Efficient and practical algorithms for sequence analysis algorithm and data analysis research

Alzamel, Mai

Awarding institution:
King's College London

Efficient and Practical Algorithms for Sequence Analysis

Algorithm and Data Analysis Research

Mai Abdulaziz Alzamel

Department of Informatics

King's College London

This dissertation is submitted for the degree of

Doctor of Philosophy

September 2021

Dedicated to

my beloved father **Abdulaziz Alzamel** and my precious mother **Turfa Almugbel**.

Declaration

I declare that this doctorate thesis and the work presented in it are my own and have been generated by me as the result of my original research. The following publications were made during my Ph.D. Specific topics and results from my list publications are discussed in detail at the body of this thesis.

List of publications

[1] M. Adamczyk, **M. Alzamel**, P. Charalampopoulos, C. Iliopoulos, and J. Radoszewski. Palindromic decompositions with gaps and errors. In *Computer Science - Theory and Applications - 12th International Computer Science Symposium in Russia, CSR 2017, Kazan, Russia, June 8-12, 2017, Proceedings*, pages 48–61, 2017.

[2] M. Adamczyk, **M. Alzamel**, P. Charalampopoulos, and J. Radoszewski. Palindromic decompositions with gaps and errors. *Int. J. Found. Comput. Sci.*, 29(8):1311–1329, 2018.

[3] H. Alamro, **M. Alzamel**, C. Iliopoulos, S. P. Pissis, S. Watts, and W. Sung. Efficient identification of k-closed strings. In *Engineering Applications of Neural Networks - 18th International Conference, EANN 2017, Athens, Greece, August 25-27, 2017, Proceedings*, pages 583–595, 2017.

[4] H. Alamro, **M. Alzamel**, C. S. Iliopoulos, S. P. Pissis, and S. Watts. Iupacpal: efficient identification of inverted repeats in iupac-encoded DNA sequences. *BMC Bioinform.*, 22(1):51, 2021.

[5] H. Alamro, **Mai Alzamel**, C. S. Iliopoulos, S. P. Pissis, W. Sung, and S. Watts. Efficient identification of k-closed strings. *Int. J. Found. Comput. Sci.*, 31(5):595–610, 2020.

[6] C. Pockrandt, **Mai Alzamel**, C. S. Iliopoulos, and K. Reinert. Genmap: ultra-fast computation of genome mappability. *Bioinform.*, 36(12):3687–3692, 2020.

[7] **M. Alzamel**, L. A. K. Ayad, G. Bernardini, R. Grossi, C. Iliopoulos, N. Pisanti, S. P. Pissis, and G. Rosone. Degenerate string comparison and applications. In *18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20-22, 2018, Helsinki, Finland*, pages 21:1–21:14, 2018.

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

[9] **M. Alzamel**, P. Charalampopoulos, C. Iliopoulos, T. Kociumaka, S. P. Pissis, J. Radoszewski, and J. Straszynski. Efficient computation of sequence mappability. In *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, pages 12–26, 2018.

[10] **M. Alzamel**, P. Charalampopoulos, C. Iliopoulos, and S. P. Pissis. How to answer a small batch of rmqs or LCA queries in practice. In *Combinatorial Algorithms - 28th International Workshop, IWOCA 2017, Newcastle, NSW, Australia, July 17-21, 2017, Revised Selected Papers*, pages 343–355, 2017.

[11] **M. Alzamel**, P. Charalampopoulos, C. Iliopoulos, S. P. Pissis, J. Radoszewski, and W. Sung. Faster algorithms for 1-mappability of a sequence. In *Combinatorial Optimization and Applications - 11th International Conference, COCOA 2017, Shanghai, China, December 16-18, 2017, Proceedings, Part II*, pages 109–121, 2017.

[12] **M. Alzamel**, M. Crochemore, C. S. Iliopoulos, T. Kociumaka, R. Kundu, J. Radoszewski, W. Rytter, and T. Walen. How much different are two words with different shortest periods. In *Artificial Intelligence Applications and Innovations - AIAI 2018*

IFIP WG 12.5 International Workshops, SEDSEAL, 5G-PINE, MHDW, and HEALTH-IOT, Rhodes, Greece, May 25-27, 2018, Proceedings, pages 168–178, 2018.

[13] **M. Alzamel**, J. Gao, C. Iliopoulos, C. Liu, and S. P. Pissis. Efficient computation of palindromes in sequences with uncertainties. In *Engineering Applications of Neural Networks - 18th International Conference, EANN 2017, Athens, Greece, August 25-27, 2017, Proceedings*, pages 620–629, 2017.

[14] **M. Alzamel**, J. Gao, C. S. Iliopoulos, and C. Liu. Efficient computation of palindromes in sequences with uncertainties. *Fundam. Inform.*, 163(3):253–266, 2018.

[15] **M. Alzamel** and C. Iliopoulos. Recent advances of palindromic factorization. In *Combinatorial Algorithms - 28th International Workshop, IWOCA 2017, Newcastle, NSW, Australia, July 17-21, 2017, Revised Selected Papers*, pages 37–46, 2017.

[16] **M. Alzamel**, C. S. Iliopoulos, W. F. Smyth, and W. Sung. Off-line and on-line algorithms for closed string factorization. *Theor. Comput. Sci.*, 792:12–19, 2019.

[17] **Mai Alzamel**, P. Charalampopoulos, C. S. Iliopoulos, S. P. Pissis, J. Radoszewski, and W. Sung. Faster algorithms for 1-mappability of a sequence. *Theor. Comput. Sci.*, 812:2–12, 2020.

Mai Abdulaziz Alzamel

September 2021

Acknowledgements

First and foremost, I would like to thank my father, **Abdulaziz Alzamel**, for encouraging me to pursue my higher education, and my mother, **Turfa Almugbel**, for her constant care and solicitude. This journey would not have happened without their comprehensive support.

I would like to express my sincere gratitude to my supervisor, **Prof. Costas Iliopoulos**, for his continuous support through the course of my Ph.D. study and related research. During my Ph.D. I have published several papers in top-ranked computer science journals and conferences. He gave me a high level of research guidance, including editing journal special issues, chairing sessions in workshops, reviewing papers and chairing and organising conferences. I would like to thank him for sending me to present my results at well-known conferences and universities. I would also like to thank him for his support in arranging research visits and international collaboration with prestigious universities and researchers. Finally, I would like to thank him for giving me a wonderful four years in my academic life. I believe my Ph.D. would not have been the same without his supervision.

I would like to thank my second supervisor, **Dr. Solon Pissis**, for his patient and precise supervision. He was a great monitor for my Ph.D. results, especially the practical parts. I would further like to thank my co-authors for their cooperation in our joint works during

my Ph.D study. Here I give special acknowledgement to **Prof. Ken Sung** from **National University of Singapore** for his encouragement and support during my study. I would also like to thank the members of the ADA group for creating a stimulating research community and fruitful discussion on various research matters.

My Ph.D. journey would not have been the same without my siblings' help, I would like to thank my beloved sisters and brothers **Aljohara**, **Mohammed**, **Reem**, **Faisal**, **Meshael** and **Abdulellah**, as well as my friends, for their support.

Last but not least, I would like to acknowledge **King Saud Univerity** and the **Ministry of Education of Saudi Arabia** for sponsoring me through this entire journey.

Abstract

This thesis studies four computational problems derived from molecular sequence analysis, which play a core role in many real-life applications. Its purpose is to develop efficient, fast and practical algorithms for use in sequence analysis. The approach is motivated by bio-informatics, but has a wide range of other applications. Firstly, we focus on the 1-mappability problem associated with a given sequence; in this the goal is to compute a table, wherein each entry comprises the number of repeats of specific substrings of a given length that start at this entry, with one mismatch at most. Furthermore, we present an "expected" linear-time algorithm (average case complexity) linked to the same problem, and we also generalise the algorithm to k-mappability that permits up to k mismatches. Additionally, we study a new type of uncertain string, called "degenerate strings"; here our goal is to locate maximal palindromes. We provide a linear algorithm for string comparison, and then use this to provide two algorithms to report maximal palindromes. Moreover, we illustrate a novel algorithm in the presence of k errors, as a way to determine whether an input string is a "closed string". Finally, we revisit the well-known Range Minimum Query (RMQ) problem and consider its variant when processing a small batch of RMQs in real-world applications, as well as the connection between the RMQ and the Lowest Common Ancestor (LCA) problem. In the case of all the new algorithms presented here, we implemented the necessary libraries and demonstrated the efficiency of the algorithms by testing the software across extensive data-sets.

Table of contents

List of figures

List of tables

List of Algorithms

List of abbreviations

CPU Central Processing Unit

DAG Directed Acyclic Graph

ED Elastic Degenerate

EXT Extension

GHz Gigahertz

GNU General Public Licence

GRCh37 Genome Reference Consortium Human Build 37

IUPAC International Union of Pure and Applied Chemistry

kbytes Kilobytes

lcs Longest Common Suffix

LPM Longest Prefix Match

LSM Longest Suffix Match

MB Megabyte

mRNA Messenger RNA

MSA Multiple Sequence Alignment

Occ Occurance

PC Personal Computer

RAM Random Access Memory

Resp Respectively

A Adenine

BFS Breadth First Search

C Cytosine

DFS Deterministic finite automaton

DNA Deoxyribonucleic Acid

G Guanine

GD Generalised Degenerate

HIV Human Immune-deficiency Virus

iSA Inverse Suffix Array

LCA Lowest Common Ancestor

LCE Longest Common Extension

LCP Longest Common Prefix

LTRs Long Terminal Repeats

NFA Non-deterministic finite automaton

PAT Tree Patricia tree

RMQ Range Minimum Query

SA Suffix Array

SETH Strong Exponential Time Hypothesis

T Thymine

TSDs Target Site Duplications

U Uracil

Chapter 1

Introduction

1.1 Background

Two well-known sequences that have motivated computer scientists to improve algorithm efficiency to facilitate their processing are the Deoxyribonucleic acid (DNA) and Ribonucleic acid (RNA) sequences in molecular biology. The DNA sequence carries genetic information for reproduction, development and growth of living organisms, and was discovered by Francis Crick and James D. Watson in 1953 [96]. The sequence consists of two strands of nucleotides, arranged into a double helix. Each nucleotide combines deoxyribose, a phosphate group and one of the nucleobases: Adenine (`A`), Guanine (`G`), Cytosine (`C`) or Thymine (`T`) [94]. The RNA sequence, meanwhile, is a polymeric molecule that plays a vital role in biology; especially in gene function, where it describes coding, decoding, expression and regularities. RNA has only one strand made up of the nucleotide bases: Adenine (`A`), Guanine (`G`), Cytosine (`C`) and Uracil (`U`) [20, 46].

A string is a sequence of symbols derived from a given alphabet Σ. The early stages of the research focused on ordinary pattern matching and several associated variants. For example, an alphabet can be fixed, variable, finite or infinite in size, and the relationship between symbols can be either ordered or unordered. The first linear method employed to determine whether a pattern occurs in another string was reported by Knuth–Morris–Pratt [64]. The regularities of strings have also been studied extensively: periods, squares, cubes, repetitions, palindromes, runs, inverted repeats and tandem repeats, etc. Thus far, a substantial amount of research has been conducted investigating strings in the areas of pattern matching, data compression, compressed matching, data structure, and the discovery of regularities.

Two of the criteria most extensively used for measuring the similarity between two sequences are: (i) the Hamming distance, introduced by Richard Hamming in [50], and (ii) the Levenshtein distance, as discovered by Vladimir Levenshtein in [67]. The Hamming distance clarifies the minimum number of substitutions required to change one sequence into another. The Levenshtein distance, meanwhile, describes the minimum number of edit operations (insertions, deletions, substitutions) required to transform one string into another. The primary aim of this thesis is to develop efficient algorithms and implementations to allow several types of approximations, including the one mentioned above.

In the case of non-standard string matching, we have "degenerate strings", wherein a degenerate symbol describes a collection of symbols, "weighted strings", of which each occurs with a certain probability, "order preserving" strings, wherein "the shape of the string" describes the matching criterion δ-strings, whereby the symbols are equal within a given tolerance δ, etc.

This work focuses on the string pattern matching problem, which relates to **genome mappability** between species, and gene classes as revealed in [26]. Analysis of data derived from massively parallel sequencing experiments often depends on the process of genome assembly as associated with re-sequencing; namely, assembly with the help of a reference sequence. In this process, a large number of reads (or short sequences) derived from a DNA donor must be mapped back to a reference sequence, comprising a few gigabases, to establish the section of genome from which each read has been derived. An extensive number of short-read alignment techniques and tools have been introduced to address this challenge, each emphasising different aspects of the process [34]. In turn, the process of re-sequencing is heavily reliant on how mappable a genome is, given a set of reads of fixed length m.

Additionally, palindromic sequences have been studied extensively in molecular biology. They are often distributed around promoters, introns, and untranslated regions, and play important roles in gene regulation and other cell processes (see e.g. [5],[2] ,[80]). Identifying palindromes in sequence has become an interesting line of research in combinatorics, and also in computational biology, following the discovery of the importance of palindromes in the DNA sequence of the HIV virus. In particular, these are strings of the form $x\bar{x}^R$, also known as complemented palindromes, that occur in single-stranded DNA, or more commonly, in RNA, where x is a string and $\bar{x}^R$ is the reverse complement of x. In DNA, `C-G` are complements, and `A-T` are complements; in RNA, `C-G` are complements, and `A-U` are complements [8, 3]. Hence, this thesis studies the task of locating palindromes in uncertain string sets. These are referred to as **degenerate strings**, and identifying them required the development of a new string comparison method, and an exploration of its potential applications.

