree T on n nodes with at most two boundary nodes, we can find $O(\log^\epsilon n)$ centroid nodes $c_1, c_2, \ldots \in T$ such that $T \setminus \{c_1, c_2, \ldots\}$ is a collection of trees $T_1, T_2, \ldots$ with the property that each T_i consists of at most $\frac{n}{\log^\epsilon n}$ nodes and contains at most two boundary nodes, which are defined as nodes corresponding to the original boundary nodes of T or nodes adjacent to any c_i in T.*

We apply Lemma 3 recursively. Now the depth of the recursion is $O(\frac{\log n}{\log\log n})$.

Note that, because the number of centroids c_i and trees T_i is no longer constant, it is no longer true that the nearest node to centroid c_i with color c in T can be computed in $O(1)$ time from the information stored for boundary nodes of the T_is. Therefore, to implement query (v, c) in $O(\frac{\log n}{\log\log n})$ time, we maintain explicitly, for each centroid node c_i, its nearest node of T with color c. This allows us to process the case when $v = c_i$ in constant time. If v is not a centroid of T, then $v \in T_j$ for some j. We recurse on T_j. The only remaining possibility is that the sought node u does not belong to T_j. In such case, the path from v to u must go through one of the boundary nodes of T_j. Each of these boundary nodes is adjacent to a constant number of the centroid nodes c_i of T (because of the constant degree assumption). We iterate through every such centroid c_i and consider its nearest node with color c as a candidate for the answer in constant total time.

Implementing updates is again done in a bottom-up fashion. However, now we also need to recalculate the nearest node with color c to every centroid node c_i. Recalculating the nearest node with color c (to either a boundary or a centroid node) takes now $O(\log^\epsilon n)$ time, because we need to consider boundary nodes of up to $O(\log^\epsilon n)$ subtrees T_i and also $O(\log^\epsilon n)$ centroid nodes. Hence the total update time at every level of recursion is $O(\log^{2\epsilon} n)$. By adjusting ϵ we get that the total update time is $O(\log^{1+\epsilon} n)$.

4 Lower Bounds

Static nearest colored node, descendant, and ancestor. First we consider the static nearest colored node. In such case, there is a lower bound stating that $O(n \operatorname{polylog} n)$ space requires $\Omega(\log\log n)$ query time. In fact, the lower bound already applies for paths, and follows easily from Belazzougui and Navarro [4]: they show (via reduction from predecessor [15]) that any data structure that uses $O(n \operatorname{polylog} n)$ space to represent a string S of length n over alphabet $\{1, \ldots, n\}$ must use time $\Omega(\log\log n)$ to answer rank queries. A $\mathsf{rank}_\sigma(i)$ query asks for the number of times the letter σ appears in $S[1, \ldots, i]$. The reduction to nearest colored node is trivial: each letter corresponds to a color, we create a path on n nodes where the color of the i-th node is $S[i]$, and additionally the node stores $\mathsf{rank}_{S[i]}(i)$. Then, to calculate an arbitrary $\mathsf{rank}_\sigma(i)$, we consider the i-th node and find its nearest node of color σ. Then, if that nearest node is on the left of i, we return its stored answer, and otherwise we return its stored answer decreased by one. This also shows that one cannot beat $O(\log\log n)$ time with a structure of size $O(n \operatorname{polylog} n)$ for the static nearest colored descendant and ancestor.

In all dynamic problems, the lower bounds hold even if we have only two colors, that is, every node is marked or not.

Dynamic nearest marked node and ancestor. We next show that the following lower bound of Alstrup-Husfeldt-Rauhe [3] for marked ancestor also applies to dynamic nearest marked node. Notice that Theorem 4 implies that any $O(\operatorname{polylog} n)$ update time requires $\Omega(\frac{\log n}{\log\log n})$ query time. In the marked ancestor problem, the query is to detect if a node has a marked ancestor, and an update marks or unmarks a node, so we immediately obtain a lower bound for the dynamic nearest marked ancestor.

▶ **Theorem 4** ([3]). *For the marked ancestor problem, if t_u is the update time and t_q is the query time then*

$$t_q = \Omega\left(\frac{\log n}{\log t_u + \log\log n}\right)$$

The lower bound holds under amortization and randomization.

The proof of Theorem 4 uses a (probabilistic) sequence of operations (mark/unmark/marked ancestor query) on an unweighted complete tree T on n leaves and out-degree ≥ 2. To show that the bounds of Theorem 4 also apply to dynamic nearest marked node, we add edge weights to T that increase exponentially with depth: edges outgoing from a node at depth d has weight 2^d. This way, if a node has a marked ancestor, then its nearest marked node is necessarily the nearest marked ancestor (because in the worst case the distance to the nearest marked ancestor is $2^0 + 2^1 + \ldots + 2^{d-1}$, while the distance to any proper descendant is at least 2^d). Hence the marked ancestor problem reduces to nearest marked node. In fact, it is possible to achieve a reduction without using weights by replacing each weight W with a path of W nodes. Since T is balanced, this will increase the space of T to be $O(n^2)$ which is fine since the bound of Theorem 4 is independent of space.

Dynamic nearest marked descendant. We next show that the bounds of Theorem 4 also apply to the case of nearest marked *descendant*. This requires three simple reductions:

1. dynamic existential marked ancestor → planar dominance emptiness.
   Dynamic existential marked ancestor is a simpler variant of the dynamic marked ancestor problem where a query does not need to find the nearest marked ancestor but only to report if there exists a marked ancestor. In fact, the proof [3] of the lower bound of Theorem 4 is for the dynamic existential marked ancestor problem. In the planar dominance emptiness problem, we need to maintain a set $S \subseteq [n]^2$ of points in the plane under insertions and deletions, such that given a query point (x, y) we can determine if there exits a point (x', y') in S that dominates (x, y) (i.e., $x' \geq x$ and $y' \geq y$). As shown in [3], since we can assume the input tree is balanced, there is a very simple reduction obtained by embedding the tree nodes as points in the plane where node (x', y') is an ancestor of node (x, y) iff $x' \geq x$ and $y' \geq y$.
2. planar dominance emptiness → dynamic SMQ.
   In the dynamic SMQ problem we are given an array $A[1, \ldots, n]$ where each entry $A[i]$ is in $\{1, \ldots, n\}$. An update (i, j) changes the value of $A[i]$ to be j, and a suffix maximum query SMQ(i) returns the maximum value in $A[i, \ldots, n]$. The reduction is as follow: For each x in $\{1, \ldots, n\}$ we set $A[x]$ to be the largest y s.t $(x, y) \in S$ (or zero if there is no $(x, y) \in S$). It is easy to see that a dominance query (x, y) in S reduces to checking whether SMQ$(x) > y$. Upon an insertion or deletion of a point (x, y) we need to update $A[x]$. For this we need to maintain for every x the maximum y s.t. $(x, y) \in S$. This can be done in $O(\log\log n)$ time and linear space using a predecessor structure for each x.
3. dynamic SMQ → dynamic nearest marked descendant.
   The reduction is as follows: Given an array A, we build a tree T of size n^2. The tree is composed of a spine $v_1, \ldots, v_n$ where each v_i has two children: the spine node v_{i+1} and the unique path $v_{i,n} \to v_{i,n-1} \to \cdots \to v_{i,1}$. The weight of each spine edge (v_i, v_{i+1}) is 1 and the weight of each non-spine edge $(v_{i,j}, v_{i,j-1})$ is n (again, we could replace weight n with n weight-1 edges, which increases $|T|$ to n^3). In each path $v_{i,n} \to v_{i,n-1} \to \cdots \to v_{i,1}$ there is exactly one marked node: If $A[i] = j$ then the marked node is $v_{i,j}$. It is easy to see that SMQ(i) indeed corresponds to the nearest marked descendant of v_i.

Dynamic nearest colored node and descendant with link-cut operations. Recall that, to support insertion and deletion of edges (i.e., maintain a forest under link and cut operations), the (top-tree based) solution of Alstrup-Holm-de Lichtenberg-Thorup [2] can be extended from two colors to k colors at the cost of increasing the update time to $\tilde{O}(k)$. We show that this is probably optimal. Namely, we prove (via a simple reduction) that a solution with $O(k^{1-\varepsilon})$ query and update time implies an $O(n^{3-\varepsilon})$ solution for the classical All Pairs Shortest Paths (APSP) problem on a general graph with n vertices.

Vassilevska Williams and Williams [19] introduced this approach and showed *subcubic equivalence* between APSP and a list of seven other problems, including: deciding if a graph has a triangle whose total length is negative, min-plus matrix multiplication, deciding if a given matrix defines a metric, and the replacement paths problem. Namely, they proved that either all these problems have an $O(n^{3-\varepsilon})$ solution or none of them does.

It is well known that in APSP we can assume w.l.o.g that the graph is tripartite. That is, it has $3n$ vertices partitioned into three sets A, B, C each of size n. The edges have lengths $\ell(\cdot)$ and are all in $A \times B \cup B \times C$. The problem is to determine for every pair $(a, c) \in A \times C$ the value $\min_{b \in B}(\ell(a, b) + \ell(b, c))$.

We now describe the reduction: Given a tripartite graph $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_n\}$, $C = \{c_1, \ldots, c_n\}$ we pick vertex a_1 in A and make it the root of the tree. We set its children to be $b_1, b_2, \ldots, b_n$ where the edge (a_1, b_j) has the same length $\ell(a_1, b_j)$ as in the tripartite graph. Each b_j has n children. The kth child has color c_k, and the corresponding edge has length $\ell(b_j, c_k)$. We get a tree that is of size $O(n^2)$, and has depth two. We then ask the n queries (a_1, c_k) where c_k is a color. This completes the handling of a_1. I.e., for every $c_k \in C$ we have found $\min_{b \in B}(\ell(a_1, b) + \ell(b, c_k))$. We next want to do the same for a_2. To this end we do n updates: for each i we change the root-to-b_j edge so that its length becomes $\ell(a_2, b_j)$. We then ask n queries, and so on.

Overall we do n^2 updates and n^2 queries on a tree that is of size $N = n^2$, and $k = \sqrt{N}$ colors. Assuming that APSP cannot be solved in $O(n^{3-\varepsilon})$ time, we get that, for dynamic nearest colored node on a tree of size N with link-cut operations, the query or the update must take $\Omega(\sqrt{N}) = \Omega(k)$. Note that, the updates in this reduction do not alter the topology of the tree, but only the edge lengths. Hence, the lower bound applies even to a dynamic nearest colored node problem with just edge-weight updates (and no link or cut updates).

References

1. I. Abraham, S. Chechik, R. Krauthgamer, and U. Wieder. Approximate nearest neighbor search in metrics of planar graphs. In *18th APPROX/RANDOM*, pages 20–42, 2015.
2. S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms (TALG)*, 1(2):243–264, 2005.
3. S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. *Technical Report DIKU 98-8, Dept. Comput. Sc., Univ. Copenhagen*, 1998. (Some of the results needed from here are not included in the FOCS'98 extended abstract).
4. B. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms (TALG)*, 11(4):1–21, 2010.
5. M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
6. O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

7 P. Bille, G.M. Landau, R. Raman, S. Rao, K. Sadakane, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing (SICOMP)*, 44(3):513–539, 2015.

8 S. Chechik. Improved distance oracles and spanners for vertex-labeled graphs. In *20th ESA*, pages 325–336, 2012.

9 M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

10 M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $o(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

11 D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

12 D. Hermelin, A. Levy, O. Weimann, and R. Yuster. Distance oracles for vertex-labeled graphs. In *38th ICALP*, pages 490–501, 2011.

13 M. Li, C. C. Ma, and L. Ning. $(1 + \epsilon)$-distance oracles for vertex-labeled planar graphs. In *10th TAMC*, pages 42–51, 2013.

14 S. Mozes and E.E. Skop. Efficient vertex-label distance oracles for planar graphs. In *13th WAOA*, pages 97–109, 2015.

15 M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *38th STOC*, pages 232–240, 2006.

16 P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. Announced by van Emde Boas at FOCS 1975.

17 B.T. Wilkinson. Amortized bounds for dynamic orthogonal range reporting. In *22nd ESA*, pages 842–856, 2014.

18 D.E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

19 V. Vassilevska Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *51st FOCS*, pages 645–654, 2010.

20 J. Łącki, J. Oćwieja, M. Pilipczuk, P. Sankowski, and A. Zych. The power of dynamic distance oracles: Efficient dynamic algorithms for the steiner tree. In *47th STOC*, pages 11–20, 2015.

On the Benefit of Merging Suffix Array Intervals for Parallel Pattern Matching

Johannes Fischer[1], Dominik Köppl[2], and Florian Kurpicz[3]

1 **Dept. of Computer Science, Technische Universität Dortmund, Germany**
johannes.fischer@cs.tu-dortmund.de
2 **Dept. of Computer Science, Technische Universität Dortmund, Germany**
dominik.koeppl@tu-dortmund.de
3 **Dept. of Computer Science, Technische Universität Dortmund, Germany**
florian.kurpicz@tu-dortmund.de

Abstract

We present parallel algorithms for exact and approximate pattern matching with suffix arrays, using a CREW-PRAM with p processors. Given a static text of length n, we first show how to compute the suffix array interval of a given pattern of length m in $\mathcal{O}\left(\frac{m}{p} + \lg p + \lg\lg p \cdot \lg\lg n\right)$ time for $p \leq m$. For approximate pattern matching with k differences or mismatches, we show how to compute all occurrences of a given pattern in $\mathcal{O}\left(\frac{m^k\sigma^k}{p}\max(k, \lg\lg n) + (1+\frac{m}{p})\lg p \cdot \lg\lg n + \mathrm{occ}\right)$ time, where σ is the size of the alphabet and $p \leq \sigma^k m^k$. The workhorse of our algorithms is a data structure for merging suffix array intervals quickly: Given the suffix array intervals for two patterns P and P', we present a data structure for computing the interval of PP' in $\mathcal{O}(\lg\lg n)$ sequential time, or in $\mathcal{O}(1 + \lg_p \lg n)$ parallel time. All our data structures are of size $\mathcal{O}(n)$ bits (in addition to the suffix array).

1998 ACM Subject Classification I.1.2 Algorithms

Keywords and phrases parallel algorithms, pattern matching, approximate string matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.26

1 Introduction

We consider parallelizing indexed pattern matching queries in static texts, using (compressed) suffix arrays [14, 16] and (compressed) suffix trees [17, 19] as underlying indexes. We work with the *concurrent read exclusive write* (CREW) *parallel random access machine* (PRAM) with p processors, as this model most accurately reflects the design of existing multi-core CPUs. Our starting point is that a (possibly very long) pattern can be split up into several subpatterns that can be matched in parallel. In a suffix array, this will result in p intervals, each corresponding to one of the subpatterns. These intervals, called *subintervals*, will then be combined (using a merge tree approach) to finally yield the interval for the entire pattern. From this interval, all occurrences of the pattern in the text could then be easily listed.

We also consider parallel indexed pattern matching with k errors, again using the same indexes as in the exact case. Here, we follow the approach of Huynh et al. [10], whose basic idea is to first make all possible modifications of the pattern within distance k, and then match those modifications in the suffix array. To avoid repeated computations of subintervals, a preprocessing is performed for every prefix and suffix of the pattern. We show how to parallelize both steps (preprocessing and the actual matching), resulting in a fast parallel matching algorithm. We stress that in the case of approximate pattern

matching, parallel pattern matching algorithms are of even more practical importance than in the exact case, as this is an inherently time-consuming task in the sequential case, even for short patterns.

1.1 Our Results

In the abstract, we stated the results for uncompressed suffix arrays [14] as the underlying index, which requires $\mathcal{O}(n \lg n)$ bits of space for a text of length n. However, there exists a wealth of compressed versions of suffix arrays (CSAs) [16], which are smaller (using $|\mathrm{CSA}|$ bits), but often have nonconstant access time t_{SA}. (See also Table 1 for known trade-offs.) Here, we state our results more generally, using the parameters $|\mathrm{CSA}|$ and t_{SA}.

Our first result (Thm. 8) is an index of size $|\mathrm{CSA}| + \mathcal{O}(n)$ bits that, with $p \leq m$ processors, allows us to compute the suffix array interval of a pattern of length m in $\mathcal{O}\left(t_{\mathsf{SA}}\left(\frac{m}{p} + \lg p + \lg\lg p \cdot \lg\lg n\right)\right)$ time and $\mathcal{O}(t_{\mathsf{SA}}\,(m + \min\,(p, \lg n)\,(\lg p + \lg\lg p \cdot \lg\lg n)))$ work. Our second result (Thm. 12) is an index of the same size $|\mathrm{CSA}| + \mathcal{O}(n)$ bits that can find all occ occurrences of a pattern in $\mathcal{O}\left(t_{\mathsf{SA}}\left(\frac{m^k\sigma^k}{p} \max\,(k, \lg\lg n) + (1 + \frac{m}{p})\lg p \cdot \lg\lg n\right) + \mathrm{occ}\right)$ time, for $p \leq m^k\sigma^k$. Both results rely on the ability to merge two suffix array intervals quickly, a task for which we give a data structure of size $\mathcal{O}(n)$ bits on top of CSA that allows us to do the merging in $\mathcal{O}(t_{\mathsf{SA}} \lg\lg n)$ sequential (Lemma 4) or in $\mathcal{O}\big(t_{\mathsf{SA}}(1 + \lg_p \lg n)\big)$ parallel time (Lemma 7).

1.2 Related Work

We are only aware of one article addressing the parallelization of single queries [11]. Their main result is to augment a suffix tree with a data structure of size $\mathcal{O}(n \lg p)$ words that answers pattern matching queries using $\mathcal{O}\left(\frac{m}{p} \lg p\right)$ time and $\mathcal{O}(m \lg p)$ work, which is worse than ours in all three dimensions. Parallelizing approximate pattern matching has not been done earlier, to the best of our knowledge. Another natural approach for exploiting parallelism would be distributing the patterns to be matched onto the different processors and answer them in parallel; this is more of a load balancing problem and cannot be compared with our approach. Parallel construction of text indices is another road of research [4, 12], and could easily be combined with our approach. Finally, in the early 1990's, some work has been done on parallelizing online pattern matching algorithms [2, 3].

2 Preliminaries

Let $T = t_1 \ldots t_n$ be a *text* of length n consisting of characters contained in an integer *alphabet* Σ of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$. $T[i..j]$ represents the *substring* $t_i \ldots t_j$ for $1 \leq i \leq j \leq n$. We call $T[i..n]$ the i-th *suffix* of T and $T[1..i]$ the i-th *prefix* of T. We denote the length of the *longest common prefix* of the i-th and j-th suffix, i.e., $T[i..n]$ and $T[j..n]$, by $\mathsf{lcp}(i, j)$. The *suffix array* of a text T of length n is a permutation of $\{1, \ldots, n\}$ such that $T[\mathsf{SA}[i]..n]$ is lexicographically smaller than $T[\mathsf{SA}[i+1]..n]$ for all $i = 1, \ldots, n-1$. We denote the inverse of SA with SA^{-1}.

An *interval* $\mathcal{I} = [b..e]$ is the set of consecutive integers from b to e, for $b \leq e$. For an interval $\mathcal{I}$, we use the notations $\mathsf{b}(\mathcal{I})$ and $\mathsf{e}(\mathcal{I})$ to denote the beginning and end of $\mathcal{I}$; i.e., $\mathcal{I} = [\mathsf{b}(\mathcal{I})..\mathsf{e}(\mathcal{I})]$. We write $|\mathcal{I}|$ to denote the length of $\mathcal{I}$; i.e., $|\mathcal{I}| = \mathsf{e}(\mathcal{I}) - \mathsf{b}(\mathcal{I}) + 1$.

For a pattern $\alpha \in \Sigma^*$, let $\mathcal{I}(\alpha)$ be the interval with $T[\mathsf{SA}[i]..\mathsf{SA}[i] + |\alpha| - 1] = \alpha \iff i \in \mathcal{I}(\alpha)$. If we consider two intervals $\mathcal{I}(\alpha)$ and $\mathcal{I}(\beta)$ and the corresponding merged in-

Table 1 Different representations of the compressed suffix array using |CSA| bits with the time bound t_{SA} for accessing a value of SA and SA^{-1}. The sampling rate s satisfies $s = \omega(\lg_\sigma n)$.

$\lvert\text{CSA}\rvert$	t_{SA}	reference
$2n \lg n$	$\mathcal{O}(1)$	[14]
$(1+\epsilon)n \lg n$	$\mathcal{O}(\epsilon)$	[15]
$nH_k + o(nH_k) + \mathcal{O}\big(n + \sigma^{k+1} \lg n + n \lg n/s\big)$	$\mathcal{O}(\lg s)$	[1]

terval $\mathcal{I}(\alpha\beta)$, we call $\mathcal{I}(\alpha)$ the *left side* interval, $\mathcal{I}(\beta)$ the *right side* interval. Let $\Psi^k[i] = \mathsf{SA}^{-1}[\mathsf{SA}[i]+k]$ be the position of the suffix $T[\mathsf{SA}[i]+k..n]$ in the suffix array.

2.1 Suffix Trees

The *suffix tree* of a text T is the tree obtained by compacting the trie of all suffixes of T; it has n leaves and less than n internal nodes, where n is the length of T. Each edge is labeled with a string. We enumerate the leaves from left to right such that the i-th leaf has $i-1$ lexicographically preceding suffixes; we write $\mathsf{leafrank}(\ell) = i$ if the leaf ℓ is the i-th leaf. We extend the notion of intervals to nodes; i.e., $\mathcal{I}(v)$ denotes the interval $[b..e]$ such that $\mathsf{SA}[b], \ldots, \mathsf{SA}[e]$ are exactly the suffixes below node v.

Since we target small space bounds, our focus is on a compressed representation of suffix trees [19, 17, 6, 7]. The main ingredient of the so-called compressed suffix tree is a *compressed suffix array* [16]. Depending on its implementation, a compressed suffix array takes |CSA| bits of space, and gives t_{SA} time access to SA and SA^{-1} – see Table 1 for a comparison of the uncompressed and a compressed suffix array. With additional $\mathcal{O}(n)$ [19] or even $o(n)$ bits [5], a compressed suffix tree can answer queries on the LCP-array that stores the values $\mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[i+1])$ for each $1 \le i \le n-1$. The last ingredient of a compressed suffix tree is the tree topology (either explicitly [19] or implicitly [17]), and $o(n)$-bit succinct data structures for navigating in it [20, 9].

For our purpose, we need the following queries on the suffix tree: $\mathsf{lca}(u, v)$ returns the lowest common ancestor of two nodes u and v, $\mathsf{label}(e)$ returns the label of an edge e, $\mathsf{pathlabel}(v)$ returns the labels on the edges of the path from the root to v. These queries can be answered by all common compressed suffix trees [17, 19, 6, 7].

2.2 Integer Dictionaries

An *integer dictionary* is a set consisting of tuples of the form (k, v), where $k \in U := [1..|U|]$ is an integer from a universe U with $|U| = n^{\mathcal{O}(1)}$; we call k a *key* and v a *value*. A common task is to find a tuple in a dictionary by a given key. Besides, we might be interested in finding the *successor* (*predecessor*) of a key k, i.e., the largest (smallest) key k' in the dictionary with $k' \le k$ ($k' \ge k$). We define the operations $\mathsf{key}((k, v)) = k$ and $\mathsf{val}((k, v)) = v$.

A well-known *dynamic* integer dictionary representation is the *y-fast trie* [23]. It can perform lookups, predecessor and successor queries in $\mathcal{O}(\lg \lg n)$ expected time, and uses $\mathcal{O}(n \lg n)$ bits of space for storing n elements. It consists of an x-fast trie whose leaves store binary search trees. In more detail, the x-fast trie stores $\mathcal{O}(n/\lg n)$ entries in $\mathcal{O}(\lg n)$ hash tables, and each leaf stores $\mathcal{O}(\lg n)$ entries in its balanced binary search tree. Here, we only need a *static* version. Therefore, we use perfect hashing [8] as our hashing method, resulting in $\mathcal{O}(\lg \lg n)$ time w.h.p. in worst case for all queries, while keeping the same space bounds

and linear deterministic construction time. Alternatively, we can construct the hash tables in $\mathcal{O}(n \lg \lg n)$ deterministic time [18, Theorem 1], resulting in $\mathcal{O}(\lg \lg n)$ deterministic worst case time for all queries. Further, we exchange the balanced binary search trees with sorted arrays, which will be useful later when we parallelize the queries.

3 Suffix Array Interval Merging

To perform the merging of two suffix array intervals in $\mathcal{O}(t_{\mathsf{SA}} \lg \lg n)$ time, we adapt the idea from Lam et al. [13, Lemma 19]. In their method, the aim is to output all occurrences resulting from the merging of two suffix array intervals in $\mathcal{O}(t_{\mathsf{SA}}(\lg \lg n + \mathrm{occ}))$ time. Here, we show how to modify their approach such that only the resulting interval is returned, leading to $\mathcal{O}(t_{\mathsf{SA}} \lg \lg n)$ time. Although our method is similar to Lam et al. [13], we give the full proof for completeness.

The idea is to sample the Ψ- and lcp-values of each $(\lg^2 n)$-th suffix array position. The sampling is stored in y-fast tries such that a search in a sorted array can be broken down to a y-fast trie query, or to a binary search on a range of size $\mathcal{O}(\lg^2 n)$ – both can be performed in $\mathcal{O}(\lg \lg n)$ time. To lower the space consumption, the sampling is done only for certain nodes determined by the heavy path decomposition of the suffix tree, whose definition follows.

3.1 Heavy Path Decomposition

The *heavy path decomposition* of a rooted tree assigns a *level* to each node of the tree. The level of the root is 1. A node inherits the level of its parent if its subtree is the largest among the subtrees of all its siblings (ties are broken arbitrarily); we call such a node *heavy*. Otherwise, it has the level of its parent incremented by one; we then call the node *light*. Further, we define the root to be light. A maximal connected subgraph consisting of nodes on the same level is called *heavy path*. A heavy path starts with a light node, called *head*, and ends at a heavy leaf.

3.2 Precomputed Data Structures

We first present a simple data structure for the child-operation $\mathsf{child}(u, c)$ in a (compressed) suffix tree, i.e., for finding the child v of u such that the label of the edge between u and v starts with character c. We use $\Delta = \Omega(\lg n)$ as the sampling rate throughout this section.

▶ **Lemma 1.** *The suffix tree of a text of length n can be augmented with a data structure of size $\mathcal{O}(n \lg n/\Delta)$ bits answering $\mathsf{child}(v, c)$ in $\mathcal{O}(t_{SA} \lg \Delta)$ time.*

Proof. We sample the children of each internal node u and store the sampled children in a y-fast trie with the first character of the edge label between u and the respective child as key. Given a node u with k children, we sample every Δ-th child of u so that u's y-fast trie contains $\lfloor k/\Delta \rfloor$ elements. Since the suffix tree has less than $2n$ nodes, storing the y-fast tries for all internal node takes $\mathcal{O}(n \lg n/\Delta)$ bits overall.

Given a character c, we search $\mathsf{child}(u, c)$ in the following way: Since the children of a node u are sorted by the first character of the edge connecting u with its respective child, the y-fast trie of u can retrieve the first child v whose edge label $\mathsf{label}(u, v)$ is lexicographically at least as large as c. If c is a prefix of $\mathsf{label}(u, v)$, then we are done. Otherwise, say that v is the i-th child of u, we can find $\mathsf{child}(u, c)$ by a binary search on the range between the $(i - \Delta)$-th child and the i-th child in $\mathcal{O}(t_{\mathsf{SA}} \lg \Delta)$ time. ◀

We also need a simple $\mathcal{O}(n)$-bit data structure to find the heavy leaf of a given heavy path in constant time [13, Lemma 15].

Next, we define three types of integer dictionaries that we are going to index in y-fast tries to allow fast lookups. For every light node v, we define the integer dictionary

$$\Gamma(v) := \left\{ (\Psi^{|\mathsf{pathlabel}(v)|}[i], i) : i \equiv 1 \pmod{\Delta} \text{ and } i \in \mathcal{I}(v) \right\}.$$

Given a heavy leaf ℓ and its head v, we define the two integer dictionaries

$$H_{\mathrm{L}}(\ell) := \{(\mathsf{lcp}(\mathsf{SA}[\mathsf{leafrank}(\ell)], \mathsf{SA}[i]), i) : i \equiv 1 \pmod{\Delta} \text{ and } i \in \mathcal{I}(v) \text{ and } i \le \mathsf{leafrank}(\ell)\}$$

and

$$H_{\mathrm{R}}(\ell) := \{(\mathsf{lcp}(\mathsf{SA}[\mathsf{leafrank}(\ell)], \mathsf{SA}[i]), i) : i \equiv 1 \pmod{\Delta} \text{ and } i \in \mathcal{I}(v) \text{ and } i > \mathsf{leafrank}(\ell)\}.$$

We store $\Gamma(v)$ in a y-fast trie for each light node v, $H_{\mathrm{L}}(\ell)$ and $H_{\mathrm{R}}(\ell)$ in a y-fast trie for each heavy leaf ℓ. Given an interval $\mathcal{J} \subseteq [1..n]$, we can find

- an $i \in \Gamma(v)$ with $\mathsf{b}(\mathcal{J}) \le \mathsf{key}(i) = \Psi^{|\mathsf{pathlabel}(v)|}[\mathsf{val}(i)] \le \mathsf{e}(\mathcal{J})$,
- an $i_{\mathrm{l}} \in H_{\mathrm{L}}(\ell)$ with $\mathsf{b}(\mathcal{J}) \le \mathsf{key}(i_{\mathrm{l}}) = \mathsf{lcp}(\mathsf{SA}[\mathsf{leafrank}(\ell)], \mathsf{SA}[\mathsf{val}(i_{\mathrm{l}})]) \le \mathsf{e}(\mathcal{J})$, and
- an $i_{\mathrm{r}} \in H_{\mathrm{R}}(\ell)$ with $\mathsf{b}(\mathcal{J}) \le \mathsf{key}(i_{\mathrm{r}}) = \mathsf{lcp}(\mathsf{SA}[\mathsf{leafrank}(\ell)], \mathsf{SA}[\mathsf{val}(i_{\mathrm{r}})]) \le \mathsf{e}(\mathcal{J})$,

all in $\mathcal{O}(t_{\mathsf{SA}} \cdot \lg \Delta)$ time.

▶ **Lemma 2.** *We need $\mathcal{O}\big(n \lg^2 n/\Delta\big)$ bits of space to store the y-fast tries for all $\Gamma(\cdot)$, $H_{\mathrm{L}}(\cdot)$, and $H_{\mathrm{R}}(\cdot)$.*

Proof. Since the subtrees of the light nodes on the same level are disjoint, summing over the sizes of $\Gamma(v)$ for all light nodes v on the same level yields at most n/Δ elements. Since the heavy path decomposition has at most $\mathcal{O}(\lg n)$ different levels and a y-fast trie uses $\mathcal{O}(\lg n)$ bits per stored element, the claim for $|\Gamma(v)|$ follows.

We analyze the size of $H_{\mathrm{L}}(\cdot)$ by identifying a leaf with its $\mathsf{leafrank}$. The sampling of $H_{\mathrm{L}}(\cdot)$ considers only n/Δ leaves. A leaf ℓ has at most $\mathcal{O}(\lg n)$ light nodes as ancestors. So there are at most $\mathcal{O}(\lg n)$ heavy leaves ℓ_{H} having $\mathsf{leafrank}(\ell)$ as a value in their dictionary $H_{\mathrm{L}}(\ell_{\mathrm{H}})$. Hence, summing over $H_{\mathrm{L}}(\ell_{\mathrm{H}})$ for all heavy leaves ℓ_{H} yields $\mathcal{O}(n \lg n/\Delta)$ elements. The same considerations lead to the same size bounds for $H_{\mathrm{R}}(\cdot)$. ◀

▶ **Lemma 3.** *Given the compressed suffix tree of T and the dictionaries $\Gamma(\cdot)$, $H_{\mathrm{L}}(\cdot)$ and $H_{\mathrm{R}}(\cdot)$ as defined above, we can merge two suffix array intervals in $\mathcal{O}(t_{\mathsf{SA}} \lg \Delta)$ time.*

Proof. Let $\mathcal{I}(\alpha)$ and $\mathcal{I}(\beta)$ be two suffix array intervals and $P := \alpha\beta$. Our task is to search the interval $\mathcal{I}(P) \subseteq \mathcal{I}(\alpha)$ with $\Psi^{|\alpha|}[i] \in \mathcal{I}(\beta)$ for all $i \in \mathcal{I}(P)$. Since $i \mapsto \Psi^{|\alpha|}[i]$ is monotonically increasing for $i \in \mathcal{I}(\alpha)$, the merge could be solved with two binary searches in $\mathcal{I}(\alpha)$. To obtain the $\mathcal{O}(t_{\mathsf{SA}} \lg \Delta)$ time bound we will either use the y-fast tries, or perform a binary search on $\mathcal{O}(\Delta)$-large intervals.

Let us take the node v whose suffix array interval is $\mathcal{I}(\alpha)$, i.e., the lowest common ancestor of the leaves with $\mathsf{leafrank}\ \mathsf{b}(\mathcal{I}(\alpha))$ and $\mathsf{e}(\mathcal{I}(\alpha))$. We consider two cases:

Node v is heavy. Let H be the heavy path to which v belongs, ℓ its heavy leaf, and u its head.

If $\Gamma(u)$ is empty, there are less than Δ leaves in the subtree rooted at u. Since $\mathcal{I}(P) \subset \mathcal{I}(u)$, we can find $\mathcal{I}(P)$ by binary search in $\mathcal{O}(t_{\mathsf{SA}} \lg \Delta)$.

Otherwise ($\Gamma(u) \neq \emptyset$), let $q := \mathsf{lcp}\big(\mathsf{SA}[\Psi^{|\alpha|}[\mathsf{leafrank}(\ell)]], \mathsf{SA}[\mathsf{b}(\mathcal{I}(\beta))]\big)$. The value q is the length of the longest common prefix of P and the path label of ℓ, subtracted by $|\alpha|$. By definition of q, there is a node r on H whose path label coincides with $\alpha\beta[1..q]$. In particular, this is the node on the path H whose path label is the longest prefix of P with respect to the path labels of all other nodes on H. Since $\mathcal{I}(P) \subset \mathcal{I}(r)$, our task is to find r in $\mathcal{O}(t_{\mathsf{SA}} \lg \Delta)$ time. To this end, we locate a leaf whose LCA with ℓ is r.

The interval boundaries can be found by a coarse search on the y-fast tries of $H_{\mathrm{L}}(\ell)$ and $H_{\mathrm{R}}(\ell)$, and a subsequent refinement step using binary search. Let $k := \mathsf{leafrank}(\ell)$. Since $i \mapsto \mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[k])$ is monotonically increasing for $i < k$, and monotonically decreasing for $i > k$, we can perform the binary search for a value on the key-sorted integer dictionaries $\{(\mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[k]), i) : i < k\}$ and $\{(\mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[k]), i) : i > k\}$ conceptionally. The y-fast tries at $H_{\mathrm{L}}(\ell)$ and $H_{\mathrm{R}}(\ell)$ help us computing the tuple $j_{\mathrm{l}} \in H_{\mathrm{L}}(\ell) \cup \{(|\mathsf{pathlabel}(\ell)|, k)\}$ with

$$\mathsf{lcp}(\mathsf{SA}[\mathsf{val}(j_{\mathrm{l}}) - \Delta], \mathsf{SA}[k]) \leq |\alpha| + q \leq \mathsf{key}(j_{\mathrm{l}}) = \mathsf{lcp}(\mathsf{SA}[\mathsf{val}(j_{\mathrm{l}})], \mathsf{SA}[k])$$

and the tuple $j_{\mathrm{r}} \in H_{\mathrm{R}}(\ell) \cup \{(|\mathsf{pathlabel}(\ell)|, k)\}$ with

$$\mathsf{key}(j_{\mathrm{r}}) = \mathsf{lcp}(\mathsf{SA}[\mathsf{val}(j_{\mathrm{r}})], \mathsf{SA}[k]) \leq |\alpha| + q \leq \mathsf{lcp}(\mathsf{SA}[\mathsf{val}(j_{\mathrm{r}}) + \Delta], \mathsf{SA}[k]) .$$

Since $\mathsf{lcp}(\mathsf{SA}[\mathsf{val}(j_{\mathrm{l}}) - \Delta], \mathsf{SA}[k]) \leq |\alpha| + q \leq \mathsf{lcp}(\mathsf{SA}[\mathsf{val}(j_{\mathrm{r}}) + \Delta], \mathsf{SA}[k])$, we can find one of the positions $i_{\mathrm{l}} \in [\mathsf{val}(j_{\mathrm{l}}) - \Delta .. \mathsf{val}(j_{\mathrm{l}})]$ and $i_{\mathrm{r}} \in [\mathsf{val}(j_{\mathrm{r}}) .. \mathsf{val}(j_{\mathrm{r}}) + \Delta]$ by binary search such that $\mathsf{lcp}(\mathsf{SA}[i_{\mathrm{l}}], \mathsf{SA}[k]) = \mathsf{lcp}(\mathsf{SA}[i_{\mathrm{r}}], \mathsf{SA}[k]) = |\alpha| + q$. The binary search takes $\mathcal{O}(t_{\mathsf{SA}} \lg \Delta)$ time. On finding i_{l} or i_{r}, we can retrieve r, i.e., the lowest common ancestor of ℓ and the i_{l}-th or i_{r}-th leaf. If the pattern P is a prefix of the path label of r, then $\mathcal{I}(P) = \mathcal{I}(r)$, and we are done. Otherwise, we choose the child w of r whose edge label S starts with $\beta[q+1]$; w can be retrieved in $\mathcal{O}(t_{\mathsf{SA}} \lg \Delta)$ time by Lemma 1. The child w must be a light node, for otherwise we get a contradiction to the definition of r. We set $v \leftarrow w$, $\alpha \leftarrow P[1.. |\alpha| + q + |S|]$, $\beta \leftarrow P[|\alpha| + 1.. |P|]$, and jump to the next case:

Node v is light. If $\Gamma(v)$ is empty, then $|\mathcal{I}(v)| < \Delta$. Therefore, we can find the interval boundaries of $\mathcal{I}(P)$ in $\mathcal{I}(v)$ with a binary search in $\mathcal{O}(t_{\mathsf{SA}} \lg \Delta)$ time. Otherwise, we use the y-fast trie storing $\Gamma(v)$ to find the tuple $j_{\mathrm{l}} \in \Gamma(v)$ with the smallest key satisfying $\mathsf{b}(\mathcal{I}(\beta)) \leq \mathsf{key}(j_{\mathrm{l}}) = \Psi^{|\alpha|}[\mathsf{val}(j_{\mathrm{l}})]$ and the tuple $j_{\mathrm{r}} \in \Gamma(v)$ with the largest key satisfying $\mathsf{key}(j_{\mathrm{r}}) = \Psi^{|\alpha|}[\mathsf{val}(j_{\mathrm{r}})] \leq \mathsf{e}(\mathcal{I}(\beta))$. If both exist, we can find the positions $\mathsf{b}(\mathcal{I}(P)) \in [\mathsf{val}(j_{\mathrm{l}}) - \Delta .. \mathsf{val}(j_{\mathrm{l}})]$ and $\mathsf{e}(\mathcal{I}(P)) \in [\mathsf{val}(j_{\mathrm{r}}) .. \mathsf{val}(j_{\mathrm{r}}) + \Delta]$ by two binary searches. If there is no tuple $i \in \Gamma(v)$ with $\mathsf{b}(\mathcal{I}(\beta)) \leq \mathsf{key}(i) \leq \mathsf{e}(\mathcal{I}(\beta))$, we search with the y-fast trie of $\Gamma(v)$ the tuple $k_{\mathrm{l}} \in \Gamma(v)$ with

$$\mathsf{key}(k_{\mathrm{l}}) = \Psi^{|\alpha|}[\mathsf{val}(k_{\mathrm{l}})] \leq \mathsf{b}(\mathcal{I}(\beta)) \leq \Psi^{|\alpha|}[\mathsf{val}(k_{\mathrm{l}}) + \Delta]$$

and the tuple $k_{\mathrm{r}} \in \Gamma(v)$ with

$$\Psi^{|\alpha|}[\mathsf{val}(k_{\mathrm{r}}) - \Delta] \leq \mathsf{e}(\mathcal{I}(\beta)) \leq \Psi^{|\alpha|}[\mathsf{val}(k_{\mathrm{r}})] = \mathsf{key}(k_{\mathrm{r}}) .$$

Both values exist, and $\mathsf{val}(k_{\mathrm{r}}) - \mathsf{val}(k_{\mathrm{l}}) \leq \Delta$. So we find the interval $\mathcal{I}(P)$ by applying two binary searches to the range $\mathsf{val}(k_{\mathrm{l}}) .. \mathsf{val}(k_{\mathrm{r}})$.

◀

Setting $\Delta := \lg^c n$ for $c \geq 2$ yields:

▶ **Lemma 4.** *Given the compressed suffix tree of T, there is a data structure of size $\mathcal{O}(n)$ bits that allows us to merge two suffix array intervals in $\mathcal{O}(t_{\mathsf{SA}} \lg \lg n)$ time.*

4 Parallel Exact Pattern Matching

We parallelize the merging of suffix array intervals that we presented in Section 3 and show that queries in the suffix tree using consecutive subpatterns and linear space can be solved in parallel on a CREW-PRAM. For this, we use parallel binary search:

▶ **Lemma 5** ([21, Theorem 2.1])**.** *Given a sorted array of size n, a binary search requires $\mathcal{O}\big(1 + \lg_p n\big)$ time when operating on a CREW-PRAM with p processors.*

We conclude that we can parallelize the query on y-fast tries in the same way:

▶ **Lemma 6.** *A y-fast trie can do lookups, predecessor and successor queries in $\mathcal{O}\big(1 + \lg_p \lg n\big)$ time using p processors.*

Proof. We can find an element in an x-fast trie in $\mathcal{O}\big(1+\lg_p \lg n\big)$ time using parallel binary search (Lemma 5) on the $\mathcal{O}(\lg n)$ hash tables. The sorted arrays stored at the leaves can similarly be searched in $\mathcal{O}\big(1+\lg_p \lg n\big)$ time, again using Lemma 5. ◀

Let us focus on the merging of two suffix array intervals as treated in Section 3. The dominant term of its running time is due to the query time of the y-fast tries and the binary searches. As we can parallelize both, a parallelization of the merging algorithm improves the time bounds significantly:

▶ **Lemma 7.** *Given p processors and two suffix array intervals $\mathcal{I}(\alpha)$ and $\mathcal{I}(\beta)$, the merged interval $\mathcal{I}(\alpha\beta)$ can be computed in $\mathcal{O}\big(t_{\mathsf{SA}}(1+\lg_p \lg n)\big)$ time and $\mathcal{O}\big(t_{\mathsf{SA}} \min(p, \lg n)\,(1+\lg_p \lg n)\big)$ work.*

Proof. We can merge two suffix array intervals in $\mathcal{O}(t_{\mathsf{SA}} \lg \lg n)$ time using Lemma 4. Recalling the proof of Lemma 3, we took the node v whose suffix array interval is $\mathcal{I}(\alpha)$. There, in both cases (v is either heavy or light), the time is dominated by searching in y-fast tries, and/or by binary searching in $\mathcal{O}(\Delta)$ sampled Ψ- or lcp-values. Both can be parallelized by Lemmas 5 and 6, respectively. This yields $\mathcal{O}\big(t_{\mathsf{SA}}(1+\lg_p \lg n)\big)$ time using p processors. During the parallel searches, we use at most $\mathcal{O}(\lg n)$ processors $\mathcal{O}\big(1+\lg_p \lg n\big)$ times. This amounts to $\mathcal{O}\big(t_{\mathsf{SA}} \min(p, \lg n)\,(1+\lg_p \lg n)\big)$ work. ◀

Being able to merge two suffix array intervals in parallel, we now show how to compute the suffix array interval of a pattern P in parallel. To this end, we decompose the pattern in subpatterns $\alpha_1, \ldots, \alpha_p$ such that $P = \alpha_1\alpha_2 \ldots \alpha_p$, and then compute the suffix array intervals for the subpatterns. Then we merge those intervals in parallel.

▶ **Theorem 8.** *Given a text of size n and a pattern of size m. With $p \leq m$ processors, we can compute the suffix array interval of the pattern in $\mathcal{O}\big(t_{\mathsf{SA}}\big(\frac{m}{p} + \lg p + \lg\lg p \cdot \lg\lg n\big)\big)$ time and $\mathcal{O}(t_{\mathsf{SA}}\,(m + \min(p, \lg n)\,(\lg p + \lg\lg p \cdot \lg\lg n)))$ work. In order to achieve this time bound we need an index of size $|\mathrm{CSA}| + \mathcal{O}(n)$ bits.*

Proof. Let $P = \alpha_1^0\alpha_2^0 \ldots \alpha_p^0$ be a pattern of length m such that $|\alpha_i^0| = \Theta\big(\frac{m}{p}\big)$ for $i = 1, \ldots, p$. The computation of all intervals $\mathcal{I}\big(\alpha_i^0\big)$ requires $\mathcal{O}\big(t_{\mathsf{SA}}\frac{m}{p}\big)$ time. In the first merge step we have two processors to compute each of the intervals $\mathcal{I}\big(\alpha_i^1\big) := \mathcal{I}\big(\alpha_{2i-1}^0\alpha_{2i}^0\big)$ for $i = 1, \ldots, \frac{p}{2}$. In each merge step we halve the number of intervals. So in the k-th merge step ($1 \leq k \leq \lg p$), we have 2^k processors to compute each of the intervals $\mathcal{I}\big(\alpha_i^k\big) := \mathcal{I}\big(\alpha_{2i-1}^{k-1}\alpha_{2i}^{k-1}\big)$ for $i = 1, \ldots, \frac{p}{2^k}$. As we require $\mathcal{O}(\lg p)$ merge steps and can use Lemma 7 with 2^k processors in the k-th merge step, the interval $\mathcal{I}(P)$ can be computed in $\mathcal{O}\Big(t_{\mathsf{SA}} \sum_{k=1}^{\lg p} (1+\lg_{2^k} \lg n)\Big) = \mathcal{O}(t_{\mathsf{SA}}\,(\lg p + \lg\lg p \cdot \lg\lg n))$ time, given the intervals $\mathcal{I}\big(\alpha_i^0\big)$ of the subpatterns – see Figure 1. In total, $\mathcal{I}(P)$ can be found in $\mathcal{O}\big(t_{\mathsf{SA}}\big(\frac{m}{p} + \lg p + \lg\lg p \cdot \lg\lg n\big)\big)$ time.

During the computation of the suffix array intervals of the subpatterns of P we use all p processors, which results in $\mathcal{O}(t_{\mathsf{SA}} m)$ work. The same holds for each merging step, as we use all processors to parallelize the binary search. We have $\mathcal{O}(\lg p)$ merge steps. During the k-th merge step, we merge $\frac{p}{2^k}$ suffix array intervals with 2^k processors each. Using Lemma 7 the total work is $\mathcal{O}(t_{\mathsf{SA}}\,(m + \min(p, \lg n)\,(\lg p + \lg\lg p \cdot \lg\lg n)))$. ◀

5 Parallel Approximate Pattern Matching

In this section, we consider two different distances for the approximate string matching problem. The first distance we consider is the *Levenshtein distance*, where the distance between two patterns P and P' is the minimal number of the operations *insert*, *change* and *remove* required to change P' into P. The second one is the *Hamming distance*, where the distance of two pattern P and P' of equal length is the number of mismatching positions, i.e., $|\{i\colon P[i] \neq P'[i], 1 \leq i \leq |P|\}|$. We consider two problems related to these distances.