Moreover, this work investigates a recently introduced type of string called a **closed string**, since the theoretical and practical relevance of palindromic strings was established via their relationship with a particular type of sequencing, referred to as a closed string. The number of closed factors in a string can be minimised if the factors are also palindromic, as shown in [21]. Additionally it emerged that the upper boundary on the number of palindromic factors of a string coincides with the lower boundary on the number of closed factors [13]. Thus, the study of closed strings creates potential in respect to the application of palindromes. A direct motivation comes from computational biology: Target Site Duplications (TSDs) are direct repeats found to arise at the insertion sites of transposable elements. They are thought to occur due to the filling in of sticky ends (borders) derived from the staggered cut by transposes. They flank transposable elements, and can be used to identify their loci in the genome. Long Terminal Repeats (LTRs) are direct repeats, which flank the transposed coding regions, and which are themselves flanked by TSDs [68][42].

Finally, in many textbook solutions for classical string matching problems (e.g. maximal palindromic factors, approximate string matching with k-mismatches, approximate string matching with k-differences, online string searching with the suffix array, etc.) we encounter an array A of a sequence with size n, and a number of queries q to be answered, $q = \Omega(n)$ and/or the queries have to be answered *online*. In other algorithms, however, q can in practice be *much smaller* on average, and therefore queries can be answered *offline*. We describe here a few solutions called **range minimum queries**. The common idea, as in many fast average-case algorithms, requires us to minimise the number of queries by *filtering out* identifying queries that can never lead to a valid solution.

1.2 Structure of This Thesis

This thesis is structured as follows:

- In Chapter 2, we present the basic concepts and definitions employed in this thesis.

- In Chapter 3, we study the *k-mappability* problem, and present an algorithm that requires an average-case time and space $\mathcal{O}(n)$ for integer alphabets of size σ if $m = \Omega(\log_\sigma n)$ of given a string x of length n and integers $m < n$ of the presence of k mismatches $k < m$. Notably, we demonstrate that this algorithm is generalisable under arbitrary conditions k, requiring average-case time $\mathcal{O}(kn)$ and space $\mathcal{O}(n)$ if $m = \Omega(k \log_\sigma n)$, assuming letters are independent and uniformly distributed according to random variables. We also provide an experimental evaluation of our average-case algorithm, demonstrating its competitiveness in respect of state-of-the-art implementation.

- In Chapter 4, we study a new type of uncertain sequence, called a *generalised degenerate string* (GD string), and present a linear-time algorithm for the purpose of GD string comparison. We then provide two efficient algorithms to identify the maximal palindrome in GD strings. In addition, we prove the concept through experimental results that are based on real data-sets.

- In Chapter 5, we address a novel problem by extending the *closed string* problem to the *k-closed string* problem. In this case, a level of approximation is permitted up to several Hamming distance errors, as set out by the parameter k. We also address

the problem of deciding whether or not a given string of length n across an integer alphabet is k-closed, additionally specifying the border, and resulting in the string being k-closed. Specifically, we present a $\mathcal{O}(kn)$-time and $\mathcal{O}(n)$-space algorithm to achieve this, along with the pseudocode for an implementation and proof-of-concept in experimental results.

- In Chapter 6, we show that answering a small batch of Range Minimum Queries (RMQs), which is a core computational task in many real-world applications, is associated in particular with the Lowest Common Ancestor (LCA) problem. By *small batch*, we mean that the number q of queries is $o(n)$, where, n is the size of the given integer array A and we have them all at hand. It is therefore not relevant to build a $\Omega(n)$-sized data structure, or spend $\Omega(n)$ time to build a more succinct one. It is well-known among practitioners and elsewhere, that these data structures for online querying carry high constants in their pre-processing and querying time. Therefore, we would like to find an efficient answer to this batch in practice. By *efficiently in practice*, we mean that we (ultimately) want to spend $n + \mathcal{O}(q)$ time and $\mathcal{O}(q)$ space. We write n to emphasise that the number of operations per entry of A should have a very small constant. We illustrate how existing algorithms can be easily modified to satisfy these conditions. The experimental results presented highlight the practicality of this new scheme. The most significant improvement obtained here relates to answering small batches of LCA queries. Finally, we produced a library detailing the process of implementing the algorithms presented.

- In Chapter 7, we provide a conclusion and discuss potential future work.

Chapter 2

Basic Concepts

2.1 Strings

Let Σ be a finite ordered alphabet of size $\sigma := |\Sigma|$. A *string* is defined as a sequence of zero or more symbols from Σ. An *empty string* is a string of length 0, denoted by ε. A string x of length n is represented by the sequence $x = x[0]x[1]\ldots x[n-1]$. Furthermore, we also consider strings according to an *integer alphabet* Σ, where each letter is replaced by its rank in such a way that the resulting string consists of integers in the range $\{0,\ldots,n-1\}$, where $\sigma = n$. Additionally, we consider cases where the alphabet σ is constant, that is $|\Sigma| = c$. For example, DNA sequences are defined according to the alphabet $\Sigma = \{\mathtt{A},\mathtt{G},\mathtt{C},\mathtt{T}\}$ and $|\Sigma| = 4$. The string $x[i\ldots j]$, $0 \leq i \leq j < |x|$ is said to be a *factor* or *substring* of x. We say that an *occurrence* of y exists in x, or, more simply, that y *occurs in* x, when y is a factor of x. Every occurrence of y can be characterised by its starting position in x. Thus, we say that y occurs at the *starting position* i in x, when $y = x[i..i+m-1]$. The factor $x[0\ldots j]$ is said to be

a *prefix* of x and the factor $x[j \ldots n-1]$ is said to be a *suffix* of x. The factor $x[0 \ldots j]$ and $j < n-1$ is a *proper prefix* of x and $x[j \ldots n-1]$, where $1 \leq j < n$ is a *proper suffix* of x. The string $x[n-1]x[n-2]\ldots x[1]x[0]$ is said to be the reverse string of x, denoted by x^R. The string $\bar{x}[0]\ldots\bar{x}[n-1]$ is said to be the *complement* of x, where $\bar{x}[i]$ is the complement of $x[i]$, $0 \leq i \leq n-1$. For example, in DNA sequences, we have $\bar{\texttt{A}} = \texttt{T}$, $\bar{\texttt{G}} = \texttt{C}$, $\bar{\texttt{T}} = \texttt{A}$ and $\bar{\texttt{C}} = \texttt{G}$.

2.2 Hamming Distance

The *Hamming distance* between two equal length strings, x and y, is defined to be the number of positions in both x and y with different symbols, denoted by $\delta_H(x,y) = |\{i : x[i] \neq y[i], i = 0,1,\ldots,|x|-1\}|$. For the sake of completeness, if x and y are of different length, $|x| \neq |y|$, we set $\delta_H(x,y) = \infty$. If two strings x and y have a Hamming distance of k or less, we say that x and y *k-match*, written as $x \approx_k y$.

2.3 Patricia Trees and Suffix Trees

A *Patricia tree* (or PAT Tree) is a tree representing all the suffixes of a string x of length n, which is a natural and easy way to store a string, and then search it, count occurrences, and many other queries. We will make use of a special symbol \$ that is not in the alphabet Σ, and we will construct the tree for x\$, with the following properties: (i) one node per common prefix; (ii) each edge is labelled with a symbol of x; (iii) each path from root to leaf represents a suffix; (iv) every suffix of x is represented by a path from root to leaf, ; (v) no two edges

outgoing from the same node have the same label. The cost to construct the Patricia tree is $\mathcal{O}(n^2)$-time and $\mathcal{O}(n^2)$- space [79].

Example 1. Given a string $x = \texttt{CAACAACC\$}$, the Patricia tree of string x is shown in Figure 2.1, where every path from the root to the leaf node that is labelled with \$ represents one suffix of x.

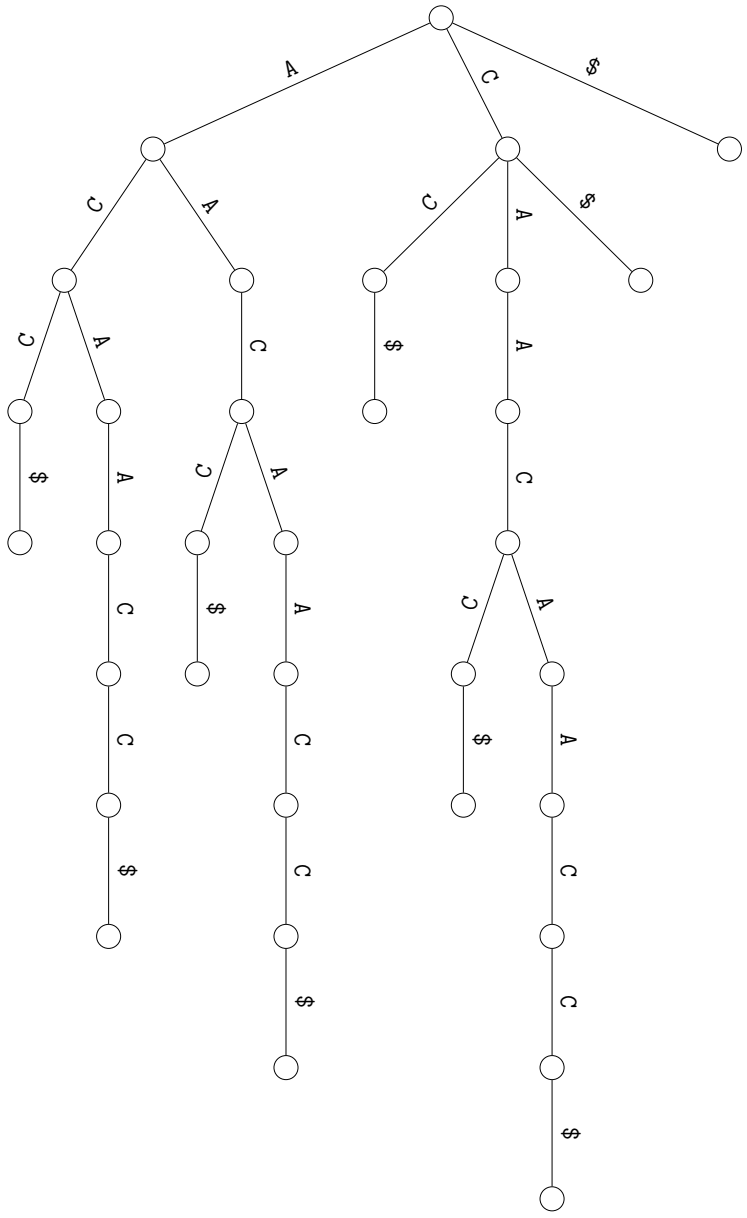


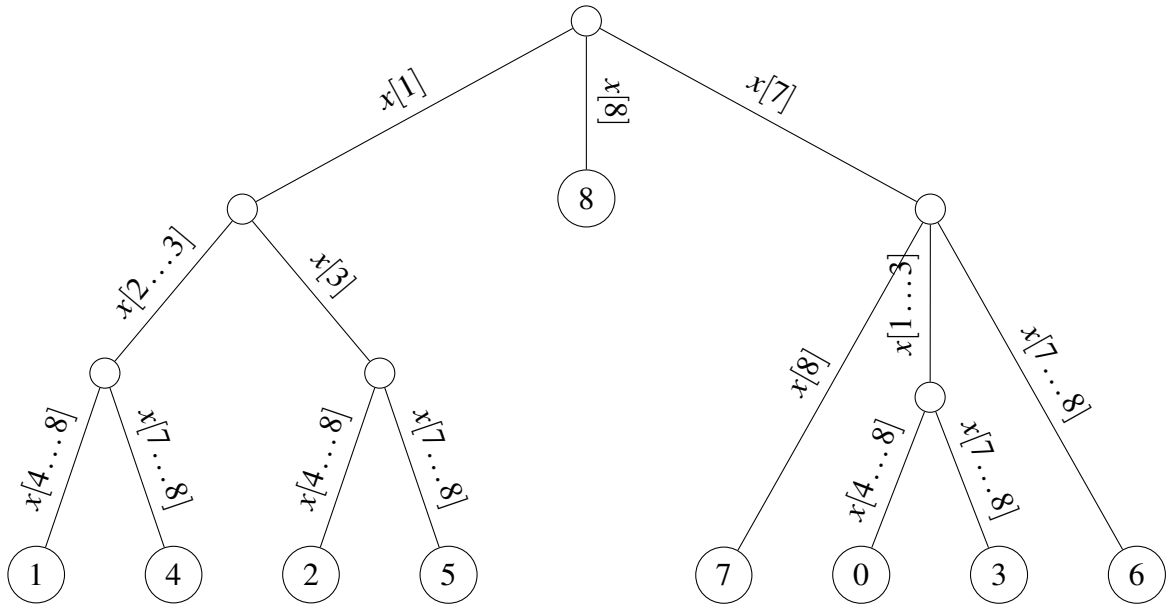
Fig. 2.1 Patricia tree for $x = \texttt{CAACAACC\$}$

.