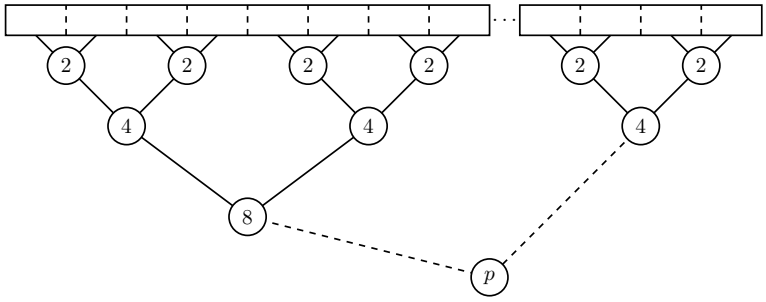


Figure 1 Schedule for the merging of p suffix array intervals, i.e., the suffix array intervals of the subpatterns. The number in each node denotes the number of processors available for the merging of the two considered suffix array intervals.

k-difference problem Given a text T of length n and a pattern P of length m, we want to report all occurrences $i \in \{1, \ldots, n\}$ such that $T[i..i+j]$ and P have a Levenshtein distance of at most k for at least one $j \in \{0, \ldots, n-i\}$.

k-mismatch problem Given a text T of length n and a pattern P of length m, we want to report all occurrences $i \in \{1, \ldots, n-m\}$ such that $T[i..i+m]$ and P have a Hamming distance of at most k.

We apply the results from Section 3 to parallelize the approximate string matching algorithm by Huynh et al. [10]. To do so, we first present an approach to compute the suffix array intervals of all prefixes and suffixes of the pattern in parallel – see Figure 2.

▶ **Lemma 9.** *Given a text of length n and its suffix array, we can compute the suffix array intervals of all prefixes and suffixes of a pattern of length m in $\mathcal{O}\big(t_{SA}(1+\frac{m}{p})\lg p \cdot \lg\lg n\big)$ parallel time operating on a CREW-PRAM with p processors.*

Proof. Let $P = \alpha_0\alpha_1 \ldots \alpha_{p-1}$ be a pattern of length m such that $|\alpha_i| = \frac{m}{p}$ for $i = 0, \ldots, p-1$. Thus, the j-th prefix of a subpattern α_i is $P[1+i\frac{m}{p}..i\frac{m}{p}+j]$ for all $i = 0, \ldots, p-1$ and $j = 1, \ldots, \frac{m}{p}$. First, we compute the suffix array intervals for all those prefixes in parallel, which requires $\mathcal{O}(1+m/p)$ time, as no merging is necessary during this step.

In the second step, we merge the suffix array intervals in parallel. Since we want the suffix array intervals of all prefixes of the pattern, during the first merge step we merge the suffix array intervals $\mathcal{I}\big(P[1+2i\frac{m}{p}..(2i+1)\frac{m}{p}]\big)$ as the left side interval with each of the intervals $\mathcal{I}\big(P[1+(2i+1)\frac{m}{p}..(2i+1)\frac{m}{p}+j\big)$ as right side interval for all $i = 0, \ldots, \frac{p}{2}-1$ and $j = 1, \ldots, \frac{m}{p}$. This results in the intervals $\mathcal{I}\big(P[1+2i\frac{m}{p}..(2i+1)\frac{m}{p}+j\big)$ for $i = 0, \ldots, \frac{p}{2}-1$ and $j = 1, \ldots, \frac{m}{p}$. During each merge step, we halve the number of left side intervals that we have to consider during the next merge step but double the number of right side intervals that are merged, i.e., in the k-th merge step, we merge the intervals $\mathcal{I}\big(P[1+2^k i\frac{m}{p}..(2^k i+2^{k-1})\frac{m}{p}]\big)$ with each of the intervals $\mathcal{I}\big(P[1+(2^k i+2^{k-1})\frac{m}{p}..(2^k i+2^{k-1})\frac{m}{p}+j]\big)$ for $i = 0, \ldots, \frac{p}{2^k}-1$ and $j = 1, \ldots, 2^{k-1}\frac{m}{p}$. This amounts to $\mathcal{O}(m)$ intervals that need to be merged in each step. In the end, we obtain the suffix array intervals of the prefixes of P, i.e., the intervals $\mathcal{I}(P[1..j])$ for $j = 1, \ldots, m$. Since we start with p left side intervals, and each merge step halves the number of left side intervals, we end up with $\lg p$ merge steps.

The computation of the suffix array intervals of all suffixes of P works analogously. Using Lemma 4, we can compute the suffix array intervals of all prefixes and suffixes of the pattern in $\mathcal{O}\big(t_{\mathsf{SA}}(1+\frac{m}{p})\lg p \cdot \lg\lg n\big)$ time. ◀

▶ **Theorem 10.** *With $|\mathrm{CSA}| + \mathcal{O}(n)$ bits of space, the 1-difference and 1-mismatch problems can be solved in parallel in $\mathcal{O}\big(t_{SA}\lg\lg n\,\big(\frac{m\sigma}{p} + (1+\frac{m}{p})\lg p\big) + occ\big)$ for $p \le m\sigma$.*

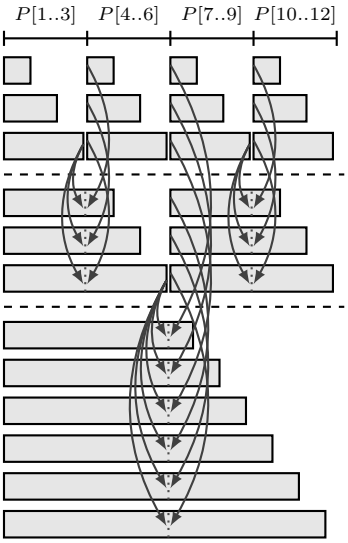


Figure 2 Schematics of the merging process to compute the suffix array intervals of all prefixes of a pattern $P = a_1 \dots a_4$ of length $m = 12$ using $p = 4$ processors. The gray blocks above the first dashed line represent the suffix array intervals of all prefixes of the subpatterns a_i (for $i = 1, \dots, 4$). The blocks between the dashed lines represent the suffix array intervals after the first merge step. The intervals that are merged are shown by arrows. The blocks below the second dashed line are the suffix array intervals computed in the second merge step.

Table 2 Let P' be the resulting string of introducing an error in the pattern $P[1..m]$ at position i. Further, let v be the suffix tree node with $\mathcal{I}(v) = \mathcal{I}(P[1..i-1])$. We can compute the two suffix array intervals considered for merging in $\mathcal{O}(t_{\mathsf{SA}} \lg \lg n)$ time, and perform the merging in the same time.

operation	c	P'	intervals to merge
substitution	$c \in \Sigma \setminus \{P[i]\}$	$P[1..i-1]cP[i+1..m]$	$\mathcal{I}(\mathsf{child}(v,c))$ and $\mathcal{I}(P[i+1..m])$
deletion	–	$P[1..i-1]P[i+1..m]$	$\mathcal{I}(v)$ and $\mathcal{I}(P[i+1..m])$
insertion	$c \in \Sigma$	$P[1..i-1]cP[i..m]$	$\mathcal{I}(\mathsf{child}(v,c))$ and $\mathcal{I}(P[i..m])$

Proof. We precompute the suffix array intervals $\mathcal{I}(P[i..m])$ and $\mathcal{I}(P[1..i])$ for all $1 \le i \le m$ in parallel by Lemma 9. This requires $\mathcal{O}\big(t_{\mathsf{SA}}(1 + \frac{m}{p}) \lg p \cdot \lg \lg n\big)$ time. The exact matches are found in the interval $\mathcal{I}(P[1..m])$. To compute the matches with one error, we iterate over all positions in $P[1..m]$, and introduce an error at one position i with $1 \le i \le m$. An error can be introduced by an insertion, a deletion, or a substitution. Let us fix one modification occurring at position i, and call the modified string P'. Our task is to find $\mathcal{I}(P')$. To this end, we exploit some already computed results, i.e., we have $\mathcal{I}(P'[1..i-1]) = \mathcal{I}(P[1..i-1])$, and either (substitution) $\mathcal{I}(P'[i+1..m]) = \mathcal{I}(P[i+1..m])$, (deletion) $\mathcal{I}(P'[i..m-1]) = \mathcal{I}(P[i+1..m])$, or (insertion) $\mathcal{I}(P'[i+1..m+1]) = \mathcal{I}(P[i..m])$ – see Table 2. If P' resulted from an insertion or substitution, the interval $\mathcal{I}(P'[1..i-1])$ can be enhanced to $\mathcal{I}(P'[1..i])$ by $\mathsf{child}(v, P'[i])$ in $\mathcal{O}(t_{\mathsf{SA}} \lg \lg n)$ time due to Lemma 1, where v is the node with $\mathcal{I}(v) = \mathcal{I}(P'[1..i-1])$. Finally, we can compute $\mathcal{I}(P')$ by merging two intervals in $\mathcal{O}(t_{\mathsf{SA}} \lg \lg n)$ time with Lemma 4. Introducing an error in P at m different positions with σ different characters is embarrassingly parallel. With $p \le m\sigma$ processors it requires $\mathcal{O}\big(t_{\mathsf{SA}} \frac{m\sigma}{p} \lg \lg n + \mathrm{occ}\big)$ time in addition to the time for the preprocessing. ◀

Up to now, we have assumed that the time for the output is in $\mathcal{O}(\mathrm{occ})$. Unfortunately, this is not always the case, as an occurrence of a pattern with k errors may be reported multiple times. For

example, if we allow one error, the pattern `aba` could be reported twice at the first position of the text `aaa`, as the second position of the pattern could either be deleted or replaced. Hence, we need to make sure that each occurrence of a pattern is reported just once, regardless how many different combinations of operations can be used to change the pattern to the corresponding substring. This problem has been discussed and solved in [10].

▶ **Lemma 11** ([10, Discussion related to Theorem 2]). *Given a pattern P, we can check whether an occurrence of the pattern P' with at most k errors is minimal regarding its distance and its edit operations to P in $\mathcal{O}(k)$ time whenever we append a character or want to report an occurrence.*

Using Lemmas 9 and 11, we can solve the 1-difference and 1-mismatch problems in parallel as described above. The same is true for the k-difference and k-mismatch problems.

▶ **Theorem 12.** *Using $|\mathrm{CSA}| + \mathcal{O}(n)$ bits of space, the k-difference and k-mismatch problems can be solved in parallel in $\mathcal{O}\Big(t_{SA}\left(\frac{m^k\sigma^k}{p}\max(k, \lg\lg n) + (1+\frac{m}{p})\lg p \cdot \lg\lg n\right) + occ\Big)$ for $p \leq m^k\sigma^k$ processors.*

Proof. The idea of the algorithm is similar to the algorithm of Theorem 10. First, we compute the suffix array intervals of all the suffixes and prefixes of the pattern using Lemma 9. This requires $\mathcal{O}\big(t_{\mathsf{SA}}(1+\frac{m}{p})\lg p \cdot \lg\lg n\big)$ time. We want to introduce at most k errors in parallel. Again, we parallelize over the positions of the introduced errors. Similar to the idea of Theorem 10, we merge different suffix array intervals. But in this case, we cannot parallelize over one position, instead we have to parallelize considering up to k positions where we can include an error.

The number of patterns P' that have a distance of at most k from P is bounded by $\mathcal{O}\big(\sigma^k m^k\big)$ [22, Theorem 6]. Thus, we require $\mathcal{O}\Big(t_{\mathsf{SA}}\frac{m^k\sigma^k}{p}\max(k, \lg\lg n) + \mathrm{occ}\Big)$ time using $p \leq \sigma^k m^k$ processors in parallel. The $\mathcal{O}(\max(k, \lg\lg n))$-term results from the check of whether the occurrence is computed with minimal distance to the pattern P, which has to be done every time we update the considered pattern and requires $\mathcal{O}(k)$ time using Lemma 11. ◀

References

1 D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms (TALG)*, 10(4):article 23, 2014.

2 Dany Breslauer and Zvi Galil. An optimal O(log log n) time parallel string matching algorithm. *SIAM J. Comput.*, 19(6):1051–1058, 1990.

3 Dany Breslauer and Zvi Galil. A lower bound for parallel string matching. *SIAM J. Comput.*, 21(5):856–862, 1992.

4 Martin Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *Proc. ICALP*, volume 1099 of *LNCS*, pages 550–561. Springer, 1996.

5 Johannes Fischer. Wee LCP. *Inform. Process. Lett.*, 110(8–9):317–320, 2010.

6 Johannes Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011.

7 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

8 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

9 Simon Gog and Johannes Fischer. Advantages of shared data structures for sequences of balanced parentheses. In *Proc. DCC*, pages 406–415. IEEE Press, 2010.

10 Trinh ND Huynh, Wing-Kai Hon, Tak-Wah Lam, and Wing-Kin Sung. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science*, 352(1):240–249, 2006.

11 Matevz Jekovec and Andrej Brodnik. Parallel query in the suffix tree. *CoRR*, abs/1509.06167, 2015. URL: `http://arxiv.org/abs/1509.06167`.

12 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In *Proc. CPM*, volume 9133 of *LNCS*, pages 329–342. Springer, 2015.

13 Tak-Wah Lam, Wing-Kin Sung, and Swee-Seong Wong. Improved Approximate String Matching Using Compressed Suffix Data Structures. *Algorithmica*, 51(3):298–314, 2007. `doi:10.1007/s00453-007-9104-8`.

14 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

15 J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012.

16 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):Article No. 2, 2007.

17 Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 322–333. Springer, 2010.

18 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Proc. ICALP (1)*, volume 5125 of *LNCS*, pages 84–95. Springer, 2008.

19 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst*, 41(4):589–607, 2007.

20 Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proc. SODA*, pages 134–149. ACM/SIAM, 2010.

21 Marc Snir. On parallel searching. *SIAM J. Comput.*, 14(3):688–708, 1985. `doi:10.1137/0214051`.

22 Esko Ukkonen. Approximate string-matching over suffix trees. In *Proc. CPM*, volume 684 of *LNCS*, pages 228–242. Springer, 1993.

23 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Inform. Process. Lett.*, 17(2):81–84, 1983.

Factorizing a String into Squares in Linear Time

Yoshiaki Matsuoka[1], Shunsuke Inenaga[2], Hideo Bannai[3], Masayuki Takeda[4], and Florin Manea[5]

1 **Department of Informatics, Kyushu University, Japan**
`yoshiaki.matsuoka@inf.kyushu-u.ac.jp`
2 **Department of Informatics, Kyushu University, Japan**
`inenaga@inf.kyushu-u.ac.jp`
3 **Department of Informatics, Kyushu University, Japan**
`bannai@inf.kyushu-u.ac.jp`
4 **Department of Informatics, Kyushu University, Japa**
`takeda@inf.kyushu-u.ac.jp`
5 **Department of Computer Science, Kiel University, Germany**
`flm@informatik.uni-kiel.de`

Abstract

A *square factorization* of a string w is a factorization of w in which each factor is a square. Dumitran et al. [SPIRE 2015, pp. 54-66] showed how to find a square factorization of a given string of length n in $O(n \log n)$ time, and they posed a question whether it can be done in $O(n)$ time. In this paper, we answer their question positively, showing an $O(n)$-time algorithm for square factorization in the standard word RAM model with machine word size $\omega = \Omega(\log n)$. We also show an $O(n + (n \log^2 n)/\omega)$-time (respectively, $O(n \log n)$-time) algorithm to find a square factorization which contains the maximum (respectively, minimum) number of squares.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Squares, Runs, Factorization of Strings

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.27

1 Introduction

Factorization problems are one of the important topics in the study of string algorithms and combinatorics on strings and their applications. Essentially, the task is to efficiently identify a decomposition of a string into factors of a specific given form. For instance, we recall here the various forms of Lempel-Ziv factorizations of a string [21, 22, 19, 20]; this class of factorizations found many applications in data-compression but also in the efficient detection of repetitive structures in strings [18, 14]. Similarly, the standard factorization of strings (also called Lyndon factorization) [17, 8] found applications in data compression, in variants of the Burrows-Wheeler transform [16]. Both these factorizations were defined in very simple ways, starting from basic combinatorial concepts: repeats (or repeated occurrences of the same factor) in a string, or lexicographically minimal factors of a string; they can be both computed in linear time; see [5] and [8], respectively.

Some other factorizations of strings, whose factors are defined by well-studied combinatorial objects, were proposed and analyzed as well in the literature. Closer to the topic of this paper, we recall here palindromic factorizations of a string (where we want to split that string into an arbitrary number of non-trivial palindromic factors, or into a minimal or fixed number of such factors), analyzed already in the seminal paper of Knuth, Morris, and Pratt [13], as well as in a series of more recent papers in [9, 15, 2, 12]. In [1], Bakobeh et al. consider

algorithms for computing closed substrings in strings; a string is closed if it contains proper substring that occurs in it as a prefix and a suffix, but not elsewhere; more precisely, the problem of greedily factorizing a string into a sequence of longest closed substrings is solved. Even more relevant to our study, in [7] it was shown how factorizations into highly repetitive factors (e.g., repetitions with an exponent greater than 2) can be efficiently computed.

On the other hand, the study of repetitive structures occurring in strings is also one of the central topics in combinatorics on strings and stringology. Main and Lorentz [18] proposed an algorithm that decides whether a string w of length n contains a square (i.e., two consecutive occurrences of some factor, called root) in $O(n \log n)$ time. Their result was improved by Crochemore [4], who showed how to identify all the squares with a primitive root of a string w in $O(n \log n)$ time. These results hold for general alphabets and are optimal in a comparison-based model, but if we use the (realistic) RAM model with logarithmic word size (see [11] and the references therein for a survey of relevant results and techniques related to this model), and we are only interested in inputs over integer alphabets, we can actually find all runs of a string in linear time [14, 3]. Thus, we can also construct a succinct representation of all primitively rooted squares of a string within the same time complexity.

Following these two research directions, in [7], the task of deciding in linear time whether a string can be split into squares was left as an open problem. In this paper we show that a square factorization of a string can be indeed computed in linear time, using the RAM model with logarithmic word size and working under the assumption that the string is over an integer alphabet. We extend this result by proposing an efficient algorithm for the computation of such square factorizations with a maximum or minimum number of factors. Finally, we discuss several connected problems.

2 Preliminaries

Let Σ be an alphabet. An element of Σ^* is called a *string*. The empty string ε is the string of length 0. Let Σ^+ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For any strings x and y, we denote by $x \cdot y$ the concatenation of x and y. For a string $w = x \cdot y \cdot z$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. A prefix x of w is called a *proper prefix* of w if $x \neq w$. The length of a string w is denoted by $|w|$. The i-th character of a string w is denoted by $w[i]$ for each $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$. For any integers i and j with $i \leq j$, we denote $[i, j] = \{i, i+1, \ldots, j\}$.

For a string x and positive integer k, let $x^0 = \varepsilon$, and $x^k = x^{k-1} \cdot x$. A string w is called *primitive* if there does not exist a string x and an integer $k \geq 2$ such that $w = x^k$. For any non-empty string x, a repetition x^2 is called a *square*. A square x^2 is called a *primitively rooted square* if x is primitive.

A positive integer p is called a *period* of string w if $w[i] = w[i+p]$ for all $1 \leq i \leq |w| - p$. For a string w, a triplet (p, s, e) is called a *run* in w if $p \leq (e - s + 1)/2$, p is the smallest period of $w[s..e]$, $s = 1$ or $w[s-1] \neq w[s-1+p]$, and $e = |w|$ or $w[e+1] \neq w[e+1-p]$.

We call a sequence $F = (f_1, ..., f_m)$ of m non-empty strings a *square factorization* of a string w if f_i is a square for each $1 \leq i \leq m$ and the concatenation $f_1 \cdots f_m$ is equal to w; the integer m is called the size of the factorization. Also, we call F a *largest square factorization* (resp. a *smallest square factorization*) of w if the size m is largest (resp. smallest) among all square factorizations of w.

▶ **Example 1.** $F_1 = (\mathtt{abaababaab}, \mathtt{bb}, \mathtt{aa}, \mathtt{bb}, \mathtt{bb})$, $F_2 = (\mathtt{abaababaab}, \mathtt{bb}, \mathtt{aa}, \mathtt{bbbb})$ and $F_3 =$

(abaaba, baabbbaabb, bb) are the square factorizations of a string $w =$ abaababaabbbaabbbb. F_1 is the largest square factorization of w and F_3 is the smallest one.

Notice that a string can have more than one largest square factorization and/or one smallest square factorization. For instance, string $w =$ aabaabaa has two largest square factorizations: (aa, baabaa) and (aabaab, aa). Notice that they are also smallest square factorizations of the string.

Let w be a string of length n over an alphabet $\Sigma = [1, n]$. Our model of computation is a standard word RAM model of machine word size $\omega = \Omega(\log n)$, where the following operations can be performed in $O(1)$ time: Let X, Y be bit arrays of length $m \leq \omega$ each, and let k be a non-negative integer. We denote by $X \,\&\, Y$, $X \mid Y$, and $X \oplus Y$, the *bitwise and*, *bitwise or*, and *bitwise exclusive or* of X and Y, respectively. We denote by $\sim X$ the *bitwise negation* of X. We denote by $X \gg k$ the *k-bit logical right shift* of X. We can also see X as an unsigned m-bit integer where the most (least) significant bit is $X[1]$ (respectively, $X[m]$); arithmetic operations on such integers take constant time.

3 Algorithms

3.1 A linear time algorithm for computing a square factorization

In this subsection we propose an $O(n)$-time algorithm for computing a square factorization of a given string w of length n. Note that if a square factorization of w exists, then there clearly exists a square factorization such that each factor is a primitively rooted square. Therefore we only consider primitively rooted squares in w.

For any run $\lambda = (p, s, e)$ in w, we denote $\rho(\lambda) = p$, $SqBegRange(\lambda) = [s, e - 2p + 1]$ and $SqEndRange(\lambda) = [s+2p, e+1]$; namely, for any position $k \in [1, |w|+1]$, $k \in SqBegRange(\lambda)$ iff $w[k..k+2\rho(\lambda)-1]$ is a primitively rooted square, and $k \in SqEndRange(\lambda)$ iff $w[k-2\rho(\lambda)..k-1]$ is a primitively rooted square. Also, we denote by R all runs in w.

▶ **Lemma 2** ([3, 14]). *$|R| < n$. Also, R can be computed in $O(n)$ time.*

▶ **Lemma 3** ([6]). *For any string v, the number of prefixes of v which are also primitively rooted squares is $O(\log |v|)$.*

▶ **Corollary 4.** $\sum_{\lambda \in R} |SqBegRange(\lambda)| = \sum_{\lambda \in R} |SqEndRange(\lambda)| = O(n \log n)$.

Proof. Clearly, both $\sum_{\lambda \in R} |SqBegRange(\lambda)|$ and $\sum_{\lambda \in R} |SqEndRange(\lambda)|$ are equal to the number of primitively rooted squares in w. By Lemma 3, the number of primitively rooted squares beginning at each position of w is $O(\log n)$. Thus we obtain $\sum_{\lambda \in R} |SqBegRange(\lambda)| = \sum_{\lambda \in R} |SqEndRange(\lambda)| = O(n \log n)$. ◀

Let C be a bit array of length $n + 1$ such that $C[i] = 1$ iff $w[i..n]$ can be factorized into squares. For convenience, let $C[i] = 0$ if $i < 1$ or $i > n + 1$. Algorithm 1 is a simple solution by dynamic programming, which is essentially equivalent to the approach of [7].

Let τ be some integer parameter such that $1 \leq \tau \leq \omega$. We split C into blocks of length τ. For each $1 \leq j \leq \lceil (n + 1)/\tau \rceil$, we call $C[(j - 1)\tau + 1..j\tau]$ as the j-th block of C.

Let $SPR_\tau = \{\lambda \in R \mid 2\rho(\lambda) < \tau\}$ and $LPR_\tau = \{\lambda \in R \mid 2\rho(\lambda) \geq \tau\}$. We call each element of SPR_τ and LPR_τ a *short period run* and a *long period run*, respectively. Also, for each position i, we denote $S_{\tau,i} = \{\lambda_{\mathrm{SP}} \in SPR_\tau \mid i \in SqEndRange(\lambda_{\mathrm{SP}})\}$.

For each $1 \leq j \leq \lceil (n + 1)/\tau \rceil$, let $B_{\tau,j} = \{\lambda_{\mathrm{LP}} \in LPR_\tau \mid [(j-1)\tau+1, j\tau] \cap SqBegRange(\lambda_{\mathrm{LP}}) \neq \emptyset\}$ and $E_{\tau,j} = \{\lambda_{\mathrm{LP}} \in LPR_\tau \mid [(j - 1)\tau + 1, j\tau] \cap SqEndRange(\lambda_{\mathrm{LP}}) \neq \emptyset\}$. Also, for each

Algorithm 1: A simple algorithm for determining whether w can be factorized into squares.

```
Input: String w of length n.
Compute R;
C[1..n] ← 0;
C[n + 1] ← 1;
for i = n + 1 down to 1 do
    if C[i] = 1 then
        foreach λ ∈ R such that i ∈ SqEndRange(λ) do
            C[i − 2ρ(λ)] ← 1;
```

$i \in [1, n+1]$, let $P_{\tau,i}$ be a bit array of length $\tau - 1$ such that for each $k \in [1, \tau - 1]$, $P_{\tau,i}[k] = 1$ iff there exists $\lambda_{\mathrm{SP}} \in S_{\tau,i}$ such that $\tau - 2\rho(\lambda_{\mathrm{SP}}) = k$.

▶ **Lemma 5.** *For any positive* τ, $\sum_{j=1}^{\lceil (n+1)/\tau \rceil} |B_{\tau,j}| = O(n + \frac{n}{\tau} \log n)$ *and* $\sum_{j=1}^{\lceil (n+1)/\tau \rceil} |E_{\tau,j}| = O(n + \frac{n}{\tau} \log n)$.

Proof. For each $\lambda \in LPR_\tau$, let $x_{\tau,\lambda}$ be the number of integers j such that $[(j-1)\tau + 1, j\tau] \cap SqBegRange(\lambda) \neq \emptyset$, and $y_{\tau,\lambda}$ be the number of integers j' such that $[(j'-1)\tau + 1, j'\tau] \subseteq SqBegRange(\lambda)$. Clearly, $x_{\tau,\lambda} \leq 2 + y_{\tau,\lambda}$ and $\tau y_{\tau,\lambda} \leq |SqBegRange(\lambda)|$ for each $\lambda \in LPR_\tau$. We obtain $\sum_{j=1}^{\lceil (n+1)/\tau \rceil} |B_{\tau,j}| = \sum_{\lambda \in LPR_\tau} x_{\tau,\lambda} \leq \sum_{\lambda \in LPR_\tau} (2 + y_{\tau,\lambda}) = 2|LPR_\tau| + \sum_{\lambda \in LPR_\tau} y_{\tau,\lambda} \leq 2|LPR_\tau| + (\sum_{\lambda \in LPR_\tau} |SqBegRange(\lambda)|)/\tau = O(n + \frac{n}{\tau} \log n)$. We can obtain $\sum_{j=1}^{\lceil (n+1)/\tau \rceil} |E_{\tau,j}| = O(n + \frac{n}{\tau} \log n)$ similarly. ◀

▶ **Lemma 6.** *For any parameter* τ *with* $1 \leq \tau \leq \omega$, *all bit arrays* $P_{\tau,1}, \ldots, P_{\tau,n+1}$ *can be computed in* $O(n)$ *time.*

Proof. Initially let $P_{\tau,i} \leftarrow 0$ for all $1 \leq i \leq n+1$. Also, for simplicity, we regard $P_{\tau,n+2} = 0$. Then, for each $\lambda_{\mathrm{SP}} \in SPR_\tau$, flip $P_{\tau,s}[\tau - 2\rho(\lambda_{\mathrm{SP}})]$ and $P_{\tau,e+1}[\tau - 2\rho(\lambda_{\mathrm{SP}})]$ where $[s, e] = SqEndRange(\lambda_{\mathrm{SP}})$. Finally, let $P_{\tau,i} \leftarrow P_{\tau,i} \oplus P_{\tau,i-1}$ for $i = 2$ to $n+1$, which can be done in $O(1)$ time for each operation since $\tau \leq \omega$. This algorithm takes $O(n + |SPR_\tau|) = O(n)$ time. Its correctness follows from the fact that for two different runs λ_1 and λ_2 in w, if $\rho(\lambda_1) = \rho(\lambda_2)$, then $SqEndRange(\lambda_1)$ and $SqEndRange(\lambda_2)$ do not overlap. ◀

In our algorithm, we process the blocks of C in descending order, from the $\lceil (n+1)/\tau \rceil$-th block to the first block of C. Suppose that we are going to process the j-th block of C. Here we assume that Algorithm 1 has already computed $C[j\tau + 1..n+1]$ correctly.

First, we handle short period runs. We process each $i \in [(j-1)\tau + 1, j\tau]$ in descending order. We assume that we have already computed $C[i..n+1]$ correctly. In Algorithm 1, if $C[i] = 1$, then we perform $C[i - 2\rho(\lambda_{\mathrm{SP}})] \leftarrow 1$ for each $\lambda_{\mathrm{SP}} \in S_{\tau,i}$. We can confirm that by the definition of $P_{\tau,i}$, it is equivalent to performing $C[i-\tau+1..i-1] \leftarrow C[i-\tau+1..i-1] \,|\, P_{\tau,i}$. Thus we can update the short period runs in $O(1)$ time for each position i.

After processing these short period runs, it is guaranteed that $C[(j-1)\tau + 1..n+1]$ is computed correctly. Next, we handle each long period run $\lambda_{\mathrm{LP}} \in E_{\tau,j}$. Let s, e be integers such that $[s + 2\rho(\lambda_{\mathrm{LP}}), e + 2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau + 1, j\tau] \cap SqEndRange(\lambda_{\mathrm{LP}})$. In Algorithm 1, we perform $C[s+k] \leftarrow C[s+k] \mid C[s+k+2\rho(\lambda_{\mathrm{LP}})]$ for each $k \in [0, e-s]$. Note that from the definition of long period runs, we obtain $e < s + \tau \leq s + 2\rho(\lambda_{\mathrm{LP}})$, which means that $[s, e]$ and $[s + 2\rho(\lambda_{\mathrm{LP}}), e + 2\rho(\lambda_{\mathrm{LP}})]$ do not overlap. Thus the operation

Algorithm 2: An $O(n + \frac{n}{\tau}\log n)$-time algorithm of determining whether w can be factorized into squares.

Input: String w of length n, and a parameter τ with $1 \leq \tau \leq \omega$.

Compute SPR_τ and LPR_τ;
Compute $E_{\tau,1}, \ldots, E_{\tau,\lceil (n+1)/\tau \rceil}$ from LPR_τ;
Compute $P_{\tau,1}, \ldots, P_{\tau,n+1}$ from SPR_τ;
$C[1..n] \leftarrow 0$; $C[n+1] \leftarrow 1$;
for $j = \lceil (n+1)/\tau \rceil$ **down to** 1 **do**
  for $i = \min\{j\tau, n+1\}$ **down to** $(j-1)\tau + 1$ **do**
    if $C[i] = 1$ **then**
      $C[i-\tau+1..i-1] \leftarrow C[i-\tau+1..i-1] \mid P_{\tau,i}$;
  foreach $\lambda_{\mathrm{LP}} \in E_{\tau,j}$ **do**
    Let s, e be integers such that
    $[s + 2\rho(\lambda_{\mathrm{LP}}), e + 2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau + 1, j\tau] \cap \mathit{SqEndRange}(\lambda_{\mathrm{LP}})$;
    $C[s..e] \leftarrow C[s..e] \mid C[s + 2\rho(\lambda_{\mathrm{LP}})..e + 2\rho(\lambda_{\mathrm{LP}})]$;

$C[s+k] \leftarrow C[s+k] \mid C[s+k+2\rho(\lambda_{\mathrm{LP}})]$ for each k can be done in parallel. Hence we perform $C[s..e] \leftarrow C[s..e] \mid C[s+2\rho(\lambda_{\mathrm{LP}})..e+2\rho(\lambda_{\mathrm{LP}})]$, which can be done in $O(1)$ time since $|[s,e]| \leq \tau \leq \omega$. Therefore it takes $O(|E_{\tau,j}|)$ time for long period runs in the j-th block of C. By Lemma 5, the computation on long period runs for all blocks can be done in $O(n + \frac{n}{\tau}\log n)$ time.

From above, we obtain Algorithm 2 and Lemma 7.

▶ **Lemma 7.** *For any parameter $1 \leq \tau \leq \omega$, Algorithm 2 determines whether w can be factorized into squares in $O(n + \frac{n}{\tau}\log n)$ time.*

Now we describe how to compute a square factorization of w. For a position i, we assume that $C[i] = 1$, i.e., $w[i..n]$ can be factorized into squares. First, we determine whether there exists $l \in [1, \tau-1]$ such that $C[i+l] = 1$ and $P_{\tau,i+l}[\tau - l] = 1$. This means that there exists some square factorization of $w[i..n]$ whose first factor is $w[i..i+l-1]$. In this case, we can spend $O(l)$ time to find such l, if any.

Next, we consider the case where there is no $l \in [1, \tau-1]$ s.t. $C[i+l] = 1$ and $P_{\tau,i+l}[\tau - l] = 1$. Then, there exists no short period run $\lambda_{\mathrm{SP}} \in S_{\tau,i+l}$ which satisfies $2\rho(\lambda_{\mathrm{SP}}) = l$ and $C[i+l] = 1$ for any l. In such a case, from the fact that $C[i] = 1$, there must exist some long period run $\lambda_{\mathrm{LP}} \in LPR_\tau$ such that $i \in \mathit{SqBegRange}(\lambda_{\mathrm{LP}})$ and $C[i + 2\rho(\lambda_{\mathrm{LP}})] = 1$. We scan all long period runs in $B_{\tau,\lceil i/\tau \rceil}$ and find such λ_{LP} in $O(|B_{\tau,\lceil i/\tau \rceil}|)$ time. Then, we use $w[i..i + 2\rho(\lambda_{\mathrm{LP}}) - 1]$ in the square factorization of $w[i..n]$. From above, we obtain Algorithm 3.

▶ **Lemma 8.** *For any parameter $1 \leq \tau \leq \omega$, Algorithm 3 computes a square factorization of w in $O(n + \frac{n}{\tau}\log n)$ time if it exists.*

Proof. We analyze the time complexity of Algorithm 3. For a position i with $C[i] = 1$, if there exists any short period run $\lambda_{\mathrm{SP}} \in B_{\tau,i+l}$ such that $2\rho(\lambda_{\mathrm{SP}}) = l$ and $C[i+l] = 1$ for any $l \geq 1$, we can compute $l = 2\rho(\lambda_{\mathrm{SP}})$ by scanning $P_{\tau,i+1}[\tau-1], P_{\tau,i+2}[\tau-2], \ldots, P_{\tau,i+l}[\tau-l]$ one by one. Note that if such $l \in [1, \tau-1]$ exists, then i increases by l, and hence we can afford to spend $O(l)$ time to find such l. Otherwise, we compute $l = 2\rho(\lambda_{\mathrm{LP}})$ for some $\lambda_{\mathrm{LP}} \in LPR_\tau$ such that $i \in \mathit{SqBegRange}(LPR_\tau)$ and $C[i + 2\rho(\lambda_{\mathrm{LP}})] = 1$; it takes $O(|B_{\tau,\lceil i/\tau \rceil}|)$

Algorithm 3: A linear-time algorithm of factorizing w into squares.

Input: String w of length n, and a parameter τ with $1 \leq \tau \leq \omega$.
Output: A square factorization of w if it exists; otherwise *nil*.

Compute SPR_τ and LPR_τ;
Compute $B_{\tau,1}, \ldots, B_{\tau,\lceil (n+1)/\tau \rceil}$ from LPR_τ;
Compute $P_{\tau,1}, \ldots, P_{\tau,n+1}, C$ by Algorithm 2;
if $C[1] = 1$ **then**
    $F \leftarrow ()$;
    $i \leftarrow 1$;
    while $i \leq n$ **do**
        $l \leftarrow 1$;
        while $l < \tau$ **do**
            if $C[i+l] = 1 \wedge P_{\tau,i+l}[\tau - l] = 1$ **then**
                break;
            $l \leftarrow l + 1$;
        if $l \geq \tau$ **then**
            Find any $\lambda_{\mathrm{LP}} \in B_{\tau,\lceil i/\tau \rceil}$ such that $i \in \mathit{SqBegRange}(\lambda_{\mathrm{LP}})$ and $C[i + 2\rho(\lambda_{\mathrm{LP}})] = 1$;
            `/* It is guaranteed that such` λ_{LP} `exists. */`
            $l \leftarrow 2\rho(\lambda_{\mathrm{LP}})$;
        Append $w[i..i+l-1]$ to the end of F;
        $i \leftarrow i + l$;
    return F;
else return *nil*;

time. Then we use $w[i..i+l-1]$ for a square factorization of $w[i..n]$ and increase i by l. Note that in this case, i increases by at least τ. Hence we can afford to spend $O(\tau)$ time before deciding to scan $B_{\tau,\lceil i/\tau \rceil}$. Moreover, for any $1 \leq j \leq \lceil (n+1)/\tau \rceil$, $B_{\tau,j}$ is scanned at most once. Therefore, using Lemma 5, we can show that Algorithm 3 takes $O(n + \frac{n}{\tau} \log n)$ time. ◀

Optimally, we choose $\tau = \omega$. Since $\omega = \Omega(\log n)$, by Lemma 8, we obtain Theorem 9.

▶ **Theorem 9.** *A square factorization of a string of length n can be computed in $O(n)$ time, if it exists.*

3.2 An algorithm for computing a largest square factorization

In this subsection we propose an algorithm for computing a largest square factorization of w. Note that any largest square factorization of w consists only of primitively rooted squares, since otherwise there exist a larger square factorization of w.

Let τ be some integer parameter such that $1 \leq \tau \leq \omega/(\lfloor \log n \rfloor + 1)$. As with Section 3.1, we define C, SPR_τ, LPR_τ, $S_{\tau,i}$ for each position $i \in [1, n+1]$, and $B_{\tau,j}$ and $E_{\tau,j}$ for each $j \in [1, \lceil (n+1)/\tau \rceil]$.

For each position i of string w, let us denote by $T[i]$ the size of largest square factorization of $w[i..n]$ if it exists; otherwise $T[i] = 0$. Algorithm 4 is a simple algorithm which computes the size of a largest square factorization of each suffix of w in $O(n \log n)$ time.

Algorithm 4: A simple algorithm for computing the size of largest square factorization of each suffix of w.

```
Input: String w of length n.
1 Compute R;
2 C[1..n] ← 0; C[n + 1] ← 1;
3 T[i] ← 0 for each i ∈ [1, n + 1];
4 for i = n + 1 down to 1 do
5     if C[i] = 1 then
6         foreach λ ∈ R such that i ∈ SqEndRange(λ) do
7             C[i − 2ρ(λ)] ← 1;
8             T[i − 2ρ(λ)] ← max{T[i − 2ρ(λ)], T[i] + 1};
```

Let $b = \lfloor \log n \rfloor$. For each position $i \in [1, n+1]$, let $P'_{\tau,i}$ be a bit array of length $(\tau-1)(b+1)$ such that for each $k \in [1, (\tau-1)(b+1)]$, $P'_{\tau,i}[k] = 1$ iff there exists $\lambda_{\mathrm{SP}} \in S_{\tau,i}$ such that $(\tau - 2\rho(\lambda_{\mathrm{SP}}))(b+1) = k$. We remark that $P'_{\tau,1}, \ldots, P'_{\tau,n+1}$ can be computed in $O(n)$ time in a similar way to Lemma 6. Also, let U be an array of bit arrays. For each $1 \leq i \leq n+1$, let $U[i]$ be a bit array of length $b+1$ such that $U[i][1] = C[i]$, and $U[i][2..b+1]$ is the binary representation of $T[i]$. Note that $T[i]$ can be represented as an unsigned b-bit integer since $T[i] \leq n/2$. For convenience, we regard $U[i] = 0$ if $i < 1$ or $i > n+1$. In addition, for two integers s, e with $s \leq e$, we denote by $U[s..e]$ the concatenation $U[s] \cdot U[s+1] \cdots U[e]$, which we also regard as an unsigned $((b+1)(e-s+1))$-bit integer where the most significant bit is $U[s][1]$ and the least significant bit is $U[e][b+1]$. To obtain a largest square factorization quickly, we compute U instead of T and C.

For each $1 \leq j \leq \lceil (n+1)/\tau \rceil$, we call $U[(j-1)\tau + 1..j\tau]$ the j-th block of U. As with Algorithm 2, we process the blocks of U in descending order, from the $\lceil (n+1)/\tau \rceil$-th block to the first block of U. Suppose that we are going to process the j-th block of U. Here we assume that our algorithm has already computed $U[j\tau + 1..n+1]$ correctly.

See Algorithm 5 for computing a largest square factorization, where M serves as a fixed-length $(\tau(b+1))$ bit array to specify the runs to be processed at once (Lines 7-8).

First, we handle short period runs. We process each $i \in [(j-1)\tau+1, j\tau]$ in descending order. We assume that we have already computed $U[i..n+1]$ correctly. As with Algorithm 4, if $C[i] = 1$, then we perform $C[i - 2\rho(\lambda_{\mathrm{SP}})] \leftarrow 1$ and $T[i - 2\rho(\lambda_{\mathrm{SP}})] \leftarrow \max\{T[i - 2\rho(\lambda_{\mathrm{SP}})], T[i] + 1\}$ for each $\lambda_{\mathrm{SP}} \in S_{\tau,i}$. In other words, if $U[i][1] = 1$, then we perform $U[i - 2\rho(\lambda_{\mathrm{SP}})][1] \leftarrow 1$ and $U[i - 2\rho(\lambda_{\mathrm{SP}})][2..b+1] \leftarrow \max\{U[i - 2\rho(\lambda_{\mathrm{SP}})][2..b+1], U[i][2..b+1] + 1\}$ for each $\lambda_{\mathrm{SP}} \in S_{\tau,i}$. It is equivalent to performing $U[i - 2\rho(\lambda_{\mathrm{SP}})] \leftarrow \max\{U[i - 2\rho(\lambda_{\mathrm{SP}})], U[i] + 1\}$ if $U[i][1] = 1$. We process all short period runs in $S_{\tau,i}$ in parallel. Note that since $2\rho(\lambda_{\mathrm{SP}}) < \tau$ for any $\lambda_{\mathrm{SP}} \in S_{\tau,i}$, we update $U[i - \tau + 1..i - 1]$ by taking $U[i]$ and $S_{\tau,i}$ into consideration. We show the method in Lines 12–17 of Algorithm 5, where M' in Line 13 serves as a bit array of length $(\tau-1)(b+1)$. The fact that $(\tau-1)(b+1) < \omega$ implies that the operations in Lines 12–17 can be performed in constant time.

▶ **Example 10.** Here, we explain the situation with some i and $\lambda_{\mathrm{SP}} \in S_{\tau,i}$ using a concrete example. Let $Y = (U[i] + 1)P'_{\tau,i}$ where Y is a bit array of length $(\tau-1)(b+1)$. In this example, we consider the case when $b = 4, \tau = 7, P'_{\tau,i} = 00001\,00000\,00000\,00000\,00001\,00000$ in binary representation and $U[i] = 10110$ in binary representation, which means that $C[i] = 1$ and $T[i] = 6$. Then we obtain $Y = 10111\,00000\,00000\,00000\,10111\,00000$. Intuitively, we *copy* $U[i] + 1$ to the appropriate positions. After that, in order to update $U[i - \tau + k]$

with $\max\{U[i-\tau+k], U[i]+1\}$ for each $k \in [1, \tau-1]$ with $P'_{\tau,i}[k(b+1)] = 1$, we perform $U[i-\tau+k] \leftarrow \max\{U[i-\tau+k], Y[(k-1)(b+1)+1..k(b+1)\}$ for each $k \in [1, \tau-1]$. This remaining part is almost same as Lines 23–26, which we will explain in Example 11.

After processing these short period runs, it is guaranteed that $U[(j-1)\tau+1..n+1]$ is computed correctly. Next, we handle long period runs in Lines 19–26, where again M' serves as a variable-length (up to $\tau(b+1)$) bit array. Consider each long period run $\lambda_{\mathrm{LP}} \in E_{\tau,j}$. Let s, e be integers such that $[s+2\rho(\lambda_{\mathrm{LP}}), e+2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau+1, j\tau] \cap \mathit{SqEndRange}(\lambda_{\mathrm{LP}})$. In Algorithm 4, for each $k \in [0, e-s]$, we perform $C[s+k] \leftarrow 1$ and $T[s+k] \leftarrow \max\{T[s+k], T[s+k+2\rho(\lambda_{\mathrm{LP}})]+1\}$ if $C[s+k+2\rho(\lambda_{\mathrm{LP}})] = 1$. Note that from the definition of long period runs, we obtain $e < s+\tau \leq s+2\rho(\lambda_{\mathrm{LP}})$, which means that $[s, e]$ and $[s+2\rho(\lambda_{\mathrm{LP}}), e+2\rho(\lambda_{\mathrm{LP}})]$ do not overlap. Thus we process all k's in parallel. We next analyze Lines 19–26 of Algorithm 5. Since $e-s+1 \leq \tau$, we obtain $(e-s+1)(b+1) \leq \tau(b+1) \leq \omega$, which means that we can perform Lines 19–26 in constant time.

▶ **Example 11.** Here, we explain the situation with some j and $\lambda_{\mathrm{LP}} \in E_{\tau,j}$ using a concrete example. Let s, e be integers such that $[s+2\rho(\lambda_{\mathrm{LP}}), e+2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau+1, j\tau] \cap \mathit{SqEndRange}(\lambda_{\mathrm{LP}})$. Also, let $s' = s+2\rho(\lambda_{\mathrm{LP}})$ and $e' = e+2\rho(\lambda_{\mathrm{LP}})$. In Lines 19–26, we update $U[s..e]$ by taking $U[s'..e']$ into consideration.

1. Let $X = U[s..e]$ and $Y = U[s'..e']$. In this example, we consider the case when $b = 4, |[s, e]| = 5, X = 10010\,00000\,00000\,00000\,11010$ in binary representation and $Y = 10110\,00000\,11000\,00000\,10101$ in binary representation, which means that $C[s..e] = (1,0,0,0,1), T[s..e] = (2,0,0,0,10), C[s'..e'] = (1,0,1,0,1)$ and $T[s'..e'] = (6,0,8,0,5)$. Also, let $M' = 10000\,10000\,10000\,10000\,10000$ in binary representation.
2. Let $Y' = Y + ((Y \,\&\, M') \gg b)$. Then we obtain $Y' = 10111\,00000\,11001\,00000\,10110$. Intuitively, it represents $T[s'+k]+1$ for each k with $C[s'+k] = 1$.
3. Let $D = (Y' \mid M') - (X \,\&\, {\sim}M')$. Then we obtain $D = 10101\,10000\,11001\,10000\,01100$. Intuitively, it represents $(T[s'+k]+1) - T[s+k]$ for each k.
4. Let $D' = ((D \,\&\, M') \gg b)(2^b - 1)$. Then we obtain $D' = 01111\,01111\,01111\,01111\,00000$. Intuitively, it indicates all positions k such that $T[s'+k]+1 \geq T[s+k]$.
5. Let $Z = D \,\&\, D'$. Then we obtain $Z = 00101\,00000\,01001\,00000\,00000$. Intuitively, it represents $\max\{(T[s'+k]+1) - T[s+k], 0\}$ for each k.
6. Let $Z' = Z + X$. Then we obtain $Z' = 10111\,00000\,01001\,00000\,11010$. Intuitively, it represents $\max\{T[s'+k]+1, T[s+k]\}$ for each k.
7. Compute $Z'' = Z' \mid (Y \,\&\, M')$ to set $C[s..e]$ appropriately. Then we obtain $Z'' = 10111\,00000\,11001\,00000\,11010$. Finally, we substitute Z'' for $U[s..e]$. Then we obtain $C[s..e] = (1,0,1,0,1)$ and $T[s..e] = (7,0,9,0,10)$ as a result.

After computing $U[1..n+1]$, we obtain a largest square factorization of w as in Algorithm 3.

▶ **Lemma 12.** *Algorithm 5 computes a largest square factorization of w in $O(n + \frac{n}{\tau}\log n)$ time for any parameter $1 \leq \tau \leq \omega/(\lfloor \log n \rfloor + 1)$.*

Proof. Clearly, Algorithm 5 requires $O(n + \sum_{j=1}^{\lceil (n+1)/\tau \rceil} |B_{\tau,j}| + \sum_{j=1}^{\lceil (n+1)/\tau \rceil} |E_{\tau,j}|)$ time. Therefore, from Lemma 5, Algorithm 5 runs in $O(n + \frac{n}{\tau}\log n)$ time in total. ◀

The optimal strategy is to choose $\tau = \lfloor \omega/(\lfloor \log n \rfloor + 1) \rfloor$. Thus, we obtain Theorem 13.