The *suffix tree* (introduced by Weiner [97]) $\mathscr{T}(x)$ of a string x is a compact tree representing all suffixes of $x\$$ ($ as above), with the following properties: (i) The tree has exactly n leaves numbered from 0 to $n-1$; (ii) Every internal node has at least two children except for the root node; (iii) each edge is labelled with $[i,l]$, where i is the starting position

of $x[i \ldots i+l]$ and l is its length; (iv) any two edges outgoing from a node have different starting string labels, and; (v) the string obtained by concatenating all the string represented by the labels found on the path from the root to leaf i, spells out the suffix $x[i \ldots n-1]$, for $0 \leq i \leq n-1$. The suffix tree was a major discovery made by Weiner in 1973, followed by many other greatly simplified constructions. The algorithms for its construction are $\mathscr{O}(n)$ for constant alphabets, and $\mathscr{O}(n \log |\Sigma|)$ for general alphabets. Farach [27] demonstrated an $\mathscr{O}(n)$ algorithm for integer alphabets. In practice, however, all algorithms are hampered by the space needed to represent the tree. In other words, there is a large hidden constant in the big $\mathscr{O}$ notation. This led to a new, more efficient, data structure called a suffix array. Full technical details of the suffix tree data structure can be found in [27] [97]. There are several applications of suffix trees; for example, finding the longest repeated substring, finding the longest common substring and finding the longest palindrome in a string.

Example 2. A compact representation of a Patricia tree (suffix tree) $\mathscr{T}(x)$ of x is shown in Figure 2.2. The numbered leafs nodes represent all suffixes of string x. For example, node 1 represents the suffix, `A`,`AC`,`AACC$`, which starts at position 1 in x.



i	0	1	2	3	4	5	6	7	8
x	C	A	A	C	A	A	C	C	$

Fig. 2.2 Suffix tree for $x =$ CAACAACC$.

2.4 Suffix Array and Inverse Suffix Array

Let x be a string of length $n > 0$. The *suffix array* of x, denoted by SA, was designed by Manber and Myers [73] in order to improve on the space needed for suffix trees. A SA can also be used as efficiently as the suffix tree for exact string matching or substring searching problems. The SA is an integer array of size n, storing the starting positions of all non-empty suffixes of x in a lexicographical order, i.e. , we have:

$$x[\mathsf{SA}[r-1]..n-1] < x[\mathsf{SA}[r]..n-1], \text{ for all } 1 \leq r < n.$$

Following [48, 72], the *inverse suffix array*, iSA, of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n$. Three different set of co-authors [63, 62, 82] simultaneously but independently demonstrated that the SA and iSA of a string of length n, defined according to an integer alphabet, can be computed in $\mathscr{O}(n)$-time for a constant and integer alphabet and in $\mathscr{O}(n \log n)$-time for general alphabets.

Example 3. Given a string $x =$ `CAACAACC$` the SA and iSA of x are shown in Table 2.1.

i	Suffix
8	`$`
7	`C$`
6	`CC$`
5	`ACC$`
4	`AACC$`
3	`CAACC$`
2	`ACAACC$`
1	`AACAACC$`
0	`CAACAACC$`

(a) The suffixes of x

i	SA[i]
0	8
1	1
2	4
3	2
4	5
5	7
6	0
7	3
8	6

(b) The suffix array of x

i	iSA[i]
0	6
1	1
2	3
3	7
4	2
5	4
6	8
7	5
8	0

(c) The inverse suffix array of x

Table 2.1 The process of generating the SA and iSA data structures of x

2.5 Longest Common Prefix

Manber & Myers [73] introduced the *longest common prefix*, LCP, array of a string x as a data structure that records the longest common prefixes between two consecutive suffixes in an SA. Formally, an LCP array is defined by: $\mathsf{LCP}[r] := \mathsf{lcp}(r-1, r)$ for all $1 \leq r < n$, and $\mathsf{LCP}[0] = 0$, where $\mathsf{lcp}(r,s)$ denotes the length of the longest common prefix between the

suffixes $x[\mathsf{SA}[r]\,..\,n-1]$ and $x[\mathsf{SA}[s]\,..\,n-1]$ for positions r and s of x. The construction cost of an LCP array is $\mathcal{O}(n)$ [31].

Example 4. Given a string $x =$ `CAACAAC$` the SA and LCP arrays of string x are illustrated as below in Table 2.2, where the LCP array is composed by comparing consecutive suffixes in SA lexicographically.

i	SA[i]	Suffix
0	8	`$`
1	1	`AACAACC$`
2	4	`AACC$`
3	2	`ACAACC$`
4	5	`ACC$`
5	7	`C$`
6	0	`CAACAACC$`
7	3	`CAACC$`
8	6	`CC$`

(a) The SA of x

i	LCP[i]
0	0
1	0
2	3
3	1
4	2
5	0
6	1
7	4
8	1

(b) The LCP array of x

Table 2.2 The SA and LCP arrays of string x

2.6 Longest Common Extension

The *longest common extension* (LCE) between two suffixes of a string x starting at positions i and j is defined as the length of the longest prefix common to both suffixes. Formally, for a given string x:

$$\mathrm{LCE}(i,j) = \max\{\ell : x[i\,..\,i+\ell-1] = x[j\,..\,j+\ell-1]\}.$$

The LCE can be computed in $\mathcal{O}(1)$ [48]. We now generalise the concept of LCE to that of an LCE with k errors, which costs $\mathcal{O}(k)$. In this case, the LCE is similarly defined, i.e. as a common prefix between two suffixes, if they match with k or fewer than k errors in terms of their Hamming distance [54]. Formally, for a given string x:

$$\mathrm{LCE}_k(i,j) = \max\{\ell : x[i..i+\ell-1] \approx_k x[j..j+\ell-1]\}.$$

A symmetric construction on x^R can answer the so-called *longest common suffix* (lcs) queries with the same complexity. The lcp and lcs queries are also known as *longest common extension* (LCE) queries.

2.7 Lowest Common Ancestor

The *lowest common ancestor* (LCA) (introduced by Harel & Tarjan [37]) of two nodes, u and v, in a tree or in a directed acyclic graph (DAG), is the lowest node that has both u and v as descendants, where we define each node to be a descendant of itself. Thus, the LCA of u and v is the ancestor of u and v such that it is located farthest from the root. Figure 2.2 below shows an example of LCA (4,6) and LCA (5,7), which are 2 and 1 respectively. It is well known that the LCA query can be processed in $\mathcal{O}(1)$ [15].

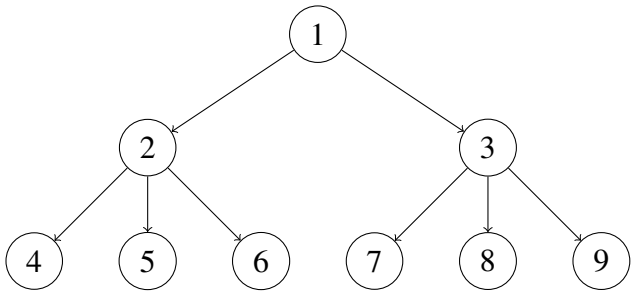


Fig. 2.3 LCA(4,6)=2, LCA(5,7)=1

2.8 Depth First Search

The *Depth First Search* (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Full technical details of the depth-first search data structure can be found in [91]. Trees can be traversed in three different ways: *Preorder*, *Inorder* and *Postorder*. An example of each of these ways of conducting a DFS search for tree T in Figure 2.4: Preorder Traversal : $(1,\ 2,\ 4,\ 5,\ 3,\ 6,\ 7)$, Inorder Traversal: $(4,\ 2,\ 5,\ 1,\ 6,\ 3,\ 7)$ and Postorder Traversal: $(4,\ 5,\ 2,\ 6,\ 7,\ 3,\ 1)$.

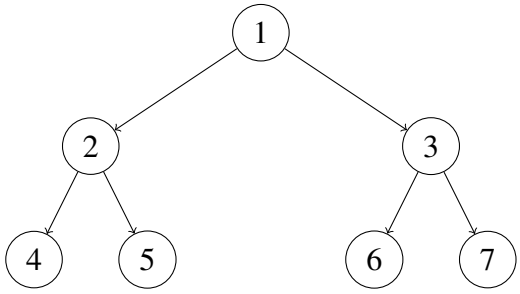


Fig. 2.4 The DFS for tree T

2.9 Breadth First Search

The *Breadth First Search* (BFS) is an algorithm invented by [78] for traversing or searching a tree or graph structure. It starts by visiting the root node and exploring all the neighbours in the same depth before moving to the next level. An example of a BFS search of a tree T is shown in Figure 2.5.

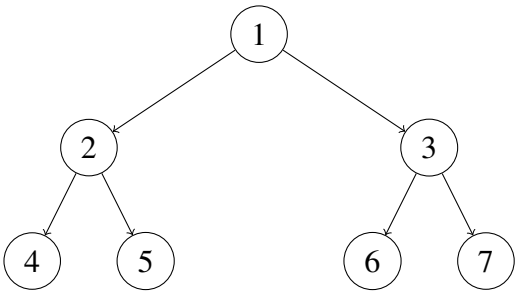


Fig. 2.5 The BFS of tree T is $(1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7)$

2.10 Deterministic Finite Automaton

The *Deterministic finite automaton* (DFA) is a finite state machine for a given input symbol that the machine can determine the next state and the number of the states is finite. The DFA can be represented formally by five tuples $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the alphabet, i.e. a finite set of symbols, δ is the transition function, $\delta : Q \text{ x } \Sigma \rightarrow Q$ is the initial state where the machine starts and F is the set of final states. See [53] for more details. Example 5 below shows a graphical representation of the DFA.

Example 5. Let $Q = \{a, b, c\}$, $\Sigma = \{0, 1\}$, $q_0 = \{a\}$, $F = \{c\}$ and $\delta : Q \text{ x } \Sigma \rightarrow Q$. The DFA will be as shown in Figure 2.6 based on the transition table (Table 2.3):

Current state	next state with input 0	next state with input 1
a	a	b
b	c	a
c	b	c

Table 2.3 The transition function δ for each state with the input 0 or 1

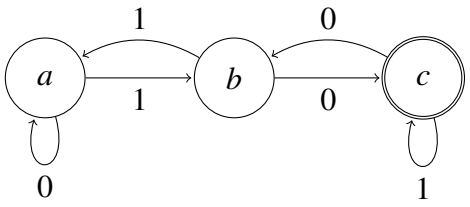


Fig. 2.6 A graphical representation of the DFA based on Q, Σ, q_0, F and δ

2.11 Non-deterministic Finite Automaton

The *Non-deterministic finite automaton* (NFA) is also a finite state machine, but the machine can move to any state with a specified input, which means that the exact next state cannot be determined. For this reason, it is called a non-deterministic finite automaton, see [75]. The NFA can be represented with a tuple of five elements, $(Q, \Sigma, \Delta, q_0, F)$. Where Q is a finite set of states, Σ is finite set of symbols of the alphabet, Δ is the transition relationship that takes a state in Q and an input symbol as arguments and returns a subset of Q, q_0 is the initial state where the machine starts and F is the set of final states [53]. An example of NFA is shown in Example 6.

Example 6. Let $Q = \{a, b, c\}$, $\Sigma = \{0, 1\}$, $q_0 = \{a\}$, $F = \{c\}$ and $\Delta : Q \text{ x } \Sigma \rightarrow Q$. The NFA will be as below in Figure 2.7 based on the transition table (Table 2.4):

Current state	next state with input 0	next state with input 1
a	a,b	b
b	c	a,c
c	b,c	c

Table 2.4 The transition function Δ for each state with the input 0 and 1

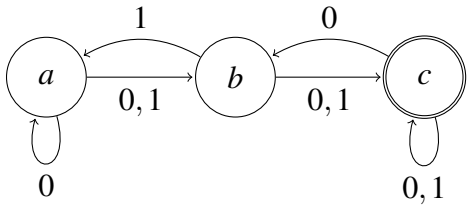
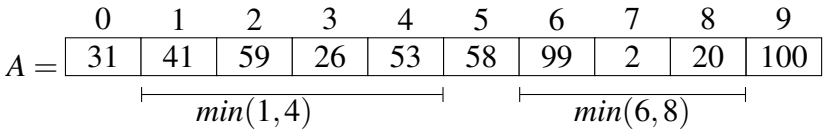


Fig. 2.7 A graphical representation of the NFA based on Q, Σ, q_0, F and Δ

2.12 Range Minimum Query

In the *Range Minimum Query* (RMQ) problem, we are given an array A of n numbers and we are asked to answer queries of the following type: for indices i and j between 0 and $n-1$, query $\text{RMQ}_A(i, j)$ returns the index of the minimum element in the subarray $A[i..j]$.

Example 7. Assume we are given array A below, the RMQ(1,4)=3, while RMQ(6,8)=7.



It is then known that a range minimum query (RMQ) data structure based on the LCP array, which can be constructed in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space [15], can answer lcp-queries in $\mathcal{O}(1)$ time per query [72].

2.13 k-mappability

In the *k-mappability* problem, we are given a string x of length n and integers m and k, and we are asked to count, for each length-m factor y of x, the number of other factors of length m of x that are at a Hamming distance at most k from y. We focus here on the version of the problem where $k = 1$. Manzini [74] published an algorithm to solve this problem for $k = 1$, requiring time $\mathcal{O}(mn\log n/\log\log n)$ using space $\mathcal{O}(n)$.