▶ **Theorem 13.** *A largest square factorization of a string of length n can be computed in $O(n + \frac{n}{\omega}\log^2 n)$ time.*

Algorithm 5: An algorithm for computing a largest square factorization of w.

Input: String w of length n, and a parameter τ with $1 \leq \tau \leq \omega/(\lfloor \log n \rfloor + 1)$.
Output: A largest square factorization of w if it exists; otherwise *nil*.

Compute SPR_τ and LPR_τ;
Compute $B_{\tau,1}, \ldots, B_{\tau,\lceil (n+1)/\tau \rceil}, E_{\tau,1}, \ldots, E_{\tau,\lceil (n+1)/\tau \rceil}$ from LPR_τ;
Compute $P'_{\tau,1}, \ldots, P'_{\tau,n+1}$ from SPR_τ;
$U[i][1..b+1] \leftarrow 0$ for each $i \in [1, n+1]$;
$U[n+1][1] \leftarrow 1$; `/* equivalent to` $C[n+1] \leftarrow 1$ `*/`
$b \leftarrow \lfloor \log n \rfloor$;
$M[1..\tau(b+1)] \leftarrow 0$ where M is a bit array of length $\tau(b+1)$;
$M[k(b+1)-b] \leftarrow 1$ for each $k \in [1, \tau]$;
for $j = \lceil (n+1)/\tau \rceil$ **down to** 1 **do**
  for $i = \min\{j\tau, n+1\}$ **down to** $(j-1)\tau + 1$ **do**
    if $U[i][1] = 1$ **then**
      $Y \leftarrow (U[i]+1)P'_{\tau,i}$ where Y is a bit array of length $(\tau-1)(b+1)$;
      $M' \leftarrow M[1..(\tau-1)(b+1)]$;
      $X \leftarrow U[i-\tau+1..i-1]$;
      $D \leftarrow (Y \mid M') - (X \,\&\, {\sim}M')$;
      $D' \leftarrow ((D \,\&\, M') \gg b)(2^b - 1)$;
      $U[i-\tau+1..i-1] \leftarrow ((D' \,\&\, D) + X) \mid (Y \,\&\, M')$;
  foreach $\lambda_{\mathrm{LP}} \in E_{\tau,j}$ **do**
    Let s, e be integers such that
    $[s + 2\rho(\lambda_{\mathrm{LP}}), e + 2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau + 1, j\tau] \cap \mathit{SqEndRange}(\lambda_{\mathrm{LP}})$;
    $M' \leftarrow M[1..(e-s+1)(b+1)]$;
    $Y \leftarrow U[s + 2\rho(\lambda_{\mathrm{LP}})..e + 2\rho(\lambda_{\mathrm{LP}})]$;
    $Y' \leftarrow Y + ((Y \,\&\, M') \gg b)$;
    $X \leftarrow U[s..e]$;
    $D \leftarrow (Y' \mid M') - (X \,\&\, {\sim}M')$;
    $D' \leftarrow ((D \,\&\, M') \gg b)(2^b - 1)$;
    $U[s..e] \leftarrow ((D' \,\&\, D) + X) \mid (Y \,\&\, M')$;
if $U[1][1] = 1$ **then** `/* equivalent to` $C[1] = 1$ `*/`
  $F \leftarrow ()$;
  $i \leftarrow 1$;
  while $i \leq n$ **do**
    $l \leftarrow 1$;
    while $l < \tau$ **do**
      if $U[i+l] + 1 = U[i] \wedge P'_{\tau,i+l}[(\tau-l)(b+1)] = 1$ **then break**;
      $l \leftarrow l + 1$;
    if $l \geq \tau$ **then**
      Find any $\lambda_{\mathrm{LP}} \in B_{\tau,\lceil i/\tau \rceil}$ such that $i \in \mathit{SqBegRange}(\lambda_{\mathrm{LP}})$ and
      $U[i + 2\rho(\lambda_{\mathrm{LP}})] + 1 = U[i]$;
      $l \leftarrow 2\rho(\lambda_{\mathrm{LP}})$;
    Append $w[i..i+l-1]$ to the end of F;
    $i \leftarrow i + l$;
  return F;
else return *nil*;

4 Other problems and further work

In this section we discuss several connected problems. The first such problem is that of computing a smallest square factorization of a string (i.e., a square factorization with a minimum number of factors). Unlike the factorizations produced as solutions to the previous problems, in such a factorization the square factors we use are no longer necessarily primitively rooted. Thus, it seems that a slightly different strategy might be needed to solve this problem. Next, we propose an $O(n \log n)$ time algorithm for computing a smallest square factorization of a string w of length n, over $\Sigma = [1, n]$.

Following Lemma 3, let us assume that $x_1^2, \ldots, x_k^2$ are all the primitively rooted squares starting at position i of w, with $|x_j| < |x_{j+1}|$ for $1 \leq j \leq k-1$. If there exists a position i' of w where x_h^2 starts, for some $h \leq k$, we have that the primitively rooted squares whose root is shorter than x_h^2 starting at i' are exactly $x_1^2, \ldots, x_{h-1}^2$, so x_h^2 is the h-th primitively rooted square occurring at both positions i and i', in the list of such squares ordered increasingly w.r.t. their length.

Further, we can produce for each $i \leq n$ the list of all primitively rooted squares starting at i in $O(n \log n)$ time; there are at most $2 \log n$ such squares for each i. We define the $(n+1) \times (1 + 2 \log n)$ matrix Q, where, for $1 \leq i \leq n$ and $1 \leq j \leq 2 \log n$ we have that $Q[i][j]$ is the number of factors in a smallest square factorization of $w[i..n]$, such that the first square of this factorization is a power of the j-th primitively rooted square in the list of primitively rooted squares starting at i, ordered increasingly w.r.t. their length (or undefined, if there are less than j primitively rooted squares starting at i). Moreover, $Q[i][0] = \min\{Q[i][j] \mid 1 \leq j \leq 2 \log n\}$, and $Q[n+1][j] = 0$ for all $0 \leq j \leq 2 \log n$.

The values stored in this matrix can be computed by dynamic programming. Assume we are computing $Q[i][j]$ with $i \leq n$ and $1 \leq j \leq 2 \log n$, and there are at least j primitively rooted squares starting at position i of w; at this point in our computation we already know all the values stored in the arrays $Q[i'][\cdot]$ for $i' > i$. When x_j^2 occurs also at position $i + 2|x_j|$, by the preliminary remark we made, we get that x_j^2 is the j-th primitively rooted square in the list of such squares occurring at position $i + 2|x_j|$; so we can compute $Q[i][j]$ as the minimum between $Q[i + 2|x_j|][0] + 1$ and $Q[i + 2|x_j|][j]$. Indeed, either the first square in the factorization of $w[i..n]$ is x_j^2, and then we continue with the smallest square factorization of $w[i + 2|x_j|..n]$; or the first square in the factorization of $w[i..n]$ is x_j^{2k} for some $k > 1$, and the smallest square factorization of $w[i + 2|x_j|..n]$ started with x_j^{2k-2}. The case when x_j^2 does not occur at $i + 2|x_j|$ is much simpler: we just set $Q[i][j]$ as $Q[i + 2|x_j|][0] + 1$. Finally, after computing $Q[i][j]$ for all $1 \leq j \leq 2 \log n$, we set $Q[i][0]$ as their minimum. Clearly, this process can be easily implemented in $O(n \log n)$ time, with the help of data structures allowing us to test whether some primitively rooted square x_j^2 occurring at position i also occurs at some other position $i + 2|x_j|$, like, e.g., the data structures *SqBegRange* and *SqEndRange*.

The number of factors in a smallest square factorization of w can be now found in $Q[1][0]$, while this factorization can be effectively obtained by tracing back the computation of $Q[1][0]$ via dynamic programming. Thus, we have shown the following result.

▶ **Theorem 14.** *A smallest square factorization of w can be obtained in $O(n \log n)$ time.*

We conjecture that a more efficient implementation of the above solution can be obtained using and extending the ideas in the previous section.

Two other open problems connected to square factorizations of strings are the following. The first one is inspired by the work of [15]: given a string w and a number k decide whether w has a square factorization with exactly k factors. The second one follows the line of

research in [2]: given a string w decide whether there exists a square factorization of w whose factors are each two distinct (i.e., a diverse square factorization of w). While we expect that the first problem can be solved efficiently, we conjecture that the second one is NP-Complete.

The strategy employed in Section 3.1 seems to also lead to an improvement in deciding whether a string w_1, of length n, can be obtained from other string w_2, of length $m < n$, by iterated prefix-suffix duplication [10]. Prefix-suffix duplication is a string operation that rewrites a string $u = xwy$ into xu (prefix-duplication) or uy (suffix-duplication). Accordingly, in the respective problem one asks whether there exists a sequence of prefix-suffix duplications that can be applied to w_1 so that in the end we get w_2; state-of-the-art algorithms [10] solved this problem in $O(n^2 \log n)$ time or, alternatively, in $O(n \log n)$ time if we allow only suffix-duplications or only prefix-duplications to be applied in order to obtain w_1 from w_2. We conjecture that using the ideas of Section 3.1 we can shave a $\log n$ factor from both of these complexities. For instance, if we consider the case of only suffix-duplications, we basically have to decide the existence of a factorization of w into $w = x_0 \cdots x_k$ such that $x_0 = w_1$ and x_i is a primitive string which is a suffix of $x_0 \cdots x_{i-1}$; in other words, x_i is a primitively rooted square centered at position $|x_0 \cdots x_{i-1}| + 1$. This problem greatly resembles to the problem of factoring a string into squares and we conjecture that it can be solved by the same methods, within the same linear time complexity.

References

1. Golnaz Badkobeh, Hideo Bannai, Keisuke Goto, Tomohiro I, Costas S. Iliopoulos, Shunsuke Inenaga, Simon J. Puglisi, and Shiho Sugimoto. Closed factorization. In *Proc. PSC 2014*, pages 162–168, 2014.
2. Hideo Bannai, Travis Gagie, Shunsuke Inenaga, Juha Kärkkäinen, Dominik Kempa, Marcin Piatkowski, Simon J. Puglisi, and Shiho Sugimoto. Diverse palindromic factorization is np-complete. In *Proc. DLT 2015*, volume 9168 of *Lecture Notes in Comput. Sci.*, pages 85–96. Springer, 2015.
3. Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *CoRR*, abs/1406.0263, 2014. URL: `http://arxiv.org/abs/1406.0263`.
4. Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.
5. Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the lempel ziv factorization. In *Proc. DCC 2008*, pages 482–488. IEEE, 2008.
6. Maxime Crochemore and Wojciech Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. `doi:10.1007/BF01190846`.
7. Marius Dumitran, Florin Manea, and Dirk Nowotka. On prefix/suffix-square free words. In *Proc. SPIRE 2015*, volume 9309 of *Lecture Notes in Comput. Sci.*, pages 54–66. Springer, 2015.
8. Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. `doi:10.1016/0196-6774(83)90017-2`.
9. Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. A subquadratic algorithm for minimum palindromic factorization. *J. Discrete Algorithms*, 28:41–48, 2014.
10. Jesús García-López, Florin Manea, and Victor Mitrana. Prefix-suffix duplication. *J. Comput. Syst. Sci.*, 80(7):1254–1265, 2014. `doi:10.1016/j.jcss.2014.02.011`.
11. Torben Hagerup. Sorting and searching on the word RAM. In *Proc. STACS 1998*, volume 1373 of *Lecture Notes in Comput. Sci.*, pages 366–398. Springer, 1998.

12 Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing palindromic factorizations and palindromic covers on-line. In *Proc. CPM 2014,*, volume 8486 of *Lecture Notes in Comput. Sci.*, pages 150–161. Springer, 2014.

13 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.

14 Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. FOCS 1999*, pages 596–604. IEEE, 1999.

15 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Pal^k is linear recognizable online. In *SOFSEM 2015*, volume 8939 of *Lecture Notes in Comput. Sci.*, pages 289–301. Springer, 2015.

16 Manfred Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. PSC 2009*, pages 65–79, 2009.

17 Roger C. Lyndon. On Burnside's problem. *Trans. Amer. Math. Soc.*, 77(2):202–215, 1954. URL: http://www.jstor.org/stable/1990868.

18 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.

19 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

20 Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

21 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977.

22 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978.

Minimal Suffix and Rotation of a Substring in Optimal Time*

Tomasz Kociumaka

Institute of Informatics, University of Warsaw, ul. Stefana Banacha 2, 02-097 Warsaw, Poland
kociumaka@mimuw.edu.pl

Abstract

For a text of length n given in advance, the substring minimal suffix queries ask to determine the lexicographically minimal non-empty suffix of a substring specified by the location of its occurrence in the text. We develop a data structure answering such queries optimally: in constant time after linear-time preprocessing. This improves upon the results of Babenko et al. (CPM 2014), whose trade-off solution is characterized by $\Theta(n \log n)$ product of these time complexities. Next, we extend our queries to support concatenations of $\mathcal{O}(1)$ substrings, for which the construction and query time is preserved. We apply these generalized queries to compute lexicographically minimal and maximal rotations of a given substring in constant time after linear-time preprocessing.

Our data structures mainly rely on properties of Lyndon words and Lyndon factorizations. We combine them with further algorithmic and combinatorial tools, such as fusion trees and the notion of order isomorphism of strings.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases minimal suffix, minimal rotation, Lyndon factorization, substring canonization, substring queries

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.28

1 Introduction

Lyndon words, as well as the inherently linked concepts of the lexicographically minimal suffix and the lexicographically minimal rotation of a string, are one of the most successful concepts of combinatorics of words. Introduced by Lyndon [26] in the context of Lie algebras, they are widely used in algebra and combinatorics. They also have surprising algorithmic applications, including ones related to constant-space pattern matching [13], maximal repetitions [6], and the shortest common superstring problem [28].

The central combinatorial property of Lyndon words, proved by Chen et al. [8], states that every string can be uniquely decomposed into a non-increasing sequence of Lyndon words. Duval [14] devised a simple algorithm computing the Lyndon factorization in linear time and constant space. He also observed that the same algorithm can be used to determine the lexicographically minimal and maximal suffix, as well as the lexicographically minimal and maximal rotation of a given string.

The first two algorithms are actually on-line procedures: in linear time they allow computing the minimal and maximal suffix of every prefix of a given string. For rotations

* This work is supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.

such a procedure was later introduced by Apostolico and Crochemore [3]. Both these solutions lead to the optimal, quadratic-time algorithms computing the minimal and maximal suffixes and rotations for all substring of a given string. Our main results are the data-structure versions of these problems: we preprocess a given text T to answer the following queries:

▶ **Problem** (Minimal Suffix Queries). *Given a substring $v = T[\ell..r]$ of T, report the lexicographically smallest non-empty suffix of v (represented by its length).*

▶ **Problem** (Minimal Rotation Queries). *Given a substring $v = T[\ell..r]$ of T, report the lexicographically smallest rotation of v (represented by the number of positions to shift).*

For both problems we obtain optimal solutions with linear construction time and constant query time. For Minimal Suffix Queries this improves upon the results of Babenko et al. [4], who developed a trade-off solution, which for a text of length n has $\Theta(n \log n)$ product of preprocessing and query time. We are not aware of any results for Minimal Rotation Queries except for a data structure only testing cyclic equivalence of two subwords [24]. It allows constant-time queries after randomized preprocessing running in expected linear time.

An optimal solution for the Maximal Suffix Queries was already obtained in [4], while the Maximal Rotation Queries are equivalent to Minimal Rotation Queries subject to alphabet reversal. Hence, we do not focus on the maximization variants of our problems.

Using an auxiliary result devised to handle Minimal Rotation Queries, we also develop a data structure answering in $\mathcal{O}(k^2)$ time the following generalized queries:

▶ **Problem** (Generalized Minimal Suffix Queries). *Given a sequence of substrings $v_1, \ldots, v_k$ ($v_i = T[\ell_i..r_i]$), report the lexicographically smallest non-empty suffix of their concatenation $v_1 v_2 \ldots v_k$ (represented by its length).*

All our algorithms are deterministic procedures for the standard word RAM model with machine words of size $W = \Omega(\log n)$ [17]. The alphabet is assumed to be $\Sigma = \{0, \ldots, \sigma - 1\}$ where $\sigma = n^{\mathcal{O}(1)}$, so that all letters of the input text T can be sorted in linear time.

Applications The last factor of the Lyndon factorization of a string is its minimal suffix. As noted in [4], this can be used to reduce computing the factorization $v = v_1^{p_1} \cdots v_m^{p_m}$ of a substring $v = T[\ell..r]$ to $\mathcal{O}(m)$ Minimal Suffix Queries in T. Hence, our data structure determines the factorization in the optimal $\mathcal{O}(m)$ time. If v is a concatenation of k substrings, this increases to $\mathcal{O}(k^2 m)$ time (which we did not attempt to optimize in this paper).

The primary use of Minimal Rotation Queries is *canonization* of substrings, i.e., classifying them according to cyclic equivalence (conjugacy); see [3]. As a proof-of-concept application of this natural tool, we propose counting distinct substring with a given exponent.

Related work Our work falls in a class of substring queries: data structure problems solving basic stringology problems for substrings of a preprocessed text. This line of research, implicitly initiated by substring equality and longest common prefix queries (using suffix trees and suffix arrays; see [10]), now includes several problems related to compression [9, 22, 24, 5], pattern matching [24], and the range longest common prefix problem [1, 2]. Closest to ours is a result by Babenko et al. [5], which after $\mathcal{O}(n\sqrt{\log n})$-expected-time preprocessing allows determining the k-th smallest suffix of a given substring, as well as finding the lexicographic rank of one substring among suffixes of another substring, both in logarithmic time.

Outline of the paper In Section 2 we recall standard definitions and two well-known data structures. Next, in Section 3, we study combinatorics of minimal suffixes, using in particular a notion of *significant suffixes*, introduced by I et al. [19, 20] to compute Lyndon factorizations of grammar-compressed strings. Section 4 is devoted to answering Minimal Suffix Queries. We use *fusion trees* by Pătraşcu and Thorup [29] to improve the query time from logarithmic to $\mathcal{O}(\log^* |v|)$, and then, by preprocessing shorts strings, we achieve constant query time. That final step uses a notion of *order-isomorphism* [25, 23] to reduce the number of precomputed values. In Section 5 we repeat the same steps for Generalized Minimal Suffix Queries. We conclude with Section 6, where we briefly discuss the applications.

2 Preliminaries

We consider strings over an alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$ with the natural order $\prec$. The empty string is denoted as ε. By Σ^* (Σ^+) we denote the set of all (resp. non-empty) finite strings over Σ. We also define Σ^∞ as the set of infinite strings over Σ. We extend the order $\prec$ on Σ in the standard way to the *lexicographic* order on $\Sigma^* \cup \Sigma^\infty$.

Let $w = w[1] \ldots w[n]$ be a string in Σ^*. We call n the *length* of w and denote it by $|w|$. For $1 \le i \le j \le n$, a string $u = w[i] \ldots w[j]$ is called a *substring* of w. By $w[i..j]$ we denote the occurrence of u at position i, called a *fragment* of w. A fragment of w other than the whole w is called a *proper* fragment of w. A fragment starting at position 1 is called a *prefix* of w and a fragment ending at position n is called a *suffix* of w. We use abbreviated notation $w[..j]$ and $w[i..]$ for a prefix $w[1..j]$ and a suffix $w[i..n]$ of w, respectively. A *border* of w is a substring of w which occurs both as a prefix and as a suffix of w. An integer p, $1 \le p \le |w|$, is a *period* of w if $w[i] = w[i+p]$ for $1 \le i \le n - p$. If w has period p, we also say that is has *exponent* $\frac{|w|}{p}$. Note that p is a period of w if and only if w has a border of length $|w| - p$.

We say that a string w' is a *rotation* (cyclic shift, conjugate) of a string w if there exists a decomposition $w = uv$ such that $w' = vu$. Here, w' is the left rotation of w by $|u|$ characters and the right rotation of w by $|v|$ characters.

Augmented suffix array The *suffix array* [27] of a text T of length n is a permutation SA of $\{1, \ldots, n\}$ defining the lexicographic order on suffixes $T[i..n]$: $T[SA[i]..n] \prec T[SA[j]..n]$ if and only if $i < j$. For a string T, both SA and its inverse permutation ISA take $\mathcal{O}(n)$ space and can be computed in $\mathcal{O}(n)$ time; see e.g. [10]. Typically, one also builds the LCP table and extends it with a data structure for range minimum queries [18, 7], so that the longest common prefix of any two suffixes of T can be determined efficiently.

Similarly to [4], we also construct these components for the reversed text T^R. Additionally, we preprocess the ISA table to answer range minimum and maximum queries. The resulting data structure, which we call the *augmented suffix array* of T, lets us perform many queries.

▶ **Theorem 1** (Augmented suffix array; see Fact 3 and Lemma 4 in [4]). *The augmented suffix array of a text T of length n takes $\mathcal{O}(n)$ space, can be constructed in $\mathcal{O}(n)$ time, and allows answering the following queries in $\mathcal{O}(1)$ time given fragments x, y of T:*

1. *determine if $x \prec y$, $x = y$, or $x \succ y$,*
2. *compute the longest common prefix $\mathrm{lcp}(x, y)$ and the longest common suffix $\mathrm{lcs}(x, y)$,*
3. *compute $\mathrm{lcp}(x^\infty, y)$ and determine if $x^\infty \prec y$, $x^\infty = y$, or $x^\infty \succ y$.*

Moreover, given indices i, j, it can compute in $\mathcal{O}(1)$ time the minimal *and the* maximal *suffix among $\{T[k..n] : i \le k \le j\}$.*

Fusion trees Consider a set $\mathcal{A}$ of W-bit integers (recall that W is the machine word size). *Rank* queries given a W-bit integer x return $\mathrm{rank}_{\mathcal{A}}(x)$ defined as $|\{y \in \mathcal{A} : y < x\}|$. Similarly, *select* queries given an integer r, $0 \le r < |\mathcal{A}|$, return $\mathrm{select}_{\mathcal{A}}(r)$, the r-th smallest element in $\mathcal{A}$, i.e., $x \in \mathcal{A}$ such that $\mathrm{rank}_{\mathcal{A}}(x) = r$. These queries can be used to determine the *predecessor* and the *successor* of a W-bit integer x, i.e., $\mathrm{pred}_{\mathcal{A}}(x) = \max\{y \in \mathcal{A} : y < x\}$ and $\mathrm{succ}_{\mathcal{A}}(x) = \min\{y \in \mathcal{A} : y \ge x\}$. We answer these queries with dynamic fusion trees by Pătraşcu and Thorup [29]. We only use these trees in a static setting, but the original static fusion trees by Fredman and Willard [15] do not have an efficient construction procedure.

▶ **Theorem 2** (Fusion trees [29, 15]). *There exists a data structure of size $\mathcal{O}(|\mathcal{A}|)$ which answers* $\mathrm{rank}_{\mathcal{A}}$, $\mathrm{select}_{\mathcal{A}}$, $\mathrm{pred}_{\mathcal{A}}$, *and* $\mathrm{succ}_{\mathcal{A}}$ *queries in $\mathcal{O}(1 + \log_W |\mathcal{A}|)$ time. Moreover, it can be constructed in $\mathcal{O}(|\mathcal{A}| + |\mathcal{A}| \log_W |\mathcal{A}|)$ time.*

3 Combinatorics of minimal suffixes and Lyndon words

For a non-empty string v the *minimal suffix* $\mathrm{MinSuf}(v)$ is the lexicographically smallest non-empty suffix s of v. Similarly, for an arbitrary string v the *maximal suffix* $\mathrm{MaxSuf}(v)$ is the lexicographically largest suffix s of v. We extend these notions as follows: for a pair of strings v, w we define $\mathrm{MinSuf}(v, w)$ and $\mathrm{MaxSuf}(v, w)$ as the lexicographically smallest (resp. largest) string sw such that s is a (possibly empty) suffix of v.

In order to relate minimal and maximal suffixes, we introduce the *reverse* order $\prec^R$ on Σ and extend it to the *reverse lexicographic* order, and an auxiliary symbol $\$ \notin \Sigma$. We extend the order $\prec$ on Σ so that $\mathtt{c} \prec \$$ (and thus $\$ \prec^R \mathtt{c}$) for every $\mathtt{c} \in \Sigma$. We define $\bar{\Sigma} = \Sigma \cup \{\$\}$, but unless otherwise stated, we still assume that the strings considered belong to Σ^*.

▶ **Observation 3.** *If $u, v \in \Sigma^*$, then $u\$ \prec v$ if and only if $v \prec^R u$.*

We use MinSuf^R and MaxSuf^R to denote the minimal (resp. maximal) suffix with respect to $\prec^R$. The following observation relates the notions we introduced:

▶ **Observation 4.**
1. $\mathrm{MaxSuf}(v, \varepsilon) = \mathrm{MaxSuf}(v)$ *for every* $v \in \bar{\Sigma}^*$,
2. $\mathrm{MinSuf}(vw) = \min(\mathrm{MinSuf}(v, w), \mathrm{MinSuf}(w))$ *for every* $v \in \bar{\Sigma}^*$ *and* $w \in \bar{\Sigma}^+$,
3. $\mathrm{MinSuf}(vc) = \mathrm{MinSuf}(v, c)$ *for every* $v \in \bar{\Sigma}^*$ *and* $c \in \bar{\Sigma}$,
4. $\mathrm{MinSuf}(v, w\$) = \mathrm{MaxSuf}^R(v, w)\$$ *for every* $v, w \in \Sigma^*$,
5. $\mathrm{MinSuf}(v\$) = \mathrm{MaxSuf}^R(v)\$$ *for every* $v \in \Sigma^*$.

A property seemingly similar to 5. is false for every $v \in \Sigma^+$: $\$ = \mathrm{MinSuf}^R(v\$) \neq \mathrm{MaxSuf}(v)\$$.

A notion deeply related to minimal and maximal suffixes is that of a Lyndon word [26, 8]. A string $w \in \Sigma^+$ is called a *Lyndon word* if $\mathrm{MinSuf}(w) = w$. Note that such w does not have proper borders, since a border would be a non-empty suffix smaller than w. A *Lyndon factorization* of a string $u \in \bar{\Sigma}^*$ is a representation $u = u_1^{p_1} \dots u_m^{p_m}$, where u_i are Lyndon words such that $u_1 \succ \dots \succ u_m$. Every non-empty word has a unique Lyndon factorization [8], which can be computed in linear time and constant space [14].

3.1 Significant suffixes

Below we recall a notion of *significant suffixes*, introduced by I et al. [19, 20] in order to compute Lyndon factorizations of grammar-compressed strings. Then, we state combinatorial properties of significant suffixes; some of them are novel and some were proved in [20].

▶ **Definition 5** (see [19, 20]). A suffix s of a string $v \in \Sigma^*$ is a *significant suffix* of v if $sw = \mathrm{MinSuf}(v, w)$ for some $w \in \bar{\Sigma}^*$.

Let $v = v_1^{p_1} \dots v_m^{p_m}$ be the Lyndon factorization of a string $v \in \Sigma^+$. For $1 \le j \le m$ we denote $s_j = v_j^{p_j} \cdots v_m^{p_m}$; moreover, we assume $s_{m+1} = \varepsilon$. Let λ be the smallest index such that s_{i+1} is a prefix of v_i for $\lambda \le i \le m$. Observe that $s_\lambda \succ \ldots \succ s_m \succ s_{m+1} = \varepsilon$, since v_i is a prefix of s_i. We define y_i so that $v_i = s_{i+1}y_i$, and we set $x_i = y_i s_{i+1}$. Note that $s_i = v_i^{p_i} s_{i+1} = (s_{i+1}y_i)^{p_i} s_{i+1} = s_{i+1}(y_i s_{i+1})^{p_i} = s_{i+1}x_i^{p_i}$. We also denote $\Lambda(w) = \{s_\lambda, \dots, s_m, s_{m+1}\}$, $X(w) = \{x_\lambda^\infty, \dots, x_m^\infty\}$, and $X'(w) = \{x_\lambda^{p_\lambda}, \dots, x_m^{p_m}\}$. The observation below lists several immediate properties of the introduced strings:

▶ **Observation 6.** *For each i, $\lambda \le i \le m$: (a) $x_i^\infty \succ x_i^{p_i} \succeq x_i \succeq y_i$, (b) $x_i^{p_i}$ is a suffix of v of length $|s_i| - |s_{i+1}|$, and (c) $|s_i| > 2|s_{i+1}|$. In particular, $|\Lambda(v)| = \mathcal{O}(\log |v|)$.*

The following lemma shows that $\Lambda(v)$ is equal to the set of significant suffixes of v. (Significant suffixes are actually defined in [20] as $\Lambda(v)$ and only later proved to satisfy our Definition 5.) In fact, the lemma is much deeper; in particular, the formula for $\mathrm{MaxSuf}(v, w)$ is one of the key ingredients of our efficient algorithms answering Minimal Suffix Queries.

▶ **Lemma 7** (I et al. [20], Lemmas 12–14)**.** *For a string $v \in \Sigma^+$ let s_i, λ, x_i, and y_i, be defined as above. Then $x_\lambda^\infty \succ x_\lambda^{p_\lambda} \succeq y_\lambda \succ x_{\lambda+1}^\infty \succ x_{\lambda+1}^{p_{\lambda+1}} \succeq y_{\lambda+1} \succ \ldots \succ x_m^\infty \succ x_m^{p_m} \succeq y_m$. Moreover, for every string $w \in \bar{\Sigma}^*$ we have*

$$\mathrm{MinSuf}(v, w) = \begin{cases} s_\lambda w & \text{if } w \succ x_\lambda^\infty, \\ s_i w & \text{if } x_{i-1}^\infty \succ w \succ x_i^\infty \text{ for } \lambda < i \le m, \\ s_{m+1} w & \text{if } x_m^\infty \succ w. \end{cases}$$

In other words, $\mathrm{MinSuf}(v, w) = s_{m+1-r} w$ where $r = \mathrm{rank}_{X(v)}(w)$.

We conclude this section with a precise characterization of $\Lambda(uv)$ for $|u| \le |v|$ in terms of $\Lambda(v)$ and $\mathrm{MaxSuf}^R(u, v)$. This is another key ingredient of our data structure, in particular letting us efficiently compute significant suffixes of a given fragment of T. The proof is deferred to the full version due to space constraints.

▶ **Lemma 8.** *Let $u, v \in \Sigma^+$ be strings such that $|u| \le |v|$. Also, let $\Lambda(v) = \{s_\lambda, \dots, s_{m+1}\}$, $s' = \mathrm{MaxSuf}^R(u, v)$, and let s_i be the longest suffix in $\Lambda(v)$ which is a prefix of s'. Then*

$$\Lambda(uv) = \begin{cases} \{s_\lambda, \dots, s_{m+1}\} & \text{if } s' \preceq^R s_\lambda \text{ (i.e., if } s_\lambda \preceq s' \text{ and } i \ne \lambda), \\ \{s', s_{i+1}, \dots, s_{m+1}\} & \text{if } s' \succ^R s_\lambda,\ i \le m, \text{ and } |s_i| - |s_{i+1}| \text{ is a period of } s', \\ \{s', s_i, s_{i+1}, \dots, s_{m+1}\} & \text{otherwise.} \end{cases}$$

Consequently, for every $w \in \bar{\Sigma}^*$, we have $\mathrm{MinSuf}(uv, w) \in \{\mathrm{MaxSuf}^R(u, v)w, \mathrm{MinSuf}(v, w)\}$.

4 Answering Minimal Suffix Queries

In this section we present our data structure for Minimal Suffix Queries. We proceed in three steps improving the query time from $\mathcal{O}(\log |v|)$ via $\mathcal{O}(\log^* |v|)$ to $\mathcal{O}(1)$. The first solution is an immediate application of Observation 4.3. and the notion of significant suffixes. Efficient computation of these suffixes, also used in the construction of further versions of our data structure, is based on Lemma 8, which yields a recursive procedure. The only "new" suffix needed at each step is determined using the following result. It can be seen as a cleaner formulation of Lemma 14 in [4].

▶ **Lemma 9.** *Let $u = T[\ell..r]$ and $v = T[r+1..r']$ be fragments of T such that $|u| \le |v|$. Using the augmented suffix array of T we can compute $\mathrm{MaxSuf}^R(u, v)$ in $\mathcal{O}(1)$ time.*

▶ **Lemma 10.** *Given a fragment v of T, we can compute $\Lambda(v)$ in $\mathcal{O}(\log|v|)$ time using the augmented suffix array of T*

Proof. If $|v| = 1$, we return $\Lambda(v) = \{v, \varepsilon\}$. Otherwise, we decompose $v = uv'$ so that $|v'| = \lceil \frac{1}{2}|v| \rceil$. We recursively generate $\Lambda(v')$ and use Lemma 9 to compute $s = \mathrm{MaxSuf}^R(u, v')$. Then, we apply the characterization of Lemma 8 to determine $\Lambda(v) = \Lambda(uv')$, using the augmented suffix array (Theorem 1) to lexicographically compare fragments of T.

We store the lengths of the significant suffixes in an ordered list. This way we can implement a single phase (excluding the recursive calls) in time proportional to $\mathcal{O}(1)$ plus the number of suffixes removed from $\Lambda(v')$ to obtain $\Lambda(v)$. Since this is amortized constant time, the total running time becomes $\mathcal{O}(\log|v|)$ as announced. ◀

▶ **Corollary 11.** Minimal Suffix Queries *can be answered in $\mathcal{O}(\log|v|)$ time using the augmented suffix array of T.*

Proof. Recall that Observation 4.3. yields $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v[1..m-1], v[m])$ where $m = |v|$. Consequently, $\mathrm{MinSuf}(v) = sv[m]$ for some $s \in \Lambda(v[1..m-1])$. We apply Lemma 10 to compute $\Lambda(v[1..m-1])$ and determine the answer among $\mathcal{O}(\log|v|)$ candidates using lexicographic comparison of fragments, provided by the augmented suffix array (Theorem 1). ◀

4.1 $\mathcal{O}(\log^*|v|)$-time Minimal Suffix Queries

An alternative $\mathcal{O}(\log|v|)$-time algorithm could be developed based just on the second part of Lemma 8: decompose $v = uv'$ so that $|v'| > |u|$ and return $\min(\mathrm{MaxSuf}^R(u, v'), \mathrm{MinSuf}(v'))$. The result is $\mathrm{MinSuf}(v)$ due to Lemma 8 and Observation 4.3. Here, the first candidate $\mathrm{MaxSuf}^R(u, v')$ is determined via Lemma 9, while the second one using a recursive call. A way to improve query time to $\mathcal{O}(1)$ at the price of $\mathcal{O}(n \log n)$-time preprocessing is to precompute the answers for *basic* fragments, i.e., fragments whose length is a power of two. Then, in order to determine $\mathrm{MinSuf}(v)$, we perform just a single step of the aforementioned procedure, making sure that v' is a basic fragment. Both these ideas are actually present in [4], along with a smooth trade-off between their preprocessing and query times.

Our $\mathcal{O}(\log^*|v|)$-time query algorithm combines recursion with preprocessing for certain *distinguished* fragments. More precisely, we say that $v = T[\ell..r]$ is distinguished if both $|v| = 2^q$ and $f(2^q) \mid r$ for some positive integer q, where $f(x) = 2^{\lfloor \log\log x \rfloor^2}$. Note that the number of distinguished fragments of length 2^q is at most $\frac{n}{2^{\lfloor \log q \rfloor^2}} = \mathcal{O}(\frac{n}{q^{\omega(1)}})$.

The query algorithm is based on the following decomposition ($x > f(x)$ for $x > 2^{16}$):

▶ **Fact 12.** *Given a fragment v such that $|v| > f(|v|)$, we can in constant time decompose $v = uv'v''$ such that $1 \le |v''| \le f(|v|)$, v' is distinguished, and $|u| \le |v'|$.*

Proof. Let $v = T[\ell..r]$, $q = \lfloor \log|v| \rfloor$ and $q' = \lfloor \log q \rfloor^2$. We determine r' as the largest integer strictly smaller than r divisible by $2^{q'} = f(|v|)$. By the assumption that $|v| > 2^{q'}$, we conclude that $r' \ge r - 2^{q'} \ge \ell$. We define $v'' = T[r'+1..r]$ and partition $T[\ell..r'] = uv'$ so that $|v'|$ is the largest possible power of two. This guarantees $|u| \le |v'|$. Moreover, $|v'| \le |v|$ assures that $f(|v'|) \mid f(|v|)$, so $f(|v'|) \mid r'$, and therefore v' is indeed distinguished. ◀

Observation 4.2. implies $\mathrm{MinSuf}(v) \in \{\mathrm{MinSuf}(uv', v''), \mathrm{MinSuf}(v'')\}$ and Lemma 8 further yields $\mathrm{MinSuf}(v) \in \{\mathrm{MaxSuf}^R(u, v')v'', \mathrm{MinSuf}(v', v''), \mathrm{MinSuf}(v'')\}$, i.e., leaves us with three candidates for $\mathrm{MinSuf}(v)$. Our query algorithm obtains $\mathrm{MaxSuf}^R(u, v')$ using Lemma 9,

computes $\mathrm{MinSuf}(v'')$ recursively, and determines $\mathrm{MinSuf}(v', v'')$ through the characterization of Lemma 7. The latter step is performed using the following component based on a fusion tree, which we build for all distinguished fragments.

▶ **Lemma 13.** *Let $v = T[\ell..r]$ be a fragment of T. There exists a data structure of size $\mathcal{O}(\log|v|)$ which answers the following queries in $\mathcal{O}(1)$ time: given a position $r' > r$ compute $\mathrm{MinSuf}(v, T[r+1..r'])$. Moreover, this data structure can be constructed in $\mathcal{O}(\log|v|)$ time using the augmented suffix array of T.*

Proof. By Lemma 7, we have $\mathrm{MinSuf}(v, w) = s_{m+1-\mathrm{rank}_{X(v)}(w)}w$, so in order to determine $\mathrm{MinSuf}(v, T[r+1..r'])$, it suffices to store $\Lambda(v)$ and efficiently compute $\mathrm{rank}_{X(v)}(w)$ given $w = T[r+1..r']$. We shall reduce these rank queries to rank queries in an integer set $R(v)$.

▶ **Claim.** *Denote $X(v) = \{x^\infty_\lambda, \ldots, x^\infty_m\}$ and let*

$$R(v) = \{r + \mathrm{lcp}(T[r+1..], x^\infty_j) : x^\infty_j \in X(w) \wedge x^\infty_j \prec T[r+1..]\}.$$

For every index r', $r < r' \le n$, we have $\mathrm{rank}_{X(v)}(T[r+1..r']) = \mathrm{rank}_{R(v)}(r')$.

Proof. We shall prove that for each j, $\lambda \le j \le m$, we have

$$x^\infty_j \prec T[r+1..r'] \iff \big(r + \mathrm{lcp}(T[r+1..], x^\infty_j) < r' \ \wedge\ x^\infty_j \prec T[r+1..]\big).$$

First, if $x^\infty_j \succ T[r+1..]$, then clearly $x^\infty_j \succ T[r+1..r']$ and both sides of the equivalence are false. Therefore, we may assume $x^\infty_j \prec T[r+1..]$. Observe that in this case $d := \mathrm{lcp}(T[r+1..], x^\infty_j)$ is strictly less than $n - r$, and $T[r+1..r+d] \prec x^\infty_j \prec T[r+1..r+d+1]$. Hence, $x^\infty_j \prec T[r+1..r']$ if and only if $r + d < r'$, as claimed. ◀

We apply Theorem 2 to build a fusion tree for $R(v)$, so that the ranks are can be obtained in $\mathcal{O}(1 + \frac{\log|R(v)|}{\log W})$ time, which is $\mathcal{O}(1 + \frac{\log\log|v|}{\log\log n}) = \mathcal{O}(1)$ by Observation 6.

The construction algorithm uses Lemma 10 to compute $\Lambda(v) = \{s_\lambda, \ldots, s_{m+1}\}$. Next, for each j, $\lambda \le j \le m$, we need to determine $\mathrm{lcp}(T[r+1..], x^\infty_j)$. This is the same as $\mathrm{lcp}(T[r+1..], (x^{p_j}_j)^\infty)$ and, by Observation 6, $x^{p_j}_j$ can be retrieved as the suffix of v of length $|s_i| - |s_{i+1}|$. Hence, the augmented suffix array can be used to compute these longest common prefixes and therefore to construct $R(v)$ in $\mathcal{O}(|\Lambda(v)|) = \mathcal{O}(\log|v|)$ time. ◀

With this central component we are ready to give a full description of our data structure.

▶ **Theorem 14.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which answers* Minimal Suffix Queries *in $\mathcal{O}(\log^*|v|)$ time and can be constructed in $\mathcal{O}(n)$ time.*

Proof. Our data structure consists of the augmented suffix array (Theorem 1) and the components of Lemma 13 for all distinguished fragments of T. Each such fragment of length 2^q contributes $\mathcal{O}(q)$ to the space consumption and to the construction time, which in total over all lengths sums up to $\mathcal{O}(\sum_q \frac{nq}{q^{\omega(1)}}) = \mathcal{O}(\sum_q \frac{n}{q^{\omega(1)}}) = \mathcal{O}(n)$.

Let us proceed to the query algorithm. Assume we are to compute the minimal suffix of a fragment v. If $|v| \le f(|v|)$ (i.e., if $|v| \le 2^{16}$), we use the logarithmic-time query algorithm given in Corollary 11. If $|v| > 2^q$, we apply Fact 12 to determine a decomposition $v = uv'v''$, which gives us three candidates for $\mathrm{MinSuf}(v)$. As already described, $\mathrm{MinSuf}(v'')$ is computed recursively, $\mathrm{MinSuf}(v', v'')$ using Lemma 13, and $\mathrm{MaxSuf}^R(u, v')v''$ using Lemma 9. The latter two both support constant-time queries, so the overall time complexity is proportional to the depth of the recursion. We have $|v''| \le f(|v|) < |v|$, so it terminates. Moreover,

$$f(f(x)) = 2^{\lfloor\log(\log f(x))\rfloor^2} \le 2^{(\log(\log\log x)^2)^2} = 2^{4(\log\log\log x)^2} = 2^{o(\log\log x)} = o(\log x).$$

Thus, $f(f(x)) \leq \log x$ unless $x = \mathcal{O}(1)$. Consequently, unless $|v| = \mathcal{O}(1)$, when the algorithm clearly needs constant time, the length of the queried fragment is in two steps reduced from $|v|$ to at most $\log |v|$. This concludes the proof that the query time is $\mathcal{O}(\log^* |v|)$. ◀

4.2 $\mathcal{O}(1)$-time Minimal Suffix Queries

The $\mathcal{O}(\log^* |v|)$ time complexity of the query algorithm of Theorem 14 is only due to the recursion, which in a single step reduces the length of the queried fragment from $|v|$ to $f(|v|)$ where $f(x) = 2^{\lfloor \log \log x \rfloor^2}$. Since $f(f(x)) = 2^{o(\log \log x)}$, after just two steps the fragment length does not exceed $f(f(n)) = o(\frac{\log n}{\log \log n})$. In this section we show that the minimal suffixes of such short fragments can precomputed in a certain sense, and thus after reaching $\tau = f(f(n))$ we do not need to perform further recursive calls.

For constant alphabets, we could actually store all the answers for all $\mathcal{O}(\sigma^\tau) = n^{o(1)}$ strings of length up to τ. Nevertheless, in general all letters of T, and consequently all fragments of T, could even be distinct. However, the answers to Minimal Suffix Queries actually depend only on the relative order between letters, which is captured by order-isomorphism.

Two strings x and y are called *order-isomorphic* [25, 23], denoted as $x \approx y$, if $|x| = |y|$ and for every two positions i, j $(1 \leq i, j \leq |x|)$ we have $x[i] \prec x[j] \iff y[i] \prec y[j]$. Note that the equivalence extends to arbitrary corresponding fragments of x and y, i.e., $x[i..j] \prec x[i'..j'] \iff y[i..j] \prec y[i'..j']$. Consequently, order-isomorphic strings cannot be distinguished using Minimal Suffix Queries or Generalized Minimal Suffix Queries.

Moreover, observe that every string of length m is order-isomorphic to a string over an alphabet $\{1, \ldots, m\}$. Consequently, order-isomorphism partitions strings of length up to m into $\mathcal{O}(m^m)$ equivalence classes. The following fact lets us compute canonical representations of strings whose length is bounded by $m = W^{\mathcal{O}(1)}$.

▶ **Fact 15.** *For every fixed integer $m = W^{\mathcal{O}(1)}$, there exists a function* oid *mapping each string w of length up to m to a non-negative integer* $\mathrm{oid}(w)$ *with $\mathcal{O}(m \log m)$ bits, so that $w \approx w' \iff \mathrm{oid}(w) = \mathrm{oid}(w')$. Moreover, the function can be evaluated in $\mathcal{O}(m)$ time.*

Proof. To compute $\mathrm{oid}(w)$, we first build a fusion tree storing all (distinct) letters which occur in w. Next, we replace each character of w with its rank among these letters. We allocate $\lceil \log m \rceil$ bits per character and prepend such a representation with $\lceil \log m \rceil$ bits encoding $|w|$. This way $\mathrm{oid}(w)$ is a sequence of $(|w| + 1) \lceil \log m \rceil = \mathcal{O}(m \log m)$ bits. Using Theorem 2 to build the fusion tree, we obtain an $\mathcal{O}(m)$-time evaluation algorithm. ◀

To answer queries for short fragments of T, we define overlapping *blocks* of length $m = 2\tau$: for $0 \leq i \leq \frac{n}{\tau}$ we create a block $T_i = T[1 + i\tau .. \min(n, (i+2)\tau)]$. For each block we apply Fact 15 to compute the identifier $\mathrm{oid}(T_i)$. The total length of the blocks is bounded $2n$, so this takes $\mathcal{O}(n)$ time. The identifiers use $\mathcal{O}(\frac{n}{\tau} \tau \log \tau) = O(n \log \tau)$ bits of space.

Moreover, for each distinct identifier $\mathrm{oid}(T_i)$, we store the answers to all the Minimal Suffix Queries in T_i. This takes $\mathcal{O}(\log m)$ bits per answer and $\mathcal{O}(2^{\mathcal{O}(m \log m)} m^2 \log m) = 2^{\mathcal{O}(\tau \log \tau)}$ in total. Since $\tau = o(\frac{\log n}{\log \log n})$, this is $n^{o(1)}$. The preprocessing time is also $n^{o(1)}$.

It is a matter of simple arithmetics to extend a given fragment v of T, $|v| \leq \tau$, to a block T_i. We use the precomputed answers stored for $\mathrm{oid}(T_i)$ to determine the minimal suffix of v. We only need to translate the indices within T_i to indices within T before returning the answer. Below, we state our results for short and arbitrary fragments, respectively:

▶ **Theorem 16.** *For every text T of length n and every parameter $\tau = o(\frac{\log n}{\log \log n})$ there exists a data structure of size $\mathcal{O}(\frac{n \log \tau}{\log n})$ which can answer in $\mathcal{O}(1)$ time* Minimal Suffix Queries *for fragments of length not exceeding τ. Moreover, it can be constructed in $\mathcal{O}(n)$ time.*

▶ **Theorem 17.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which can be constructed in $\mathcal{O}(n)$ time and answers* Minimal Suffix Queries *in $\mathcal{O}(1)$ time.*

5 Answering Generalized Minimal Suffix Queries: Overview

In this section we sketch our solution for Generalized Minimal Suffix Queries, focusing on the differences compared to the data structure developed in Section 4. As in Section 4, we proceed in three steps gradually improving the query time; we start, however, with some terminology.