Example 8. In Table 2.5 below we show an example of the mappability of a factor with length 4 up to 0 and 1 mismatches, called 0-mappability and 1-mappability, respectively. Reviewing the factor at $x[0]$=`A T C T`, it occurs exactly two times at $x[0]$ and $x[13]$, hence 0-mappability[0]=1 and 0-mappability[13]=1, as shown in Table 2.5a. However, the same factor "`A T C T`" occurs one time at x=[4] and x=[13] with up to one mismatch, hence 1-mappability[0]=2, 1-mappability[4]=3 and 1-mappability[13]=1. Note, we do not compare the factor with itself.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x	A	T	C	T	A	G	C	T	T	C	C	T	A	A	T	C	T
0-mappability	**1**	0	0	0	0	0	0	0	0	0	0	0	0	**1**			

(a) $(0,4)$-mappability

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x	A	T	C	T	A	G	C	T	T	C	C	T	A	A	T	C	T
1-mappability	**2**	1	2	1	**3**	1	1	1	1	2	0	0	0	**1**			

(b) (1,4)-mappability

Table 2.5 (k,m)-mappability table for m=4 and $k \in (0,1)$

2.14 Degenerate String Comparison and Applications

A *generalised degenerate string* (GD string) $\hat{S}$ over Σ is a sequence of n sets of strings over Σ of total size N, where the ith set contains strings of the same length $k_i > 0$, but where this length can vary between different sets. We denote the sum of these lengths $k_0, k_1, \ldots, k_{n-1}$ by W. Thus a GD string can be used to represent a *gapless* multiple sequence alignment (MSA) of fixed width, that is, for example, a high-scoring local alignment of multiple sequences, in a compact form; see Figure 2.8. This type of alignment is used for finding *functional* sequence elements [36]. For instance, searching for palindromic motifs in these types of alignments is an important problem since many transcription factors bind as homodimers to palindromes [76]. Specifically, a set of virus species can be clustered using high-scoring MSA to obtain subsets of viruses that have a *common* hairpin structure [80].

```
CA--AGCTCTATCTCGTA--TT
C---AGCCGAAGCTCGTATATT
CATCAAGTCAACGCAG----TT
```

(a) Multiple sequence alignment

```
AGCTCTATCTCG
AGCCGAAGCTCG
AAGTCAACGCAG
```

(b) Local gapless alignment

$$\hat{S} = \{\mathtt{A}\} \cdot \left\{ \begin{matrix} \mathtt{GC} \\ \\ \mathtt{AG} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \mathtt{TCT} \\ \\ \mathtt{CGA} \\ \\ \mathtt{TCA} \end{matrix} \right\} \cdot \left\{ \mathtt{A} \right\} \cdot \left\{ \begin{matrix} \mathtt{TCTC} \\ \\ \mathtt{GCTC} \\ \\ \mathtt{CGCA} \end{matrix} \right\} \cdot \left\{ \mathtt{G} \right\}$$

(c) GD string obtained from the local gapless alignment

Fig. 2.8 A GD string representing a gapless multiple sequence alignment.

2.15 k-closed Strings

A *closed string* is a string that has u both as a prefix and as a suffix but not elsewhere in the string. Closed strings were introduced by Fici [30] as objects of combinatorial interest.

If a string b is both a proper prefix and a proper suffix of a non-empty string x, then b is called a *border* of x. A string x is said to be *closed* if, and only if, it is empty or if there exists a border b of x that occurs exactly twice in x (i.e. only as a prefix and suffix). In other words, b satisfies (1) $b = x[0..|b|-1] = x[|x|-|b|..|x|-1]$ and (2) $b \neq x[i..i+|b|-1]$, for all $1 \leq i \leq |x|-|b|-1$. If x is closed, we call such a b the *closed border* of x. We additionally define the special case of a single letter $a \in \Sigma$ to be closed, with the empty string ε as the border of a.

Example 9. In Figure 2.9 string `AAACCGTAAA` is closed, since the factor `AAA` occurs only as a prefix and as a suffix. The string `AAACAAAGTAAA`, on the contrary, is not closed since `AAA` occurs within the string as well as at the start and end.

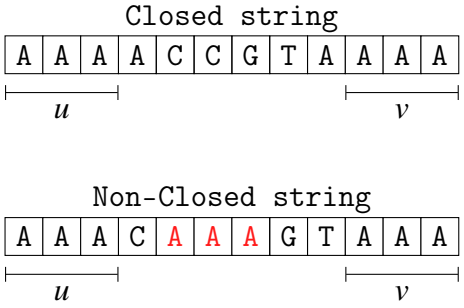


Fig. 2.9 Closed string and non-closed string with a border length of 3

Chapter 3

k-Mappability

The work presented in this chapter is published as: M. Alzamel, P. Charalampopoulos, C. Iliopoulos, S. P. Pissis, J. Radoszewski, and W.-K. Sung. Faster algorithms for 1-mappability of a sequence. *Theoretical Computer Science*. 812: 2-12 (2020)

3.1 Background and Contributions

3.1.1 Background

The focus of this work is directly motivated by the well-known and challenging application of *genome re-sequencing*—the assembly of a genome directed by a reference sequence. New developments in sequencing technologies [77] allow whole-genome sequencing to be turned into a routine procedure, creating massive amounts of sequencing data. Short sequences, known as *reads*, are produced in huge amounts (tens of gigabytes); and in order to determine the part of the genome from which a read was derived, it must be mapped (aligned) back

to some reference sequence that consists of a few gigabases. A wide variety of short-read alignment techniques and tools have been published in the past years to address the challenge of efficiently mapping tens of millions of reads to a genome, focusing on different aspects of the procedure: speed, sensitivity and accuracy [34]. These tools allow for a small number of errors in the alignment. The *k-mappability* problem was first introduced in the context of genome analysis in [26] (and in some sense earlier in [9]), where a heuristic algorithm was proposed to approximate the solution. The aim from a biological perspective is to compute the mappability of each region of a genome sequence; i.e. for every factor of a given length of the sequence, we are asked to count how many other times it occurs in the genome with up to a given number of errors. This is particularly useful in the application of genome re-sequencing. By computing the mappability of the reference genome, we can then assemble the genome of an individual with greater confidence by first mapping the segments of the DNA that correspond to regions with low mappability. Interestingly, it has been shown that genome mappability varies greatly between species and gene classes [26]. Formally, we are given a string x of length n and integers $m < n$ and $k < m$, and we are asked to count, for each length-m factor y of x, the number of other length-m factors of x that are at a Hamming distance of at most k from y.

Example 10. Consider the string $x = \texttt{AACAAACCCC}$ and $m = 3$. Table 3.1 shows the k-mappability counts for $k = 0$ and $k = 1$.

For instance, consider the position 0. Here, the 0-mappability is 1, since the factor `AAC` occurs also at position 4. The 1-mappability at this position is 3 due to the occurrence of

position	0	1	2	3	4	5	6	7
factor occurrence	AAC	ACA	CAA	AAA	AAC	ACC	CCC	CCC
0-mappability	1	0	0	0	1	0	1	1
1-mappability	3	2	1	4	3	5	2	2

Table 3.1 The mappability table for x=AACAAACCCC up to 0 and 1 mismatches

AAC at position 4 and occurrences of two factors at Hamming distance 1 from AAC: AAA at position 3 and ACC at position 5.

The 0-mappability problem can be solved in $\mathcal{O}(n)$ time with the well-known LCP data structure [31]. For $k = 1$, to the best of our knowledge, the fastest known algorithm is that of Manzini [74]. His solution runs in $\mathcal{O}(mn\log n/\log\log n)$ time and $\mathcal{O}(n)$ space and works only for strings over a constant-sized alphabet. Since the problem for $k = 0$ can be solved in $\mathcal{O}(n)$ time, one may focus on counting, for each length-m factor y of x, the number of other factors of x that are at Hamming distance *exactly* 1 — instead of at most 1 — from y.

3.1.2 Contributions

(a) We present an algorithm that, given a string x of length n over an integer alphabet of size $\sigma > 1$ and a positive integer $m = \Omega(\log_\sigma n)$, solves the 1-mappability problem for x in average-case time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$. Notably, we show that this algorithm is generalisable for arbitrary k requiring average-case time $\mathcal{O}(kn)$ and space $\mathcal{O}(n)$ if $m = \Omega(k\log_\sigma n)$. Here, we assume that the letters are independently and uniformly distributed random variables.

(d) We provide an open-source implementation of our average-case algorithm for arbitrary k, and also experimental results demonstrating that it is competitive with the state-of-the-art implementation for the same problem [26].

3.2 Preliminaries and Definitions

Let y be a string of length m with $0 < m \leq n$. We say that an *occurrence* of y exists in x, or, more simply, that y *occurs in* x, when y is a factor of x. Every occurrence of y can be characterised by a starting position in x. Thus, we say that y occurs at the *starting position* i in x when $y = x[i..i+m-1]$. The scope of the computational problem can be formally stated as follows.

1-MAPPABILITY
Input: A string x of length n and an integer m, where $1 \leq m < n$
Output: An integer array C of size $n-m+1$ such that $C[i]$ stores the number of factors of x that are at a Hamming distance of 1 from $x[i..i+m-1]$

.

3.3 Efficient Average-Case Algorithm

In this section we assume that x is a string derived from an integer alphabet Σ. For clarity of presentation, we first describe the algorithm for $k = 1$ and then show how it can be generalised for arbitrary k. Recall that if two strings, y and z, are at a Hamming distance of 1, we write $y \approx_1 z$.

Fact 1 (Pigeonhole principle)**.** Given two strings, y and z, of length m, we have that if $y \approx_1 z$, then y and z share at least one factor of length $\lfloor m/2 \rfloor$.

Example 11. In Figure 3.1, given two strings, x and z, of length $m = 8$, $x \approx_1 z$ and at least they share a factor of length $\lfloor 8/2 \rfloor = 4$. In this example there are two cases: x and z share exactly block 1 and share block 2 with one mismatch or they share exactly block 2 and block 1 with one mismatch.

	BLOCK 1	BLOCK 2	BLOCK 1	BLOCK 2
x	ACGT	CCCC	ACGT	CCCC
z	ACGT	CCCT	ACGC	CCCC

Fig. 3.1 Pigeonhole principle of x and z with one mismatch

Fact 2. Given a string x, and any two positions i, j on x, we have that if $x[i..i+m-1] \approx_1 x[j..j+m-1]$, then $x[i..i+m-1]$ and $x[j..j+m-1]$ have at least one common factor of length $L = \lfloor m/3 \rfloor$ starting at positions $i' \in \{i, \ldots, i+m-L\}$ and $j' \in \{j, \ldots, j+m-L\}$ of x, such that $i' - i = j' - j$ and $i' = 0 \pmod{L}$.

Example 12. As shown in Figure 3.2, given a string $x = \ldots$ A C G T A C C C C A... A C G T C C C C C A..., and given that $m = 10$, $x[i \ldots i+10-1]$ and $x[j \ldots j+10-1]$ share exactly two blocks of length $L = 3$, which are A C G and C C C.

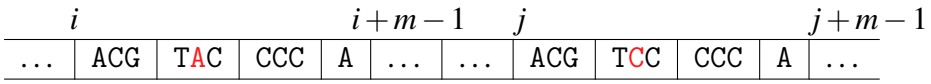


Fig. 3.2 Two substrings share at least one common factor length $L = 3$

Proof. It should be clear that every factor of x of length m fully contains at least two factors of length L starting at positions equal to 0 mod L. Then, if $x[i..i+m-1]$ and $x[j..j+m-1]$ are at a Hamming distance of 1, analogously to Fact 1, at least one of the two factors of length L that are fully contained in $x[i..i+m-1]$ occurs at a corresponding position in $x[j..j+m-1]$; otherwise we would have a Hamming distance greater than 1. □

We first initialise an array C of size $n-m+1$, with 0 in all positions. For all i, $C[i]$ will eventually store the number of factors of x that are at a Hamming distance of 1 from $x[i..i+m-1]$. We apply Fact 2 by implicitly splitting the string x into $B = \lfloor \frac{n}{\lfloor m/3 \rfloor} \rfloor$ *blocks* of length $L = \lfloor m/3 \rfloor$—the suffix of length n mod $\lfloor m/3 \rfloor$ is not taken as a block—starting at the positions of x that are equal to 0 mod L. In order to find all pairs of length-m factors that are at a Hamming distance of 1 from each other, we can find all the exact matches of every block and try to extend each of them to the left and to the right, allowing at most one mismatch. We need to tackle some technical details if we are to update our counters correctly and avoid double counting, however.

We start by constructing the SA and LCP arrays for x in $\mathcal{O}(n)$ time. We also construct RMQ data structures over the LCP arrays in order to answer LCE queries in a constant time per query. By exploiting the LCP array information, we can then find in $\mathcal{O}(n)$ time all maximal sets of indices, such that the longest common prefix between any two of the suffixes

starting at these indices is at least L, and at least one of them is the starting position of some block.