We define a *k-fragment* of a text T as a concatenation $T[\ell_1..r_1]\cdots T[\ell_k..r_k]$ of k fragments of the text T. Observe that a k-fragment can be stored in $\mathcal{O}(k)$ space as a sequence of pairs (ℓ_i, r_i). If a string w admits such a decomposition using k' ($k' \le k$) substrings, we call it a *k-substring* of T. Every k'-fragment (with $k' \le k$) whose value is equal to w is called an *occurrence* of w as a k-substring of T. Observe that a substring of a k-substring w of T is itself a k-substring of T. Moreover, given an occurrence of w, one can canonically assign each fragment of w to a k'-fragment of T ($k' \le k$). This can be implemented in $\mathcal{O}(k)$ time and referring to $w[\ell..r]$ in our algorithms, we assume that such an operation is performed.

Basic queries regarding k-fragments easily reduce to their counterparts for 1-fragments:

▶ **Observation 18.** *The augmented suffix array can answer queries 1., 2., and 3. in $\mathcal{O}(k)$ time if x and y are k-fragments of T.*

Generalized Minimal Suffix Queries can be reduced to the following auxiliary queries:

▶ **Problem** (Auxiliary Minimal Suffix Queries)**.** *Given a fragment v of T and a k-fragment w of T, compute* $\mathrm{MinSuf}(v, w)$ *(represented as a $(k+1)$-fragment of T).*

▶ **Lemma 19.** *For every text T, the minimal suffix of a k-fragment v can be determined by k* Auxiliary Minimal Suffix Queries *(with $k' < k$) and additional $\mathcal{O}(k^2)$-time processing using the augmented suffix array of T.*

Proof. Let $v = v_1 \cdots v_k$. By Observation 4.2., $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v_k)$ or for some i, $1 \le i < k$, we have $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v_i, v_{i+1} \cdots v_k)$. Hence, we apply Auxiliary Minimal Suffix Queries to determine $\mathrm{MinSuf}(v_i, v_{i+1} \cdots v_k)$ for each $1 \le i < k$. Observation 4.3. lets reduce computing $\mathrm{MinSuf}(v_k)$ to another auxiliary query. Having obtained k candidates for $\mathrm{MinSuf}(v)$, we use the augmented suffix array to return the smallest among them using $k-1$ comparisons, each performed in $\mathcal{O}(k)$ time; see Theorem 1 and Observation 18. ◀

Below we focus on the auxiliary queries only. Answering them in $\mathcal{O}(k \log |v|)$ time is easy: We apply Lemma 10 to determine $\Lambda(v)$, and then we compute the smallest string among $\{sw : s \in \Lambda(v)\}$. These strings are $(k+1)$-fragments of T and thus a single comparison takes $\mathcal{O}(k)$ time using the augmented suffix array.

5.1 $\mathcal{O}(k \log^* |v|)$-time Auxiliary Minimal Suffix Queries

Our solution is based on that in Section 4.1. The only big challenge is to generalize Lemma 13: preprocess v to compute $\mathrm{MinSuf}(v, w)$ for an arbitrary k-fragment w in $\mathcal{O}(k)$ time. We still apply Lemma 7, but this time we actually determine $\mathrm{rank}_{X'(v)}(w)$, which differs from $\mathrm{rank}_{X(v)}(w)$ by at most one (and therefore leaves us with two candidates for $\mathrm{rank}_{X(v)}(w)$).

This is because in general we are able to preprocess a family A of fragments of T to determine $\mathrm{rank}_A(w)$ given a k-fragment w of T. Our solution is based on the compressed trie

of fragments in A, accompanied with several fusion trees to allow efficient navigation. For $|A| \leq W^{\mathcal{O}(1)}$ it takes $\mathcal{O}(|A|^2)$ time to construct and determines ranks in the optimal time $\mathcal{O}(k)$. By our choice of distinguished fragments v of length 2^q, building this component for all sets $X'(v)$ takes $\mathcal{O}(\frac{nq^2}{q^{\omega(1)}}) = \mathcal{O}(\frac{n}{q^{\omega(1)}})$ time, which is $\mathcal{O}(n)$ in total (over all values of q).

5.2 $\mathcal{O}(k)$-time Auxiliary Minimal Suffix Queries

To achieve the optimal query time, we again focus on $|v| \leq \tau$ with $\tau = o(\frac{\log n}{\log \log n})$. Computing $\mathrm{MinSuf}(v, w)$, we need to handle k-fragments w of arbitrary length, which might be scattered around the text T (not just in a block T_i containing v), so the task is much more difficult than in Section 4.2. Our approach is to replace w with a similar k'-fragment w' of $T_i\$$, such that $k' \leq k+1$ and $\mathrm{rank}_{X'(v)}(w) = \mathrm{rank}_{X'(v)}(w')$. This is achieved again using fusion trees.

As already noted, a fixed value of $\mathrm{rank}_{X'(v)}(w)$ gives two candidates for $\mathrm{rank}_{X(v)}(w)$, i.e., for $\mathrm{MinSuf}(v, w)$. Simultaneously $\mathrm{rank}_{X'(v)}(w')$ depends only on the relative order of letters of $T_i\$$. Hence, for each distinct $\mathrm{oid}(T_i)$ and for each fragment v of $T_i\$$, we construct $\Lambda(v)$ and a data structure able to efficiently rank k-fragments of $T_i\$$ in $X'(v)$. This component is built using the general tool for ranking k-fragments in a collection of fragments, which we mentioned in Section 5.1. This ultimately leads to the strongest result of this paper:

▶ **Theorem 20.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which can be constructed in $\mathcal{O}(n)$ time and answers* Generalized Minimal Suffix Queries *in $\mathcal{O}(k^2)$ time.*

6 Applications

As already noted in [4], Minimal Suffix Queries can be used to compute Lyndon factorization. For fragments of T, and in general $k = \mathcal{O}(1)$, we obtain an optimal solution:

▶ **Corollary 21.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which given a k-fragment v of T determines the Lyndon factorization $v = v_1^{q_1} \dots v_m^{q_m}$ in $\mathcal{O}(k^2 m)$ time. The data structure takes $\mathcal{O}(n)$ time to construct.*

Our main motivation of introducing Generalized Minimal Suffix Queries, however, was to answer Minimal Rotation Queries, for which we obtain constant query time after linear-time preprocessing. This is achieved using the following observation; see [10]:

▶ **Observation 22.** *The minimal cyclic rotation of v is the prefix of $\mathrm{MinSuf}(v, v)$ of length $|v|$.*

▶ **Theorem 23.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which given a k-fragment v of T determines the lexicographically smallest cyclic rotation of v in $\mathcal{O}(k^2)$ time. The data structure takes $\mathcal{O}(n)$ time to construct.*

Using Minimal Rotation Queries, we can compute the Karp-Rabin fingerprint [21] of the minimal rotations of a given fragment v of T (or in general, of a k-fragment). This can be interpreted as computing fingerprints up to cyclic equivalence, i.e., evaluating a function h such that $h(\ell, r) = h(\ell', r')$ if and only if $T[\ell..r]$ and $T[\ell'..r']$ are cyclically equivalent.

Consequently, we are able, for example, to count distinct substrings of T with a given exponent $1 + 1/\alpha$. They occur within runs or α-gapped repeats, which can be generated in time $\mathcal{O}(n\alpha)$ [6, 12, 16] and classified using Minimal Rotation Queries according to the cyclic equivalence class of their period. For a fixed equivalence class the set of substrings generated by a single repeat can be represented as a cyclic interval, and the cardinality of a union of intervals is simple to determine; see also [11], where this approach was used to count and list squares and, in general, substrings with a given exponent 2 or more.

Acknowledgements I would like to thank the remaining co-authors of [4], collaboration with whom on earlier results about minimal and maximal suffixes sparked some of my ideas used in this paper. Special acknowledgments to Paweł Gawrychowski and Tatiana Starikovskaya for numerous discussions on this subject.

References

1 Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range LCP. *J. Comput. Syst. Sci.*, 80(7):1245–1253, 2014. doi:10.1016/j.jcss.2014.02.010.

2 Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. Range LCP queries revisited. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval, SPIRE 2015*, volume 9309 of *LNCS*, pages 350–361. Springer, 2015. doi:10.1007/978-3-319-23826-5.

3 Alberto Apostolico and Maxime Crochemore. Optimal canonization of all substrings of a string. *Inf. Comput.*, 95(1):76–95, 1991. doi:10.1016/0890-5401(91)90016-U.

4 Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, Ignat Kolesnichenko, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theor. Comput. Sci.*, 2015. In press. doi:10.1016/j.tcs.2015.08.023.

5 Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.

6 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem, 2015. arXiv:1406.0263v7.

7 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *Latin American Symposium on Theoretical Informatics, LATIN 2000*, volume 1776 of *LNCS*, pages 88–94. Springer Berlin Heidelberg, 2000. doi:10.1007/10719839_9.

8 Kuo Tsai Chen, Ralph Hartzler Fox, and Roger Conant Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Ann. Math.*, 68(1):81–95, 1958. doi:10.2307/1970044.

9 Graham Cormode and S. Muthukrishnan. Substring compression problems. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 321–330. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070478.

10 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

11 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theor. Comput. Sci.*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

12 Maxime Crochemore, Roman Kolpakov, and Gregory Kucherov. Optimal bounds for computing α-gapped repeats. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications, LATA 2016*, volume 9618 of *LNCS*, pages 245–255. Springer, 2016. doi:10.1007/978-3-319-30000-9_19.

13 Maxime Crochemore and Dominique Perrin. Two-way string-matching. *J. ACM*, 38(3):650–674, July 1991. doi:10.1145/116825.116845.

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

15 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

16 Paweł Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Efficiently finding all maximal α-gapped repeats. In Nicolas Ollinger and Heribert Vollmer, editors, *Symposium on Theoretical Aspects of Computer Science, STACS 2016*, volume 47 of *LIPIcs*, pages 39:1–39:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.STACS.2016.39.

17 Torben Hagerup. Sorting and searching on the word RAM. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *Symposium on Theoretical Aspects of Computer Science, STACS 1998*, volume 1373 of *LNCS*, pages 366–398. Springer, Berlin Heidelberg, 1998. doi:10.1007/BFb0028575.

18 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.

19 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient Lyndon factorization of grammar compressed text. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching, CPM 2013*, volume 7922 of *LNCS*, pages 153–164. Springer, 2013. doi:10.1007/978-3-642-38905-4.

20 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *String Processing and Information Retrieval, SPIRE 2013*, volume 8214 of *LNCS*, pages 174–185. Springer International Publishing Switzerland, 2013. doi:10.1007/978-3-319-02432-5_21.

21 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.

22 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:45–54, 2014. doi:10.1016/j.tcs.2013.10.010.

23 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014. doi:10.1016/j.tcs.2013.10.006.

24 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

25 Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013. doi:10.1016/j.ipl.2013.03.015.

26 Roger Conant Lyndon. On Burnside's problem. *T. Am. Math. Soc.*, 77(2):202–215, 1954. doi:10.1090/S0002-9947-1954-0064049-X.

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

28 Marcin Mucha. Lyndon words and short superstrings. In Sanjeev Khanna, editor, *24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 958–972. SIAM, 2013. doi:10.1137/1.9781611973105.

29 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 166–175. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.26.

Optimal Prefix Free Codes with Partial Sorting*

Jérémy Barbay

Departamento de Ciencias de la Computación (DCC), Universidad de Chile, Santiago, Chile
jeremy@barbay.cl

Abstract

We describe an algorithm computing an optimal prefix free code for n unsorted positive weights in less time than required to sort them on many large classes of instances, identified by a new measure of difficulty for this problem, the alternation α. This asymptotical complexity is within a constant factor of the optimal in the algebraic decision tree computational model, in the worst case over all instances of fixed size n and alternation α. Such results refine the state of the art complexity in the worst case over instances of size n in the same computational model, a landmark in compression and coding since 1952, by the mere combination of van Leeuwen's algorithm to compute optimal prefix free codes from sorted weights (known since 1976), with Deferred Data Structures to partially sort multisets (known since 1988).

1998 ACM Subject Classification F.2.2 Analysis of Algorithms and Problem Complexitity / Nonnumerical Algorithms and Problems (Sorting and Searching), E.4 Coding and Information Theory (Data compaction and compression)

Keywords and phrases Deferred Data Structure, Huffman, Median, Optimal Prefix Free Codes, van Leeuwen

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.29

1 Introduction

Given n positive weights $W[1..n]$ coding for the frequencies $\left\{W[i]/\sum_{j=1}^{n} W[j]\right\}_{i\in[1..n]}$ of n messages, and a number D of output symbols, an Optimal Prefix Free Code [11] is a set of n code strings on alphabet $[1..D]$, of variable lengths $L[1..n]$ and such that no string is prefix of another, and the average length of a code is minimized (i.e. $\sum_{i=1}^{n} L[i]W[i]$ is minimal).

Any prefix free code can be computed in linear time from a set of code lengths satisfying the Kraft inequality $\sum_{i=1}^{n} D^{-L[i]} \leq 1$. The original description of the code by Huffman [11] yields a heap-based algorithm performing $O(n \log n)$ algebraic operations, using the bijection between D-ary prefix free codes and D-ary cardinal trees [8]. This complexity is asymptotically optimal for any constant value of D in the algebraic decision tree computational model[1], in the worst case over instances composed of n positive weights, as computing the optimal binary prefix free code for the weights $W[0,\ldots,Dn] = \{D^{x_1},\ldots,D^{x_1},D^{x_2},\ldots,D^{x_2},\ldots,D^{x_n},\ldots,D^{x_n}\}$ is equivalent to sorting the positive integers $\{x_1,\ldots,x_n\}$. We consider here only the binary case, where $D = 2$.

Yet, not all instances require the same amount of work to compute an optimal code:

* Extended abstract, see the full version [1] on http://arxiv.org/abs/1602.03934 for complete proofs and comments.

[1] The algebraic decision tree computational model is composed of algorithms which can be modelled as a decision tree where the decision made in each node is based only on algebraic operations on the input.

- When the weights are given in sorted order, van Leeuwen [14] showed that an optimal code can be computed using within $O(n)$ algebraic operations.
- When the weights consist of $r \in [1..n]$ distinct values and are given in a sorted, compressed form, Moffat and Turpin [17] showed how to compute an optimal code using within $O(r(1 + \log(n/r)))$ algebraic operations, which is often sublinear in n.
- In the case where the weights are given unsorted, Belal *et al.* [5, 6] described several families of instances for which an optimal prefix free code can be computed in linear time, along with an algorithm claimed to perform $O(kn)$ algebraic operations, in the worst case over instances formed by n weights such that there is an optimal binary prefix free code with k distinct code lengths[2]. This complexity was later downgraded to $O(16^k n)$ in an extended version[4] of their article. Both results are better than the state of the art when k is finite, but worse when k is larger than $\log n$.

In the context described above, various questions are left unanswered, from the confirmation of the existence of an algorithm running in time $O(16^k n)$ or $O(kn)$, to the existence of an algorithm taking advantage of small values of both n and k, less trivial than running two algorithms in parallel and stopping both whenever one computes the answer. Given n positive integer weights, *can we compute an optimal binary prefix free code in time better than* $O(\min\{kn, n \log n\})$ *in the algebraic decision tree computational model?* We answer in the affirmative for many classes of instances, identified by the alternation measure α defined in Section 3.1:

▶ **Theorem 1.** *Given n positive weights of alternation $\alpha \in [1..n-1]$, there is an algorithm which computes an optimal binary prefix free code using within $O(n(1+\log\alpha)) \subseteq O(n \lg n)$ algebraic instructions, and this complexity is asymptotically optimal among all algorithms in the algebraic decision tree computational model in the worst case over instances of size n and alternation α.*

Proof. We show in Lemma 12 that any algorithm A in the algebraic decision tree computational model performs within $\Omega(n \lg \alpha)$ algebraic operations in the worst case over instances of size n and alternation α. We show in Lemma 9 that the `GDM` algorithm, a variant of the van Leeuwen's algorithm [14], modified to use the deferred data structure from Lemma 5, performs $q \in O(\alpha(1+\lg \frac{n-1}{\alpha}))$ such queries, which yields in Corollary 10 a complexity within $O(n(1+\log\alpha) + \alpha(\lg n)(\lg \frac{n}{\alpha}))$, all within the algebraic decision tree computational model. As $\alpha \in [1..n-1]$ and $O(\alpha(\lg n)(\lg \frac{n}{\alpha})) \subseteq O(n(1+\log\alpha))$ for this range (Lemma 11), the optimality ensues. ◀

We discuss our solution in Section 2 in three parts: the intuition behind the general strategy in Section 2.1, the deferred data structure which maintains a partially sorted list of weights while supporting `rank`, `select` and `partialSum` queries in Section 2.2, and the algorithm which uses those operators to compute an optimal prefix free code in Section 2.3. Our main contribution consists in the analysis of the running time of this solution, described in Section 3: the formal definition of the parameter of the analysis in Section 3.1, the upper bound in Section 3.2 and the matching lower bound in Section 3.3. We conclude with a comparison of our results with those from Belal *et al.* [5] in Section 4.

[2] Note that k is not uniquely defined, as for a given set of weights there can exist several optimal prefix free codes varying in the number of distinct code lengths used.

2 Solution

The solution that we describe is a combination of two results: some results about deferred data structures for multisets, which support queries in a "lazy" way; and some results about the relation between the computational cost of sorting and that of computing an optimal prefix free code. We describe the general intuition of our solution in Section 2.1, the deferred data structure in Section 2.2, and the algorithm in Section 2.3.

2.1 General Intuition

The algorithm suggested by Huffman [11] starts with a heap of external nodes, selects the two nodes of minimal weight, pairs them into a new node which it adds to the heap, and iterates untill only one node is left. Whereas the type of the nodes selected, external or internal, does not matter in the analysis of the complexity of Huffman's algorithm, we claim that the computational cost of optimal prefix free codes can be greatly reduced on instances where many external nodes are selected consecutively. We define the "`EI` signature" of an instance as the first step toward the characterization of such instances:

▶ **Definition 2.** Given an instance of the optimal prefix free code problem formed by n positive weights $W[1..n]$, its *`EI` signature* $\mathcal{S}(W) \in \{E, I\}^{2n-1}$ is a string of length $2n-1$ over the alphabet $\{E, I\}$ (where E stands for "External" and I for "Internal") marking, at each step of the algorithm suggested by Huffman [11], whether an external or internal node is chosen as the minimum (including the last node returned by the algorithm, for simplicity).

The analysis described in Section 3 is based on the number $|S|_{EI}$ of blocks formed only of E in the `EI` signature of the instance S. We can already show some basic properties of this measure:

▶ **Lemma 3.** *Given the `EI` signature S of n unsorted positive weights $W[1..n]$, $|S|_E = n$; $|S|_I = n-1$; $|S| = 2n-1$; S starts with two E; S finishes with one I; $|S|_{EI} = |S|_{IE} + 1$; $|S|_{EI} \in [1..n-1]$.*

Proof. The three first properties are simple consequences of basic properties on binary trees. S starts with two E as the first two nodes paired are always external. S finishes with one I as the last node returned is always (for $n > 1$) an internal node. The two last properties are simple consequences of the fact that S is a binary string starting with an E and finishing with an I. ◀

For example, the text $T =$ "ABBCCCDDDDEEEEEFFFFFGGGGGGHHHHHHH" has frequencies $W =$ | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 7 |. It corresponds to an instance of size $n = 8$, of *`EI` signature* $\mathcal{S}(W) =$ `EEEIEEEEIEIIIII` of length 15, which starts with EE, finishes with I, and contains only $\alpha = 3$ occurrences of EI, corresponding to a decomposition into $\alpha = 3$ maximal blocks of consecutive Es.

Instances such as this, with very few blocks of E, are easier to solve than instances with many such blocks. For example, an instance W of length n such that its `EI` signature $\mathcal{S}(W)$ is composed of a single run of n Es followed by a single run of $n-1$ Is (such as the one described in Figure 1) can be solved in linear time, and in particular without sorting the weights: it is enough to assign the codelength $l = \lfloor \log_2 n \rfloor$ to the $n - 2^l$ largest weights and the codelength $l+1$ to the 2^l smallest weights. Separating those weights is a simple `select` operation, supported by the data structures described in the following section.

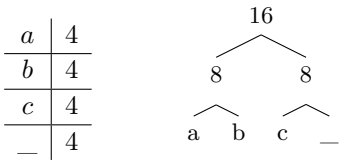


Figure 1 Frequencies and code tree for the text $T =$ "ba_bb_caca_ba_cc", minimizing the number of occurrences of "EI" in its `EI` signature $\mathcal{S}(T) =$ "EEEEIII".

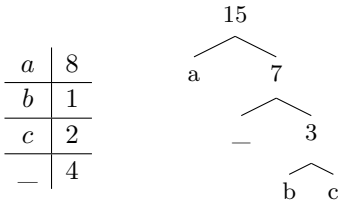


Figure 2 Frequencies and code tree for the text $T =$ "aaaaaaaabcc_____", maximizing the number of occurrences of "EI" in its `EI` signature $\mathcal{S}(T) =$ "EEIEIEI".

We describe two extreme examples. First, consider the text $T =$ "`ba_bb_caca_ba_cc`". Each of the four symbols of its alphabet $\{a, b, c, \_\}$ occurs exactly 4 times, so that an optimal prefix free code assigns a uniform codelength of 2 bits to all symbols (see Figure 1). There is no need to sort the symbols by frequency (and the prefix free code does not yield any information about the order in which the symbols would be sorted by monotone frequencies), and accordingly the `EI` signature of this text, $\mathcal{S}(T) =$ "EEEEIII", has a single block of Es, indicating a very easy instance. The same holds if the text is such that the frequencies of the symbols are all within a factor of two of each other. On the other hand, consider the text $T =$ "aaaaaaaabcc_____", where the frequencies of its symbols follow an exponential distribution, so that an optimal prefix free code assigns different codelengths to almost all symbols (see Figure 2). The prefix free code does yield a lot of information about the order in which the symbols would be sorted by monotone frequencies, and accordingly the `EI` signature of this text, $\mathcal{S}(T) =$ "EEIEIEI", has three blocks of Es, indicating a more difficult instance. The same holds with more general distribution, as long as no two pairs of symbol frequencies are within a factor of two of each other.

2.2 Partial Sum Deferred Data Structure

Given a MULTISET $W[1..n]$ on alphabet $[1..\sigma]$ of size n, Karp *et al.* [13] defined the first deferred data structure supporting for all $x \in [1..\sigma]$ and $r \in [1..n]$ queries such as `rank`(x), the number of elements which are strictly smaller than x in W; and `select`(r), the value of the r-th smallest value (counted with multiplicity) in W. Their data structure supports q queries in time within $O(n(1 + \lg q) + q \lg n)$, all in the comparison model.

Karp *et al.*'s data structure [13] supports only `rank` and `select` queries in the comparison model, whereas the computation of optimal prefix free codes requires to sum pairs of weights from the input, and the algorithm that we propose in Section 2.3 requires to sum weights from a range in the input. Such a requirement can be reduced to `partialSum` queries. Whereas `Partial Sum` queries have been defined in the literature based on the positions in the input array, we define such queries here in a way that depends only on the content of the MULTISET (as opposed to a definition depending on the order in which it is given), so that it can be generalized to deferred data structures.

▶ **Definition 4.** Given n unsorted positive weights $W[1..n]$, a `Partial Sum` data structure supports the following queries: `rank`(x), the number of elements which are strictly smaller than x in W; `select`(r), the value of the r-th smallest value (counted with multiplicity) in W; `partialSum`(r), the sum of the r smallest elements (counted with multiplicity) in W.

For example, given the array $A =$ | 5 | 3 | 1 | 5 | 2 | 4 | 6 | 7 |, this definition of the operators yields `rank`$(5) = 4$, `select`$(6) = 5$, and `partialSum`$(2) = 3$.