Example 13. Given a string $x =$ `A C G T A C C C C A A C G T C C C C C A C C G T A C C C C A`, and given that $m = 10$ and $k = 1$, the SA and LCP will be built as shown in Table 3.2, while $L = \lfloor 10/3 \rfloor = 3$. In Table 3.2 below, we exploit the LCP array and mark the starting positions of each block, which are $x[i] = 6$, $x[i] = 15$, $x[i] = 21$, $x[i] = 12$ and $x[i] = 3$, respectively. Each marked block value LCP[i], will be grouped with LCP[i-1] and its following LCP[j] values, if $\mathsf{LCP}[j] \geq L$.

i	$x[i]$	SA[i]	LCP[i]	
0	29	A	0	
1	9	AACGTCCCCCACCGTACCCCA	1	
2	24	ACCCCA	1	
3	4	ACCCCAACGTCCCCCACCGTACCCCA	6	
4	19	ACCGTACCCCA	3	
5	0	ACGTACCCCAACGTCCCCCACCGTACCCCA	2	
6	10	ACGTCCCCCACCGTACCCCA	4	
7	28	CA	0	
8	8	CAACGTCCCCCACCGTACCCCA	2	
9	18	CACCGTACCCCA	2	
10	27	CCA	1	
11	7	CCAACGTCCCCCACCGTACCCCA	3	
12	17	CCACCGTACCCCA	3	
13	26	CCCA	2	
14	**6**	CCCAACGTCCCCCACCGTACCCCA	4	
15	16	CCCACCGTACCCCA	4	
16	25	CCCCA	3	Group 1
17	5	CCCCAACGTCCCCCACCGTACCCCA	5	
18	**15**	CCCCACCGTACCCCA	5	
19	14	CCCCCACCGTACCCCA	4	
20	20	CCGTACCCCA	2	
21	**21**	CGTACCCCA	1	
22	1	CGTACCCCAACGTCCCCCACCGTACCCCA	9	Group 2
23	11	CGTCCCCCACCGTACCCCA	3	
24	22	GTACCCCA	0	
25	2	GTACCCCAACGTCCCCCACCGTACCCCA	8	Group 3
26	**12**	GTCCCCCACCGTACCCCA	2	
27	23	TACCCCA	0	
28	**3**	TACCCCAACGTCCCCCACCGTACCCCA	7	Group 4
29	13	TCCCCCACCGTACCCCA	1	

Table 3.2 The SA and LCP arrays of string x

In group 1, the starting block is at LCP[14] and this shares at least one block with LCP[15]...[19] and LCP[13]. In group 2 the starting position is at LCP[21] and this shares at least one block with LCP[20], LCP[22] and LCP[23]. This is the case also for groups 3 and 4. Figure 3.3 shows the longest common prefixes between LCP[18] and LCP[17], which is `C C C C A`, and they all share shares one full size block and at least one of them is the starting position of some block.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
x[i]	A	C	G	T	A	C	C	C	C	A	A	C	G	T	C	C	C	C	C	A

i	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
x[i]	A	C	G	T	C	C	C	C	C	A	C	C	G	T	A	C	C	C	C	A

Fig. 3.3 The longest common prefix between LCP[18] and LCP[17].

Then for each such set, denoted by P, we have to go through the following procedure for each index $i \in P$ such that $i = 0 \pmod{L}$. For every other $j \in P$, we try to extend the match by asking two LCE queries in each direction. i.e., we ask an $\mathsf{lcs}(i-1, j-1)$ query to find the first mismatch positions, ℓ_1 and ℓ'_1, respectively, and then $\mathsf{lcs}(\ell_1 - 1, \ell'_1 - 1)$ to find the second mismatch positions, ℓ_2 and ℓ'_2, respectively. A symmetrical procedure computes the mismatches r_1, r'_1 and r_2, r'_2 to the right, as shown in Figure 3.4. We omit here some technical details with regards to reaching the start or end of x.

Now we are interested in positions p such that $\ell_2 < p \leq \ell_1$ and $i + L - 1 \leq p + m - 1 < r_1$ and positions q such that $\ell_1 < q \leq i$ and $r_1 \leq q + m - 1 < r_2$. Each such position p (resp. q)

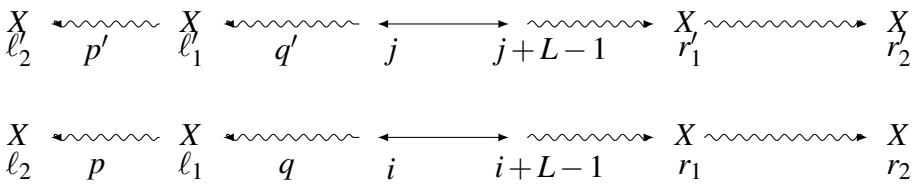


Fig. 3.4 Performing two LCE queries in each direction.

implies that $x[p\,..\,p+m-1] \approx_1 x[p'\,..\,p'+m-1]$, where $p' = j-(i-p)$. Henceforth, we only consider positions of the type p, p'.

Note that, if $x[p\,..\,p+m-1] \approx_1 x[p'\,..\,p'+m-1]$, we will identify the unordered pair $\{p, p'\}$ based on the described approach $t_{p,p'}$ times, where $t_{p,p'}$ is the total number of full blocks contained in $x[p\,..\,p+m-1]$ and in $x[p'\,..\,p'+m-1]$ after the mismatch position. It is not hard to compute the number $t_{p,p'}$ in $\mathcal{O}(1)$ time, based on the starting positions p and p', as well as ℓ_1 and r_1 each time we identify $x[p\,..\,p+m-1] \approx_1 x[p'\,..\,p'+m-1]$. To avoid double counting, we then increment the $C[p]$ and $C[p']$ counters by $1/t_{p,p'}$.

By $\mathsf{EXT}_{i,j}$ we denote the time required to process a pair of elements i, j of a set P such that at least one of them, i or j, equals 0 mod L.

Example 14. Let us consider the same positions in Example 13 ($\mathsf{LCP}[17]$ and $\mathsf{LCP}[18]$). In Figure 3.5, we show that ℓ_1, ℓ_1' represent `A` and `C` respectively, and that r_1, r_1', r_2 and r_2' represent `A`, `C`, `C` and `A` consecutively.

Lemma 1. The time $\mathsf{EXT}_{i,j}$ is $\mathcal{O}(m)$.

Proof. Given $i, j \in P$, with at least one of them being equal to 0 mod L, we can find the pairs (p, p') of positions that satisfy the inequalities discussed above in $\mathcal{O}(m)$ time. They are a subset of $\{(i-m+L, j-m+L), \ldots, (i-1, j-1)\}$. For each such pair (p, p'), we

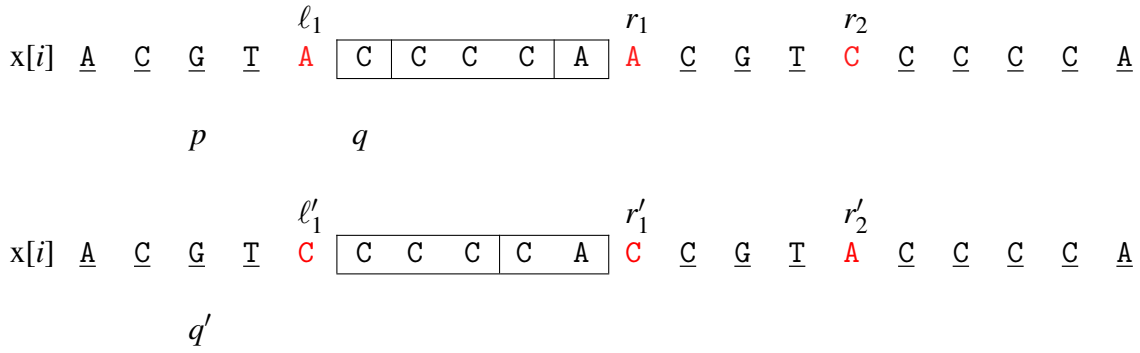


Fig. 3.5 Performing $\mathsf{EXT}_{i,j}$ for $x[5]$ and $x[15]$

can compute $t_{p,p'}$ and increment $C[p]$ and $C[p']$ accordingly in $\mathscr{O}(1)$ time. The total time to process all pairs (p,p') for a given i,j is thus $\mathscr{O}(m)$. □

It should be clear that the aforementioned algorithm is generalisable for arbitrary k. We now proceed to prove the following theorem.

Theorem 1. Given a string x of length n derived from an integer alphabet Σ of size $\sigma > 1$ with the letters of x being independently and identically distributed random variables, uniformly distributed over Σ, the k-mappability problem can be solved in average-case time $\mathscr{O}(kn)$ and space $\mathscr{O}(n)$ if $m \geq (k+2)\cdot(\log_\sigma n+1)$.

Proof. The time and space required for constructing the SA and LCP array for x and $\mathsf{rev}(x)$ and the RMQ data structures over the LCP arrays is $\mathscr{O}(n)$.

Let B denote the number of blocks over x and L be the block length. We set

$$L = \lfloor \tfrac{m}{k+2} \rfloor, \quad B = \lfloor \tfrac{n}{L} \rfloor$$

to apply the pigeon-hole principle: at least one block must be an exact match (generalisation of Fact 2). Recall that by P we denote a maximal set of indices of the LCP array such that the length of the longest common prefix between any two suffixes starting at these indices is at least L, and at least one of them is the starting position of some block. Processing all such sets P requires time

$$\mathsf{EXT}_{i,j} \cdot Occ$$

where $\mathsf{EXT}_{i,j}$ is the time required to process a pair i, j of elements of a set P; and Occ is the sum of the multiples of the cardinality of each set P times the number of the elements of set P that are equal to 0 mod L. We generalise Lemma 1 for arbitrary k, showing that $\mathsf{EXT}_{i,j} = \mathscr{O}(m)$ as follows. We perform at most $2k+2$ longest common extension queries (to the left and to the right); list all $\mathscr{O}(k)$ blocks that do not contain a mismatch within these extensions; and then consider $\mathscr{O}(m)$ positions to be updated. Additionally, by the stated assumption on the string x, the expected value for Occ is no more than $\frac{Bn}{\sigma^L}$. Hence, the algorithm on average requires time

$$\mathscr{O}(n + m \cdot \frac{B \cdot n}{\sigma^L}).$$

Let $m = (k+2)q + r$, for $0 \leq r \leq k+1$, $q \geq 1$; note that here we assume that $m \geq k+2$; further note that $\lfloor m/(k+2) \rfloor = q$. If q satisfies $n \leq \sigma^q$ we have

$$m \cdot \frac{B}{\sigma^L} = \frac{m \cdot \lfloor \frac{n}{\lfloor m/(k+2) \rfloor} \rfloor}{\sigma^{\lfloor \frac{m}{k+2} \rfloor}} = \frac{m \cdot \lfloor \frac{n}{q} \rfloor}{\sigma^q} \leq \frac{m \cdot \frac{n}{q}}{\sigma^q} \leq \frac{m}{q} = \frac{(k+2)q + r}{q}$$

$$= k + 2 + \frac{r}{q} \leq 2k + 3.$$

Consequently, in the case when

$$m \geq (k+2) \cdot (\log_\sigma n + 1)$$

we have that

$$m\frac{B \cdot n}{\sigma^L} \leq (2k+3)n$$

and hence the algorithm requires $\mathcal{O}(kn)$ time on average. The extra space usage is $\mathcal{O}(n)$. □

We thus obtain the following corollary with respect to the *1-mappability* problem; namely, for $k = 1$.

Corollary 1. Given a string x of length n derived from an integer alphabet Σ of size $\sigma > 1$ with the letters of x being independently and identically distributed random variables, uniformly distributed over Σ, the *1-mappability* problem can be solved in average-case time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$ if $m \geq 3 \cdot \log_\sigma n + 3$.

3.4 Implementation

We have implemented the average-case algorithm described in Section 3.3 as a program to compute the mappability values. The program has been implemented in the `C++` programming language and developed under the GNU/Linux operating system. Our open-source

implementation is made available at https://github.com/maialzamel/k-map under the GNU General Public License.

Our task in this section is to evaluate the performance of our implementation with respect to the performance of the implementation provided in [26]. We call our implementation `k-map` and that of [26] `Gemtool`. Let us stress, however, that `Gemtool` is a heuristic algorithm as opposed to `k-map`, which is an exact algorithm: it always returns the correct solution.

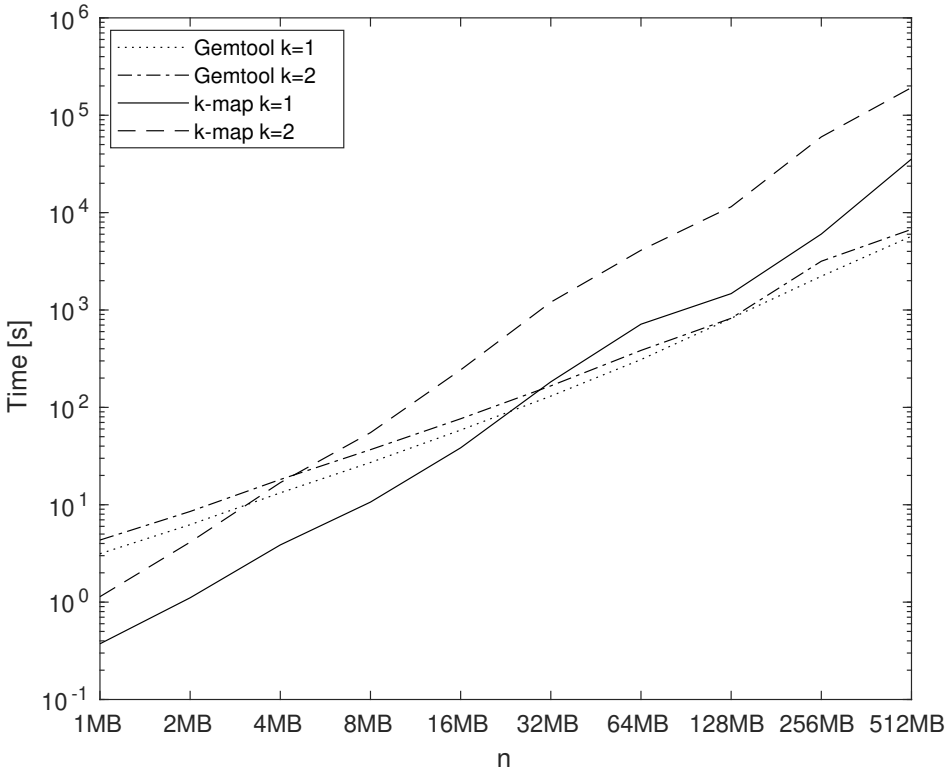
As input we used sequences extracted from a real DNA corpus ranging in length from 1MB to 512MB. This DNA corpus is available at http://pizzachili.dcc.uchile.cl/texts/dna/. For each input sequence we used different values for m and k. All experiments were conducted on a Desktop PC using one core of an Intel Core CPU i5-4690 at 3.50GHz. Both implementations were compiled with g++ version 6.2.0 at optimisation level 3 (-O3).

The experimental results (recorded elapsed times and memory usage) are depicted in Figure 3.6 and Figure 3.7:

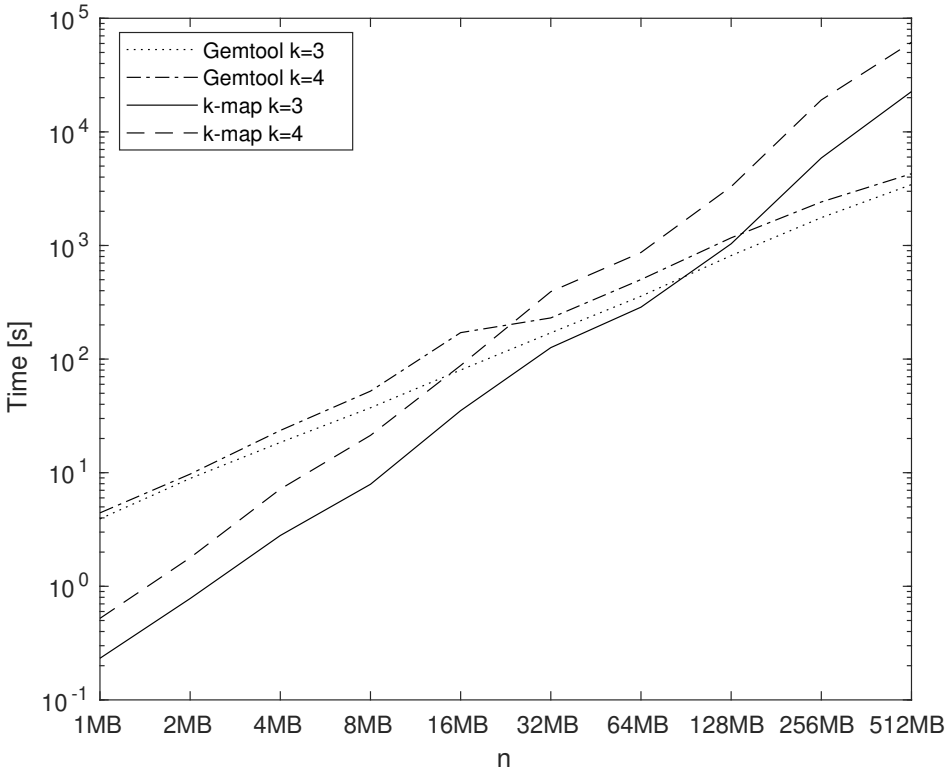
1. For fixed values of k and m, our implementation requires time linear in n up until a certain value of n (see Theorem 1—notice that the restriction is not exactly the one stated as the input is not uniformly random). After that n value, the performance of `k-map` starts approaching the performance of `Gemtool`, which eventually becomes faster.

2. For fixed values of n, our implementation becomes considerably faster with increasing values of m (see Theorem 1).

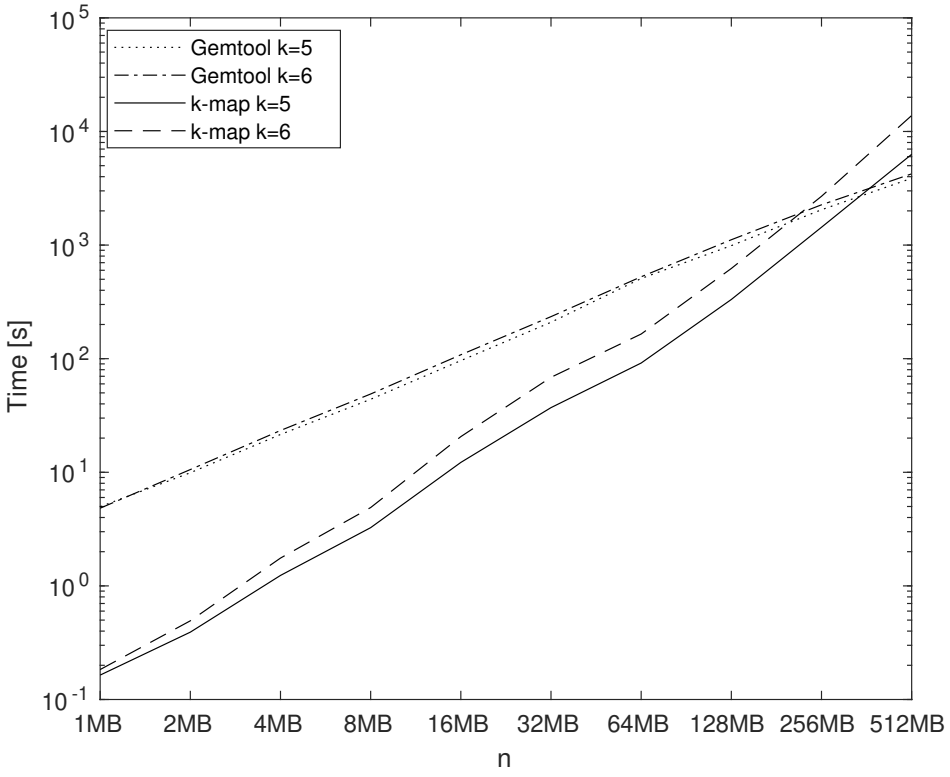
3. The memory usage of our implementation grows linearly with n (see Theorem 1). The memory usage of `Gemtool` also grows linearly with n but with a lower constant factor.



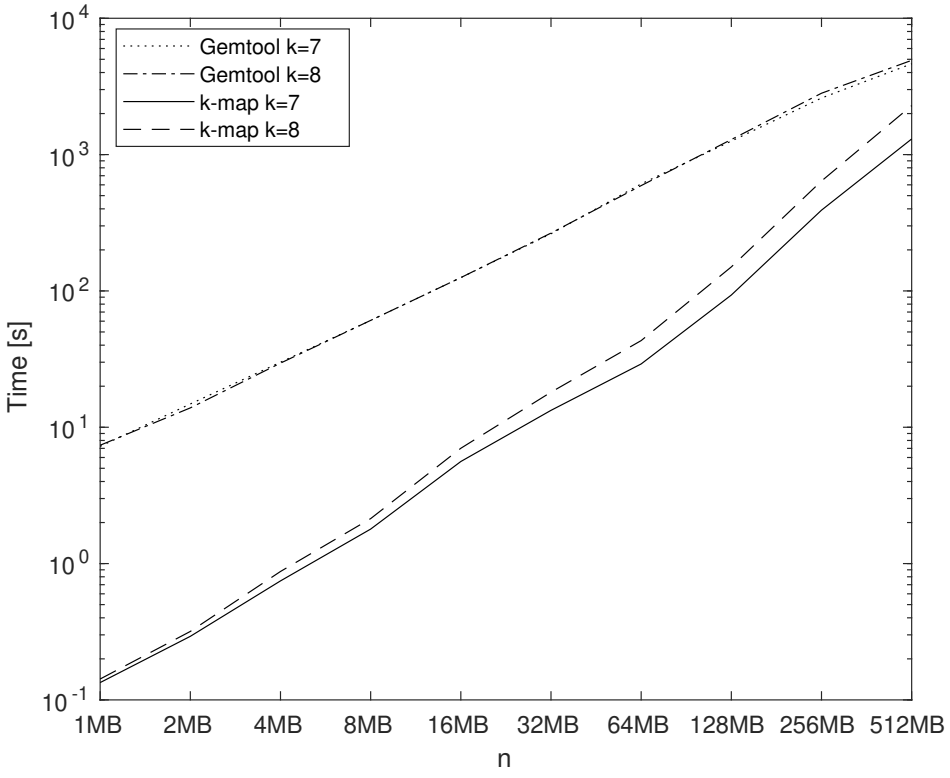
(a) Elapsed time for $m = 32$ and $k = 1, 2$