We describe below how to extend Karp *et al.*'s deferred data structure [13], which supports `rank` and `select` queries on MULTISETS, in order to add the support for `partialSum` queries, with an amortized running time within a constant factor of the original asymptotic time. Note that the operations performed by the data structure are not any more within the comparison model, but rather in the algebraic decision tree computational model, as they introduce algebraic operations (additions) on the elements of the MULTISET. The result is a direct extension of Karp *et al.* [13], adding a sub-task taking linear time (updating partial sums in an interval of positions) to a sub-task which was already taken linear time (partitioning this same interval by a pivot):

▶ **Lemma 5.** *Given n unsorted positive weights $W[1..n]$, there is a `PartialSum` Deferred Data Structure which supports q operations of type `rank`, `select` and `partialSum` in time within $O(n(1+\lg q)+q(1+\log n))$, all within the algebraic decision tree computational model.*

Proof. Karp *et al.* [13] described a deferred data structure which supports the `rank` and `select` queries (but not `partialSum` queries). It is based on median computations and $(2,3)$-trees, and performs q queries on n values in time within $O(n(1+\lg q)+q(1+\log n))$, all within the algebraic decision tree computational model. We describe below how to modify their data structure in a simple way to support `partialSum` queries with asymptotically negligible additional cost. At the initialization of the data structure, compute the n partial sums corresponding to the n positions of the unsorted array. After each median computation and partitioning in a `rank` or `select` query, recompute the partial sums on the range of values newly partitioned, adding only a constant factor to the cost of the query. When answering a `partialSum` query, perform a `select` query and then return the value of the partial sum corresponding to the value by the `select` query: the asymptotic complexity is within a constant factor of the one described by Karp *et al.* [13]. ◀

In the next section we describe an algorithm that uses the deferred data structure described above to batch the operations on the external nodes, and to defer the computation of the weights of some internal nodes for later, so that for many instances the input is not completely sorted at the end of the execution, which reduces the execution cost.

2.3 Algorithm "`Group-Dock-Mix`" (`GDM`)

There are five main phases in the `GDM` algorithm: the *Initialization*, three phases (*Grouping*, *Docking* and *Mixing*, giving it the name "`GDM`" to the algorithm) inside a loop running until only internal nodes are left to process, and the *Conclusion*:

- In the *Initialization* phase, initialize the `Partial Sum` deferred data structure with the input, and the first internal node by pairing the two smallest weights of the input.
- In the *Grouping* phase, group the weights smaller than the smallest internal node: this corresponds to a run of consecutive E in the `EI` signature of the instance.
- In the *Docking* phase, pair the consecutive *positions* of those weights (as opposed to the weights themselves, which can be reordered by future operations) into internal nodes, and pair those internal nodes until the weight of at least one such internal node becomes equal or larger than the smallest remaining weight: this corresponds to a run of consecutive I in the `EI` signature of the instance.
- In the *Mixing* phase, rank the smallest unpaired weight among the weights of the available internal nodes: this corresponds to an occurrence of IE in the `EI` signature of the instance. This is the most complicated (and most costly) phase of the algorithm.

- In the *Conclusion* phase, with i internal nodes left to process, assign codelength $l = \lfloor \log_2 i \rfloor$ to the $i - 2^l$ largest ones and codelength $l+1$ to the 2^l smallest ones: this corresponds to the last run of consecutive I in the `EI` signature of the instance.

The algorithm and its complexity analysis distinguish two types of internal nodes: *pure* nodes, which descendants were all paired during the same *Grouping* phase; and *mixed* nodes, each of which either is the ancestor of such a *mixed* node, or pairs a *pure* internal node with an external node, or pairs two *pure* internal nodes produced at distinct phases of the `GDM` algorithm. The distinction is important as the algorithm computes the weight of any *mixed* node at its creation (potentially generating several data structure operations), whereas it defers the computation of the weight of some *pure* nodes for later, and does not compute the weight of some pure nodes.

Before describing each phase more in detail, it is important to observe the following invariant of the algorithm:

▶ **Lemma 6.** *Given an instance of the optimal prefix free code problem formed by $n > 1$ positive weights $W[1..n]$, between each phase of the algorithm, all unpaired internal nodes have weight within a constant factor of two (i.e. the maximal weight of an unpaired internal node is strictly smaller than twice the minimal weight of an unpaired internal node).*

We now proceed to describe each phase in more details:

- **Initialization:** Initialize the deferred data structure `Partial Sum` with the input; compute the weight `currentMinInternal` of the first internal node through the operation `partialSum`(2) (the sum of the two smallest weights); create this internal node, of weight `currentMinInternal` and children 1 and 2 (the positions of the first and second weights, in any order); compute the weight `currentMinExternal` of the first unpaired weight (i.e. the first available external node) by the operation `select`(3); setup the variables `nbInternals` $= 1$ and `nbExternalProcessed` $= 2$.
- **Grouping:** Compute the position r of the first unpaired weight which is larger than the smallest unpaired internal node, through the operation `rank`(`currentMinInternal`); pair the $((r -$ `nbExternalProcessed`$)$ modulo 2) indices to form $\lfloor \frac{r - \texttt{nbExternalProcessed}}{2} \rfloor$ *pure* internal nodes; if the number $r -$ `nbExternalProcessed` of unpaired weights smaller than the first unpaired internal node is odd, select the r-th weight through the operation `select`(r), compute the weight of the first unpaired internal node, compare it with the next unpaired weight, to form one *mixed* node by combining the minimal of the two with the extraneous weight.
- **Docking:** Pair all internal nodes by batches (by Lemma 6, their weights are all within a factor of two, so all internal nodes of a generation are processed before any internal node of the next generation); after each batch, compare the weight of the largest such internal node (compute it through `partialSum` on its range if it is a *pure* node, otherwise it is already computed) with the first unpaired weight: if smaller, pair another batch, and if larger, the phase is finished.
- **Mixing:** Rank the smallest unpaired weight among the weights of the available internal nodes by a doubling search starting from the beginning of the list of internal nodes. For each comparison, if the internal node's weight is not already known, compute it through a `partialSum` operation on the corresponding range (if it is a *mixed* node, it is already known). If the number r of internal nodes of weight smaller than the unpaired weight is odd, pair all but one, compute the weight of the last one and pair it with the unpaired weight. If r is even, pair all of the r internal nodes of weight smaller than the unpaired weight, compare the weight of the next unpaired internal node with the weight of the

next unpaired external node, and pair the minimum of the two with the first unpaired weight. If there are some unpaired weights left, go back to the *Grouping* phase, otherwise continue to the *Conclusion* phase.

- **Conclusion:** There are only internal nodes left, and their weights are all within a factor of two from each other. Pair the nodes two by two in batches as in the *Docking* phase, computing the weight of an internal node only when the number of internal nodes of a batch is odd.

The combination of those phases forms the `GDM` algorithm, which computes an optimal prefix free code given an unsorted sets of positive integers. In the next section, we analyze the number q of `rank`, `select` and `partialSum` queries performed by the `GDM` algorithm, and deduce from it the complexity of the algorithm in terms of algebraic operations.

3 Analysis

The `GDM` algorithm runs in time within $O(n \lg n)$ in the worst case over instances of size n (which is optimal (if not a new result) in the algebraic decision tree computational model), but much faster on instances with few blocks of consecutive Es in the `EI` signature of the instance. We formalize this concept by defining the *alternation* α of the instance in Section 3.1. We then proceed in Section 3.2 to show upper bounds on the number of queries and operations performed by the `GDM` algorithm in the worst case over instances of fixed size n and alternation α. We finish in Section 3.3 with a matching lower bound for the number of operations performed.

3.1 Alternation $\alpha(W)$

We suggested in Section 2.1 that the number $|S|_{EI}$ of blocks of consecutive Es in the `EI` signature of an instance can be used to measure its difficulty. Indeed, some "easy" instances have few such blocks, and the instance used to prove the $\Omega(n \lg n)$ lower bound on the computational complexity of optimal prefix free codes in the algebraic decision tree computational model in the worst case over instances of size n has $n-1$ such blocks (the maximum possible in an instance of size n). We formally define this measure as the "alternation" of the instance (it measures how many times the van Leeuwen algorithm "alternates" from an external node to an internal node) and denote it by the parameter α:

▶ **Definition 7.** Given an instance of the optimal prefix free code problem formed by n positive weights $W[1..n]$, its *alternation* $\alpha(W) \in [1..n-1]$ is the number of occurrences of the substring "EI" in its `EI` signature $\mathcal{S}(W)$.

This number is of particular interest as it measures the number of iteration of the main loop in the `GDM` algorithm:

▶ **Lemma 8.** *Given an instance of the optimal prefix free code problem of alternation α, the `GDM` algorithm performs α iterations of its main loop.*

In the next section, we refine this result to the number of data structure operations and algebraic operations performed by the `GDM` algorithm.

3.2 Upper Bound

In order to measure the number of queries performed by the `GDM` algorithm, we detail how many queries are performed in each phase of the algorithm.

- The *Initialization* corresponds to a constant number of data structure operations: a `select` operation to find the third smallest weight, and a simple `partialSum` operation to sum the two smallest weights of the input.
- Each *Grouping* phase corresponds to a constant number of data structure operations: a `partialSum` operation to compute the weight of the smallest internal node if needed, and a `rank` operation to identify the unpaired weights which are smaller or equal to this node.
- The number of operations performed by each *Docking* and *Mixing* phase is better analyzed together: if there are i symbols in the I-block corresponding to this phase in the `EI` signature, and if the internal nodes are grouped on h levels before generating an internal node larger than the smallest unpaired weights, the *Docking* phase corresponds to at most h `partialSum` operations, whereas the *Mixing* phase corresponds to at most $\log_2(i/2^h)$ `partialSum` operations, which develops to $\log_2(i)-h$, for a total of $h+\log_2(i)-h = \log_2 i$ data structure operations.
- The *Conclusion* phase corresponds to a number of data structure operations logarithmic in the size of the last block of Is in the `EI` signature of the instance: in the worst case, the weight of one *pure* internal node is computed for each batch, through one single `partialSum` operation each time.

Lemma 8 and the concavity of the log yields the total number of data structure operations performed by the `GDM` algorithm:

▶ **Lemma 9.** *Given an instance of the optimal prefix free code problem of alternation α, the `GDM` algorithm performs within $O(\alpha(1+\lg\frac{n-1}{\alpha}))$ data structure operations on the deferred data structure given as input.*

Proof. For $i \in [1..\alpha]$, let n_i be the number of internal nodes at the beginning of the i-th *Docking* phase. According to Lemma 8 and the analysis of the number of data structure operations performed in each phase, the `GDM` algorithm performs in total within $O(\alpha + \sum_{i=1}^{\alpha} \lg n_i)$ data structure operations. Since there are at most $n-1$ internal nodes, by concavity of the logarithm this is within $O(\alpha + \alpha \lg\frac{n-1}{\alpha}) = O(\alpha(1+\lg\frac{n-1}{\alpha}))$. ◀

Combining this result with the complexity of the `Partial Sum` deferred data structure from Lemma 5 directly yields the complexity of the `GDM` algorithm in algebraic operation (and running time):

▶ **Lemma 10.** *Given an instance of the optimal prefix free code problem of alternation α, the `GDM` algorithm runs in time within $O(n(1+\log\alpha) + \alpha(\lg n)(1+\lg\frac{n-1}{\alpha}))$, all within the algebraic decision tree computational model.*

Proof. Let q be the number of queries performed by the `GDM` algorithm. Lemma 9 implies that $q \in O(\alpha(1+\lg\frac{n-1}{\alpha}))$. Plunging this into the complexity of $O(q\lg n + n\lg q)$ from Lemma 5 yields the complexity $O(n(1+\log\alpha) + \alpha(\lg n)(1+\lg\frac{n-1}{\alpha}))$. ◀

Some simple functional analysis further simplifies the expression to our final upper bound:

▶ **Lemma 11.** *Given two positive integers $n > 0$ and $\alpha \in [1..n-1]$,*

$$O(\alpha(\lg n)(\lg\frac{n}{\alpha})) \subseteq O(n(1+\lg\alpha))$$

Proof. Given two positive integers $n > 0$ and $\alpha \in [1..n-1]$, $\alpha < \frac{n}{\lg n}$ and $\frac{\alpha}{\lg\alpha} < n$. A simple rewriting yields $\frac{\alpha}{\lg\alpha} < \frac{n}{\lg^2 n}$ and $\alpha\lg^2 n > n\lg\alpha$. Then, $n/\alpha < n$ implies $\alpha \times \lg n \times \lg\frac{n}{\alpha} < n\lg\alpha$, which yields the result. ◀

In the next section, we show that this complexity is indeed optimal in the algebraic decision tree computational model, in the worst case over instances of fixed size n and alternation α.

3.3 Lower Bound

A complexity within $O(n(1+\lg\alpha))$ is exactly what one could expect, by analogy with the sorting of Multisets: there are α groups of weights, so that the order within each group does not matter much, but the order between weights from different groups matter a lot. We prove a lower bound within $\Omega(n\lg\alpha)$ by reduction to Multiset sorting:

▶ **Lemma 12.** *Given the integers $n \leq 2$ and $\alpha \in [1..n-1]$, for any algorithm A in the algebraic decision tree computational model, there is a set $W[1..n]$ of n positive weights of alternation α such that A performs within $\Omega(n\lg\alpha)$ algebraic operations.*

Proof. For any Multiset $A[1..n] = \{x_1, \ldots, x_n\}$ of n values from an alphabet of α distinct values, define the instance $W_A = \{2^{x_1}, \ldots, 2^{x_n}\}$ of size n, so that computing an optimal prefix free code for W, sorted by codelength, provides an ordering for A. W has alternation α: for any two distinct values x and y from A, the van Leeuwen algorithm pairs all the weights of value 2^x before pairing any weight of value 2^y, so that the `EI` signature of W_A has α blocks of consecutive Es. The lower bound then results from the classical lower bound on sorting Multisets in the comparison model in the worst case over Multisets of size n with α distinct symbols, itself based on the number α^n of such multisets. ◀

We compare our results to previous ones in the next section.

4 Discussion

We described an algorithm computing an optimal prefix free code for n unsorted positive weights in time within $O(n(1+\lg\alpha)) \subseteq O(n\lg n)$, where the alternation $\alpha \in [1..n-1]$ roughly measures the amount of sorting required by the computation by combining van Leeuwen's results about optimal prefix free codes [14], known since 1976, with Karp *et al.*'s results about Deferred Data Structures [13], known since 1988. The results described above yield many new questions, of which we discuss only a few in the following sections: how do those results relate to previous results on optimal prefix free codes (Section 4.1), or to other results on Deferred Data Structures obtained since 1988 (Section 4.2 and 4.3). We discuss the potential lack of practical applications of our results on optimal prefix free codes in Section 4.4, and the perspectives of research on this topic in Section 4.5.

4.1 Relation to previous work on optimal prefix free codes

In 2006, Belal *et al.* [5], described a variant of Milidiú *et al.*'s algorithm [16, 15] to compute optimal prefix free codes, announcing that it performs $O(kn)$ algebraic operations when the weights are not sorted, where k is the number of distinct code lengths in some optimal prefix free code. They describe an algorithm claimed to run in time $O(16^k n)$ when the weights are unsorted, and propose to improve the complexity to $O(kn)$ by partitioning the weights into smaller groups, each corresponding to disjoint intervals of weights[3]. The claimed complexity

[3] Those results were downgraded in the December 2010 update of their initial 2005 publication through `Arxiv` [4].

is asymptotically better than the one suggested by Huffman when $k \in o(\log n)$, and they raise the question of whether there exists an algorithm running in time $O(n \log k)$.

Like the `GDM` algorithm, the algorithm described by Belal *et al.* [5] for the unsorted case is based on several computations of the median of the weights within a given interval, in particular, in order to select the weights smaller than some well chosen value. The essential difference between both works is the use of a deferred data structure, which simplifies both the algorithm and the analysis of its complexity.

While an algorithm running in time within $O(n \lg k)$ would improve over the running time within $O(n(1 + \lg \alpha))$ of our proposed solution, such an algorithm has not been defined yet, and for $\alpha < 2^k$ our solution is superior to the complexity within $O(nk)$ claimed by Belal and Elmasry [5] (and even more so over the complexity of $O(16^k n)$).

4.2 Applicability of dynamic results on Deferred Data Structures

Karp *et al.* [13] defined the first Deferred Data Structures, supporting `rank` and `select` on MULTISETS and other queries on CONVEX HULL. They left as an open problem the support of dynamic operators such as `insert` and `delete`. Ching *et al.* [7] quickly demonstrated how to add such support in good amortized time.

The dynamic addition and deletion of elements in a deferred data structure (added by Ching *et al.* [7] to Karp *et al.* [13]'s results) does not seem to have any application to the computation of optimal prefix free codes: even if the list of weights was dynamic, further work is required to build a deferred data structure supporting prefix free code queries.

4.3 Applicability of refined results on Deferred Data Structures

Karp *et al.*'s analysis [13] of the complexity of the deferred data structure is in function of the total number q of queries and operators, while Kaligosi *et al.* [12] analyzed the complexity of an offline version in function of the size of the gaps between the positions of the queries. Barbay *et al.*[2] combined the three results into a single deferred data structure for MULTISETS which supports the operators `rank` and `select` in amortized time proportional to the entropy of the distribution of the sizes of the gaps between the positions of the queries.

At first view, one could hope to generalize the refined entropy analysis (introduced by Kaligosi *et al.* [12] and applied by Barbay *et al.*[2] to the online version) of MULTISETS deferred data structures supporting `rank` and `select` to the computational complexity of optimal prefix free codes: a complexity proportional to the entropy of the distribution of codelengths in the output would nicely match the lower bound of $\Omega(k(1 + \mathcal{H}(n_1, \ldots, n_h)))$ suggested by information theory, where the output contains n_i codes of length l_i, for some integer vector $(l_1, \ldots, l_h)$ of distinct codelengths and some integer h measuring the number of distinct codelengths. Our current analysis does not yield such a result: the gap lengths between queries in the list of weights are not as regular as $(l_1, \ldots, l_h)$.

4.4 Potential (lack of) Practical Impact of our Results

We expect the impact of our faster algorithm on the execution time of optimal prefix free code based techniques to be of little importance in most cases: compressing a sequence S of $|S|$ messages from an input alphabet of size n requires not only computing the code (in time $O(n(1 + \lg \alpha))$ using our solution), but also computing the weights of the messages (in time $|S|$), and encoding the sequence S itself using the computed code (in time $O(|S|)$), which usually dominates the total cost. Improving the code computation time will improve on the

compression time only in cases where the size n of the input alphabet is very large compared to the length $|S|$ of the compressed sequence. One such application is the compression of texts in natural language, where the input alphabet is composed of all the natural words [18]. Another potential application is the boosting technique from Ferragina *et al.* [9], which divides the input sequence into very short subsequence and computes a prefix free code for each subsequences on the input alphabet of the whole sequence.

Another argument for the potential lack of practical impact of our result is that there exist algorithms computing optimal prefix free codes in time within $O(n \lg \lg n)$ within the RAM model[4]: a time complexity within $O(n(1 + \lg \alpha))$ is an improvement only for values of $\alpha \in o(\lg n)$.

4.5 Perspectives

One could hope for an algorithm with a complexity that matches the lower bound of $\Omega(k(1+\mathcal{H}(n_1, \ldots, n_h)))$ suggested by information theory, where the output contains n_i codes of length l_i, for some integer vector $(l_1, \ldots, l_h)$ of distinct codelengths and some integer h measuring the number of distinct codelengths. Our current analysis does not yield such a result: the gap lengths between queries in the list of weights are not as regular as $(l_1, \ldots, l_h)$, but a refined analysis might. Minor improvements of our results could be obtained by studying the problem in external memory, where deferred data structures have also been developed [19, 3], or when the alphabet size is larger than two, as in the original article from Huffman [11].

Another promising line of research is given by variants of the original problem, such as Optimal Bounded Length Prefix Free Codes, where the maximal length of each word of the prefix free code must be less than or equal to a parameter l, while still minimizing the entropy of the code; or such as the Order Constrained Prefix Free Codes, where the order of the words of the codes is constrained to be the same as the order of the weights. Both problems have complexity $O(n \log n)$ in the worst case over instances of fixed input size n, while having linear complexity when all the weights are within a factor of two of each other, exactly as in the original problem.

Many communication solutions use an optimal prefix free code computed offline. A logical step would be to study if any can now afford to compute a new optimal prefix free code more frequently, and see their compression performance improved by a faster prefix free code algorithm.

Acknowledgements: The author is funded by the Millennium Nucleus RC130003 "Information and Coordination in Networks". He would like to thank Peyman Afshani and Seth Pettie for interesting discussions during the author's visit to the center MADALGO in January 2014; Jouni Siren for detecting a central error in a previous version of this work; Gonzalo Navarro for suggesting the application to the boosting technique from Ferragina *et al.* [9]; Charlie Clarke, Gordon Cormack, and J. Ian Munro for helping clarify the history of van Leeuwen's algorithm [14]; Renato Cerro for various English corrections; various people who have reviewed and commented on various preliminary drafts and presentations of re-

[4] The algorithm proposed by van Leeuwen [14] reduces in time linear in the number of symbols of the alphabet the computation of an optimal prefix free code to their sorting, and Han [10] described how to sort a set of n integers (which input symbol frequencies are) in time within $O(n \lg \lg n)$ in the RAM model.

lated work: Carlos Ochoa, Francisco Claude-Faust, Javiel Rojas, Peyman Afshani, Roberto Konow, Seth Pettie, Timothy Chan, and Travis Gagie.

References

1 Jérémy Barbay. Optimal prefix free codes with partial sorting. *arXiv preprint arXiv:1602.00023*, 2016. URL: `http://arxiv.org/1602.00023`.

2 Jérémy Barbay, Ankur Gupta, Seungbum Jo, Srinivasa Rao Satti, and Jonathan Sorenson. Theory and implementation of online multiselection algorithms. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*, 2013.

3 Jérémy Barbay, Ankur Gupta, S. Srinivasa Rao, and Jonathan Sorenson. Dynamic online multiselection in internal and external memory. In *Proceedings of the International Workshop on Algorithms and Computation (WALCOM)*, 2014.

4 Ahmed A. Belal and Amr Elmasry. Distribution-sensitive construction of minimum-redundancy prefix codes. *CoRR*, abs/cs/0509015, 2005. Version of Tue, 21 Dec 2010 14:22:41 GMT, with downgraded results from the ones in the conference version [5].

5 Ahmed A. Belal and Amr Elmasry. Distribution-sensitive construction of minimum-redundancy prefix codes. In Bruno Durand and Wolfgang Thomas, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3884 of *Lecture Notes in Computer Science*, pages 92–103. Springer, 2006. `doi:10.1007/11672142_6`.

6 Ahmed A. Belal and Amr Elmasry. Verification of minimum-redundancy prefix codes. *IEEE Transactions on Information Theory (TIT)*, 52(4):1399–1404, 2006. `doi:10.1109/TIT.2006.871578`.

7 Yu-Tai Ching, Kurt Mehlhorn, and Michiel H.M. Smid. Dynamic deferred data structuring. *Information Processing Letters (IPL)*, 35(1):37–40, June 1990.

8 Shimon Even and Guy Even. *Graph Algorithms, Second Edition*. Cambridge University Press, 2012.

9 Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005. `doi:10.1145/1082036.1082043`.

10 Yijie Han. Deterministic sorting in $o(n \log\log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004. `doi:10.1016/j.jalgor.2003.09.001`.

11 David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers (IRE)*, 40(9):1098–1101, September 1952.

12 Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In *Proceedings of the International Conference on Automata, Languages, and Programming (ICALP)*, pages 103–114, 2005. `doi:10.1007/11523468_9`.

13 R. Karp, R. Motwani, and P. Raghavan. Deferred data structuring. *SIAM Journal of Computing (SJC)*, 17(5):883–902, 1988. `doi:10.1137/0217055`.

14 J. Van Leeuwen. On the construction of Huffman trees. In *Proceedings of the International Conference on Automata, Languages, and Programming (ICALP)*, pages 382–410, Edinburgh University, 1976.

15 Ruy Luiz Milidiú, Artur Alves Pessoa, and Eduardo Sany Laber. A space-economical algorithm for minimum-redundancy coding. Technical report, Departamento de Informática, PUC-RJ, Rio de, 1998.

16 Ruy Luiz Milidiú, Artur Alves Pessoa, and Eduardo Sany Laber. Three space-economical algorithms for calculating minimum-redundancy prefix codes. *IEEE Transactions on Information Theory (TIT)*, 47(6):2185–2198, September 2001. `doi:10.1109/18.945242`.

17 Alistair Moffat and Andrew Turpin. Efficient construction of minimum-redundancy codes for large alphabets. *IEEE Transactions on Information Theory (TIT)*, pages 1650–1657, 1998.

18 E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

19 Jop F. Sibeyn. External selection. *Journal of Algorithms (JALG)*, 58(2):104–117, 2006. doi:10.1016/j.jalgor.2005.02.002.