(b) Elapsed time for $m = 64$ and $k = 3, 4$

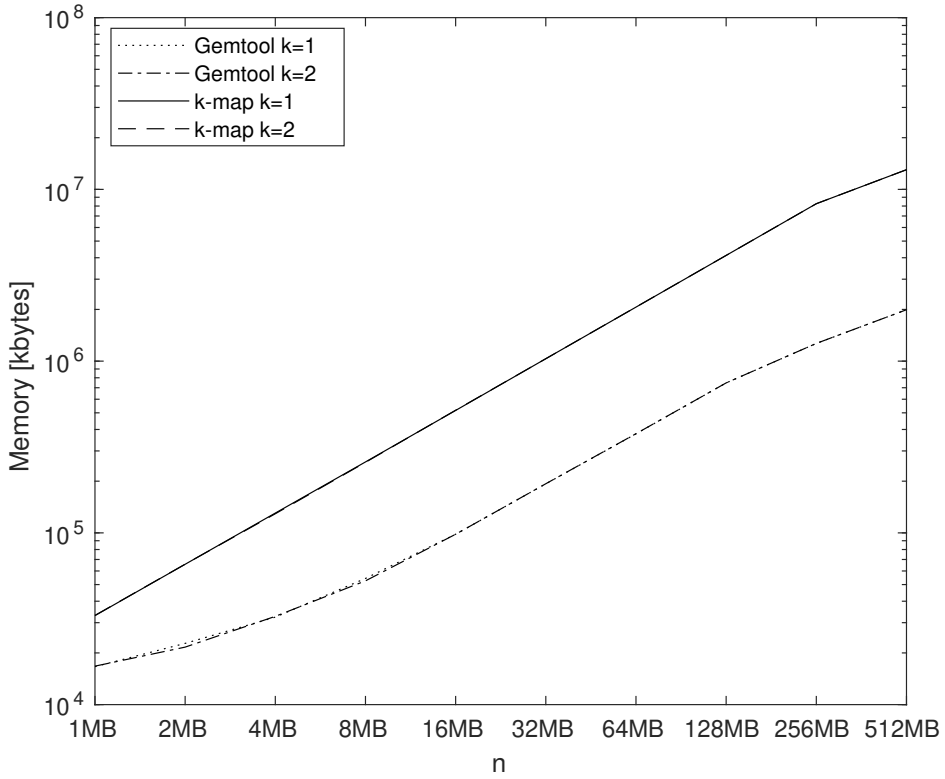


(c) Elapsed time for $m = 128$ and $k = 5, 6$

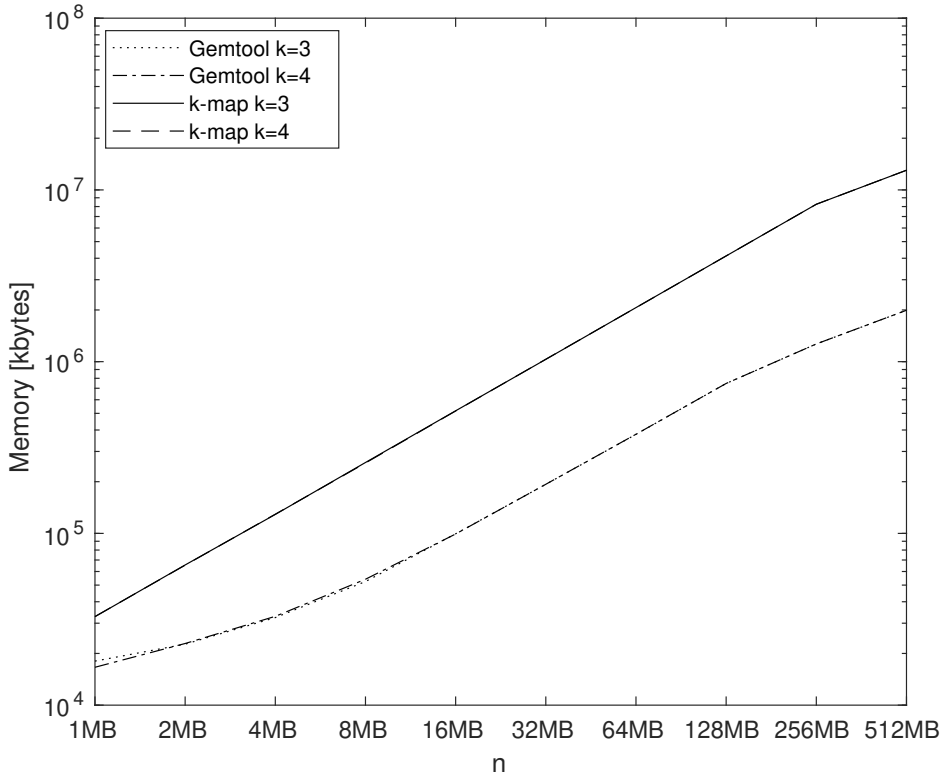


(d) Elapsed time for $m = 256$ and $k = 7, 8$

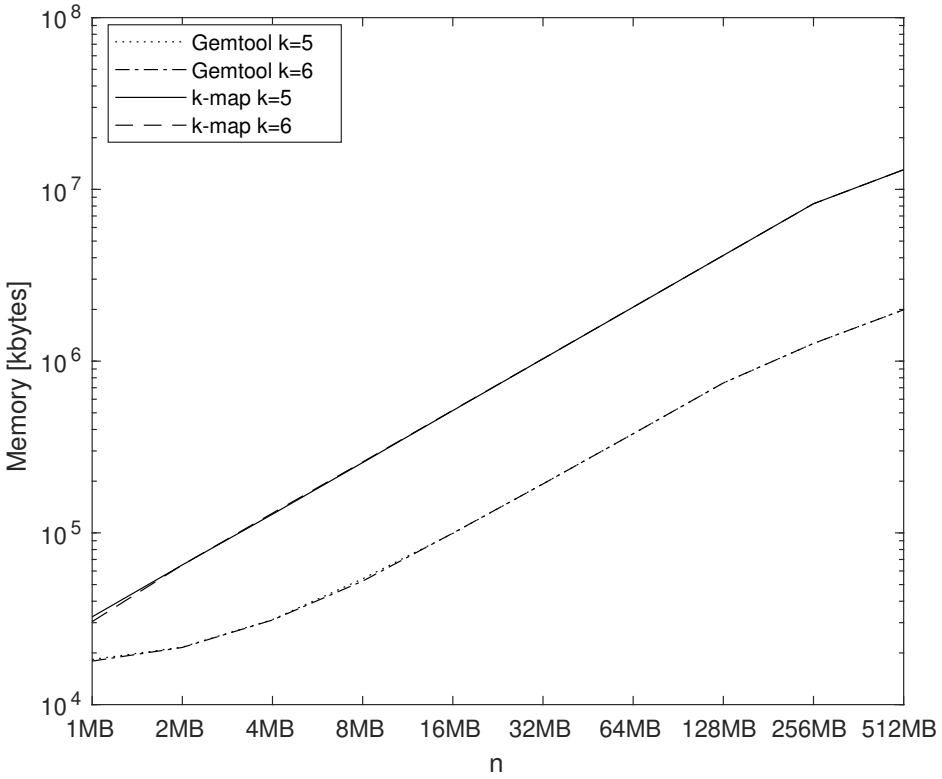
Fig. 3.6 Elapsed-time comparison between `k-map` and `Gemtool`.



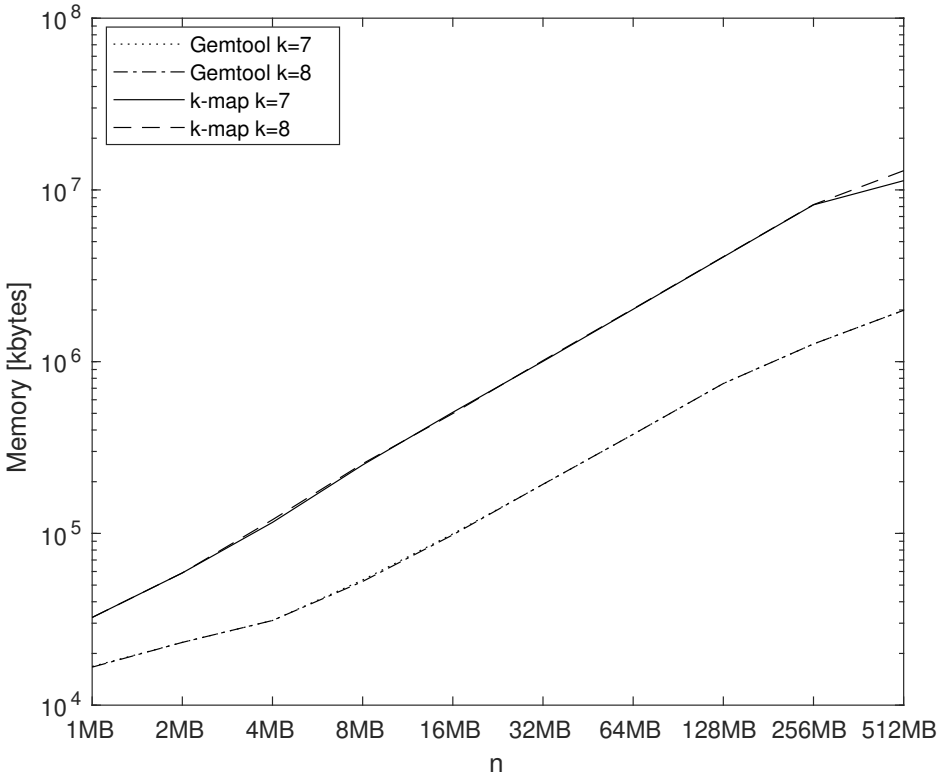
(a) Memory usage for $m = 32$ and $k = 1, 2$



(b) Memory usage for $m = 64$ and $k = 3, 4$



(c) Memory usage for $m = 128$ and $k = 5, 6$



(d) Memory usage for $m = 256$ and $k = 7, 8$

Fig. 3.7 Memory-usage comparison between `k-map` and `Gemtool`.

3.5 Conclusion

In this chapter, we investigated the special case of $k = 1$. We presented an algorithm that requires average-case time and space $\mathcal{O}(n)$ for integer alphabets of size σ if $m = \Omega(\log_\sigma n)$, and showed that this algorithm is generalisable for arbitrary k, requiring average-case time $\mathcal{O}(kn)$ and space $\mathcal{O}(n)$ if $m = \Omega(k \log_\sigma n)$. We have provided an open-source implementation of this algorithm and also experimental results demonstrating its competitiveness with respect to the state-of-the-art implementation [26]. We also presented an algorithm that requires $\mathcal{O}(\min\{nm, n \log n \log\log n\})$ time and $\mathcal{O}(n)$ space for this special case, see Appendix A. Later, Alzamel et al. [6] showed that the k-mappability problem can be solved in $\mathcal{O}(\min\{nm^k, n \log^{k+1} n\})$ time and $\mathcal{O}(n)$ space for $k = \mathcal{O}(1)$ and constant-sized alphabets. Let us note that it seems possible to apply the technique of Thankachan et al. [93] to obtain $\mathcal{O}(\min\{nm, n \log n\})$ time and $\mathcal{O}(n)$ space for $k = 1$ (for a preliminary exposition of the ideas, see [52]). We leave as an open question whether a $o(n \log n)$-time algorithm exists for the 1-*mappability* problem.

Another approach to considering the k-mappability problem is the edit distance model. In this model, a decision needs to be made as to whether factors only of length exactly m or of all lengths between $m-k$ and $m+k$ should be counted. Later in [86] we present **GenMap**, a more practical algorithm to compute the mappability of genomes up to k errors, which is based on the C++ sequence analysis library SeqAn library [88]. This is significantly faster, often by a magnitude, than the algorithm from the widely used *GEM suite* in [26] while refraining from approximations.

Chapter 4

Degenerate String Comparison and Applications

The work presented in this chapter is published as: M. Alzamel, L. A. K. Ayad, G. Bernardini, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, G. Rosone: Comparing Degenerate Strings. Fundam. Informaticae 175(1-4): 41-58 (2020)

4.1 Background and Contributions

4.1.1 Background

A *degenerate string* (or indeterminate string) over an alphabet Σ is a sequence of subsets of Σ. A great deal of research has been conducted on degenerate strings (see [1, 24, 58, 84, 90] and references therein). These types of uncertain sequences have been used extensively for flexible modelling of DNA sequences known as IUPAC-encoded DNA sequences [61].

In [57], the authors introduced a general definition of degenerate strings: an *elastic-degenerate string* (ED string) $\tilde{S}$ over Σ is a sequence of subsets of Σ^* (see also network expressions [81]) with the aim of representing multiple genomic sequences [22]. That is, any set of $\tilde{S}$ does not contain, in general, only letters; it may also contain strings, including the empty string. In a few recent papers on this notion, the authors provided several algorithms for pattern matching; specifically, for finding all exact [47] and approximate [19] occurrences of a standard string pattern in an ED text; see Figure 4.1.

CTCTCTAAATAATCTCG

CC--CTAAATAAGCTCG

CTC-CTAAATAACGCAG

CTC-CTAAATAA----G

$$\hat{S} = \cdot\{\mathtt{C}\}\cdot\left\{\begin{matrix}\mathtt{TCT}\\ \mathtt{C}\\ \mathtt{TC}\end{matrix}\right\}\cdot\{\mathtt{CTAAATA}\}\cdot\{\mathtt{A}\}\cdot\left\{\begin{matrix}\mathtt{TCTC}\\ \mathtt{GCTC}\\ \mathtt{CGCA}\\ \varepsilon\end{matrix}\right\}\cdot\{\mathtt{G}\}$$

Fig. 4.1 A set of sequences can be compacted to an ED string $\hat{S}$

We introduce here another special type of uncertain sequence, called a generalised degenerate string; this can be viewed as an extension of degenerate strings or as a restricted variant of ED strings. Formally, in a *generalised degenerate string* (GD string) $\hat{S}$ derived from Σ is a sequence of n sets of strings derived from Σ of total size N, where the ith set contains strings of the same length $k_i > 0$, but this length can vary between different sets.

We denote the sum of these lengths $k_0, k_1, \ldots, k_{n-1}$ by W. Thus a GD string can be used to represent a *gapless* multiple sequence alignment (MSA) of fixed width; that is, for example, a high-scoring local alignment of multiple sequences, in a compact form; see Figure 4.2.

AGCTCTATCTCG

AGCCGAAGCTCG

AAGTCAACGCAG

(a) Local gapless alignment

$$\hat{S} = \{\mathtt{A}\} \cdot \left\{\begin{matrix}\mathtt{GC}\\ \mathtt{AG}\end{matrix}\right\} \cdot \left\{\begin{matrix}\mathtt{TCT}\\ \mathtt{CGA}\\ \mathtt{TCA}\end{matrix}\right\} \cdot \{\mathtt{A}\} \cdot \left\{\begin{matrix}\mathtt{TCTC}\\ \mathtt{GCTC}\\ \mathtt{CGCA}\end{matrix}\right\} \cdot \{\mathtt{G}\}$$

(b) GD string obtained from the local gapless alignment

Fig. 4.2 A GD string representing a gapless multiple sequence alignment.

This type of alignment is used for finding *functional* sequence elements [36]. For instance, searching for palindromic motifs in these type of alignments is an important problem since many transcription factors bind as homodimers to palindromes [76]. Specifically, a set of virus species can be clustered using high-scoring MSA to obtain subsets of viruses that have a common hairpin structure [80].

Our motivation for this paper comes from finding palindromes in these types of uncertain sequences. Let us start off with standard strings. A palindrome is a sequence that reads the same from left to right and from right to left.

Detection of palindromic factors in texts is a classical and well-studied problem in algorithms on strings and combinatorics on words, with a lot of variants arising out of different practical scenarios. Palindromic sequences have been extensively studied in molecular

biology, for instance, where they are of interest because they are often distributed around promoters, introns and untranslated regions, playing important roles in gene regulation and other cell processes (e.g. see [5]). In particular, these are strings of the form $x\bar{x}^R$, also known as complemented palindromes, occurring in single-stranded DNA or, more commonly, in RNA, where x is a string and $\bar{x}^R$ is the reverse complement of x. In DNA, `C` and `G` are complements of each other (`C`-`G` are complements) and `A` and `T` are complements of each other (`A`-`T` are complements); in RNA, `C`-`G` are complements and `A`-`U` are complements.

A string $x = x[0]x[1]\ldots x[n-1]$ is said to have an initial palindrome of length k, if its prefix of length k is a palindrome. Manacher first discovered an on-line algorithm that finds all initial palindromes in a string [71]. Gusfield presented an off-line linear-time algorithm to find all maximal palindromes in a string and also discussed the relationship between biological sequences and gapped palindromes [49].

For uncertain sequences, we first need to have an algorithm for efficient *string comparison*, where automata provide the following baseline. Let $\hat{X}$ and $\hat{Y}$ be two GD (or two ED) strings of total sizes N and M, respectively. We first build the non-deterministic finite automaton (NFA) A of $\hat{X}$ and the NFA B of $\hat{Y}$ in time $\mathcal{O}(N+M)$. We then construct the product NFA C such that $L(C) = L(A) \cap L(B)$ in time $\mathcal{O}(NM)$. The non-emptiness decision problem, namely, checking if $L(C) \neq \emptyset$, is decidable in time linear in the size of C, using breadth-first search (BFS). Hence the comparison of $\hat{X}$ and $\hat{Y}$ can be done in time $\mathcal{O}(NM)$. It is known that if faster methods existed for obtaining the automata intersection, then significant improvements would be implied for many long-standing open problems [69]. Hence, an immediate reduction to the problem of NFA intersection does not particularly help. At the beginning of Section 4.3.1,

we show that, for GD strings, we can build an ad-hoc deterministic finite automaton (DFA) for $\hat{X}$ and $\hat{Y}$, so that the intersection can be performed efficiently, but this simple solution cannot achieve $\mathcal{O}(N+M)$ time as its cost is alphabet-dependent.

4.1.2 Contributions

Our first result in this paper is an $\mathcal{O}(N+M)$-time algorithm for deciding whether the intersection of two GD strings of sizes N and M, respectively, derived from an integer alphabet is non-empty. This result is based on a combinatorial result of independent interest: although the intersection of two GD strings can be exponential according to the total size of the two strings, it can only be represented in linear space. An automata model of computation can also be employed to obtain these results but we present here an efficient implementation in the standard word RAM model with word size $w = \Omega(\log(N+M))$ that also works for integer alphabets. We then apply our string comparison tool to compute palindromes in GD strings. We present an $\mathcal{O}(\min\{W, n^2\}N)$-time algorithm for computing all palindromes in $\hat{S}$. Furthermore, we show a non-trivial $\Omega(n^2|\Sigma|)$ lower bound under the Strong Exponential Time Hypothesis [59, 60] for computing all maximal palindromes, see Appendix B. Note that there is an infinite family of GD strings derived from an integer alphabet of size $|\Sigma| = \Theta(N)$ in respect to which our algorithm requires time $\mathcal{O}(n^2N)$, thus matching the conditional lower bound. Finally, proof-of-concept experimental results are presented using real protein datasets; specifically, we apply our tools to find the location of palindromes in immunoglobulins genes of the human V regions.

4.2 Preliminaries and Definitions

A string P is said to be a *palindrome* if, and only if, $P = P^R$. If factor $x[i \dots j]$, $0 \leq i \leq j \leq n-1$, of string x of length n is a palindrome, then $\frac{i+j}{2}$ is the *centre* of $x[i \dots j]$ in x and $\frac{j-i+1}{2}$ is the *radius* of $x[i \dots j]$. In other words, a palindrome is a string that reads the same both forwards and backwards, i.e. a string P is a palindrome if $P = YaY^R$ where Y is a string, Y^R is the reversal of Y and a is either a single letter or the empty string; see Figure 4.3 shows an ordinary palindrome $P = YaY^R$, for a sequence that can be read the same in either direction , where $a = x[6] =$C with radius 6 and centre at $x[6]$ and Figure 4.4 as an example of a complement palindrome.

Fig. 4.3 A sequence that represents an ordinary palindrome in DNA

Fig. 4.4 A sequence that represents a complement palindrome in DNA

Moreover, $x[i \dots j]$ is called a *palindromic factor* of x. It is said to be a *maximal palindrome* if there is no other palindrome in x with centre $\frac{i+j}{2}$ and a larger radius. Hence, x has exactly $2n-1$ maximal palindromes. A maximal palindrome P of x can be encoded as a pair (c,r), where c is the centre of P in x and r is the radius of P. Figure 4.5 shows a maximal palindrome with radius 4 and centre at $x[6]$, while $x[3 \dots 9]$ is a palindromic factor.

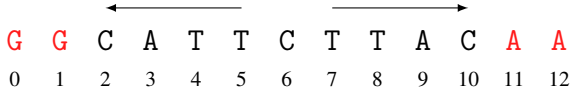


Fig. 4.5 A sequence that represents a maximal palindrome in DNA

Definition 1. A *generalised degenerate string (GD string)* $\hat{S} = \hat{S}[0]\hat{S}[1]\ldots\hat{S}[n-1]$ of *length* n derived from an alphabet Σ is a finite sequence of n degenerate letters. Every *degenerate letter* $\hat{S}[i]$ of *width* $k_i > 0$, denoted also by $w(\hat{S}[i])$, is a finite non-empty set of strings $\hat{S}[i][j] \in \Sigma^{k_i}$, with $0 \leq j < |\hat{S}[i]|$. For any GD string $\hat{S}$, we denote by $\hat{S}[i]\ldots\hat{S}[j]$ the *GD substring* of $\hat{S}$ that starts at position i and ends at position j. Figure 4.6 shows an example of GD string $\hat{S}$.

$$\hat{S} = \{\mathtt{A}\} \cdot \left\{\begin{matrix}\mathtt{GC}\\ \mathtt{AG}\end{matrix}\right\} \cdot \left\{\begin{matrix}\mathtt{TCT}\\ \mathtt{CGA}\\ \mathtt{TCA}\end{matrix}\right\} \cdot \{\mathtt{A}\} \cdot \left\{\begin{matrix}\mathtt{TCTC}\\ \mathtt{GCTC}\\ \mathtt{CGCA}\end{matrix}\right\} \cdot \{\mathtt{G}\}$$

Fig. 4.6 A GD string $\hat{S}$ starts at $\hat{S}[0]$ and ends at $\hat{S}[5]$.

Definition 2. The *total size* N and *total width* W, denoted also by $w(\hat{S})$, of a GD string $\hat{S}$ are defined, respectively, as $N = \sum_{i=0}^{n-1} |\hat{S}[i]| \times k_i$ and $W = \sum_{i=0}^{n-1} k_i$.

In this work, we generally consider GD strings derived from an *integer alphabet* of size $\sigma = N^{\mathcal{O}(1)}$.

Example 15. The GD string $\hat{S}$ of Figure 4.6 has length $n = 6$, size $N = 28$ and $W = 12$. When a GD string $\hat{X}$ has length $n = 1$, then $\hat{X}$ is simply a set of strings of the same length, which we also refer to as a *degenerate letter*.

Definition 3. Given two degenerate letters $\hat{X}$ and $\hat{Y}$, their *Cartesian concatenation* is

$$\hat{X} \otimes \hat{Y} = \{xy \mid x \in \hat{X}, y \in \hat{Y}\}.$$

When $\hat{Y} = \emptyset$ (resp. $\hat{X} = \emptyset$) we set $\hat{X} \otimes \hat{Y} = \hat{X}$ (resp. $= \hat{Y}$). Notice that $\otimes$ is associative.

Example 16. Given two GD letters, $\hat{X}$ and $\hat{Y}$, respectively, their $\hat{X} \otimes \hat{Y}$ is shown below:

$$\hat{X} \otimes \hat{Y} = \left\{ \begin{array}{l} \texttt{GC} \\ \\ \texttt{AG} \end{array} \right\} \otimes \left\{ \begin{array}{l} \texttt{TCT} \\ \\ \texttt{CGA} \\ \\ \texttt{TCA} \end{array} \right\} = \left\{ \begin{array}{l} \texttt{GCTCT} \\ \\ \texttt{GCCGA} \\ \\ \texttt{GCTCA} \end{array} \right\} \left\{ \begin{array}{l} \texttt{AGTCT} \\ \\ \texttt{AGCGA} \\ \\ \texttt{AGTCA} \end{array} \right\}$$

Definition 4. Consider a GD string $\hat{S}$ of length n. The *language of* $\hat{S}$ is:

$$L(\hat{S}) = \hat{S}[0] \otimes \hat{S}[1] \otimes \cdots \otimes \hat{S}[n-1].$$

Given two GD strings $\hat{R}$ and $\hat{S}$ of equal total width the *intersection* of their languages is defined by $L(\hat{R}) \cap L(\hat{S})$.

Definition 5. Let $\hat{X} = \{ x_i \in \Sigma^k \}$ and $\hat{Y} = \{ y_j \in \Sigma^h \}$ be two degenerate letters from alphabet Σ. Further let us assume, without loss of generality, that $\hat{Y}$ is the set that contains the shorter strings (i.e. $h \leq k$). We define the *chop* of $\hat{X}$ and $\hat{Y}$ and the *active suffixes* of $\hat{X}$ and $\hat{Y}$ as follows:

- $\text{chop}_{\hat{X},\hat{Y}} = \{ y_j \in \hat{Y} \mid y_j \text{ matches a prefix of } x_i \in \hat{X} \}$
- $\text{active}_{\hat{X},\hat{Y}} = \{ x_i[h \ldots k-1] \mid x_i[0 \ldots h-1] \in \text{chop}_{\hat{X},\hat{Y}} \}$

Let $w(\text{chop}_{\hat{X},\hat{Y}}) = \min\{w(\hat{X}), w(\hat{Y})\}$. When $\text{active}_{\hat{X},\hat{Y}} = \{\varepsilon\}$, we set $\text{active}_{\hat{X},\hat{Y}} = \emptyset$. We then have that $\text{active}_{\hat{X},\hat{Y}} = \emptyset$ either if $h = k$, or if there is no match between any of the strings in $\hat{Y}$ and the prefix of a string in $\hat{X}$; i.e. $\text{chop}_{\hat{X},\hat{Y}} = \emptyset$.

Example 17. Consider the following degenerate letters $\hat{X}$ and $\hat{Y}$ where $w(\hat{Y}) < w(\hat{X})$. The underlined strings in letter $\hat{Y}$ are prefixes of strings in letter $\hat{X}$, hence they are in $\text{chop}_{\hat{X},\hat{Y}}$. The suffixes of such strings in $\hat{X}$ are the active suffixes in $\text{active}_{\hat{X},\hat{Y}}$.

$$\hat{X} = \left\{ \begin{array}{l} \underline{\texttt{TCC}}\ \texttt{TA} \\ \texttt{ATCGA} \\ \underline{\texttt{TCC}}\texttt{AC} \\ \underline{\texttt{CAT}}\texttt{TA} \end{array} \right\} \quad \hat{Y} = \left\{ \begin{array}{l} \texttt{GCA} \\ \underline{\texttt{CAT}} \\ \underline{\texttt{TCC}} \end{array} \right\} \quad \text{chop}_{\hat{X},\hat{Y}} = \left\{ \begin{array}{l} \texttt{CAT} \\ \texttt{TCC} \end{array} \right\} \quad \text{active}_{\hat{X},\hat{Y}} = \left\{ \begin{array}{l} \texttt{TA} \\ \texttt{AC} \end{array} \right\}$$

Definition 6. Let $\hat{R}$ and $\hat{S}$ be two GD strings of length r and s, respectively. $\hat{R}[0]\ldots\hat{R}[i]$ is the *prefix* of $\hat{R}$ that ends at position i. It is called *proper* if $i \neq r-1$. We say that $\hat{R}[0]\ldots\hat{R}[i]$ is *synchronised* with $\hat{S}[0]\ldots\hat{S}[j]$ if $w(\hat{R}[0]\ldots\hat{R}[i]) = w(\hat{S}[0]\ldots\hat{S}[j])$. We call these the *shortest synchronised prefixes* of $\hat{R}$ and $\hat{S}$, respectively, when $\forall\ i' < i, j' < j$ $w(\hat{R}[0]\ldots\hat{R}[i']) \neq w(\hat{S}[0]\ldots\hat{S}[j'])$. If no prefixes of $\hat{R}$ and $\hat{S}$ can be synchronised, then we say that $\hat{R}$ and $\hat{S}$ are *unsynchronised*.

4.3 Algorithm

4.3.1 GD String Comparison

In this section, we consider the fundamental problem of GD string comparison. Let $\hat{R}$ and $\hat{S}$ be of total size N and M, respectively. We provide an $\mathcal{O}(N+M)$-time algorithm in the

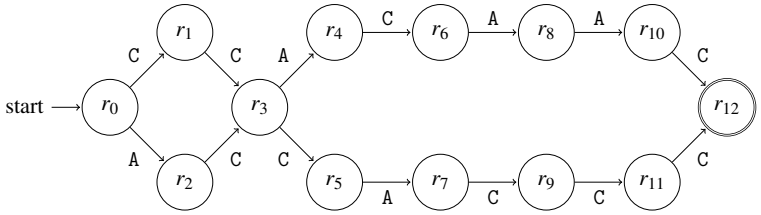
standard word RAM model with word size $w = \Omega(\log(N+M))$ that works also for integer alphabets.

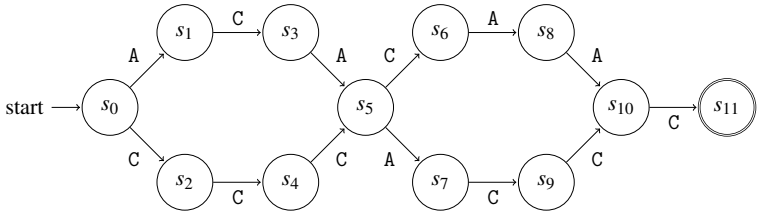
Before presenting our efficient implementation, we observe that there is the following simple algorithm based on DFAs. Each degenerate letter of $\hat{R}$ and $\hat{S}$ can be represented by a tree, whose leaves are collapsed to a single one. For every two consecutive degenerate letters, the collapsed leaves of the former tree coincide with the root of the latter tree. An acyclic DFA is obtained in this way, as illustrated in Example 18. We can perform the comparison of $\hat{R}$ and $\hat{S}$ by intersecting their corresponding DFAs using BFS on their product DFA. The trivial upper bound on the number of reachable states is $\mathcal{O}(NM)$, but this can be improved to $\mathcal{O}(N+M)$ by exploiting the structure of the two input DFAs. Each state in such a DFA has a unique level: the common length of paths from the initial state; and this structure is *inherited* by the product DFA. In other words, a level-i state in the product DFA corresponds to a pair of level-i states in the input DFAs. Observe that a level-i state in one DFA is uniquely represented by the label of the path from the root of its tree, and for a fixed DFA and level, these labels have uniform lengths. Considering the two states composing a reachable state in the product DFA, it is easy to see that the shorter label must be a suffix of the longer label. Hence, the state in the DFA with longer labels at level i uniquely determines the state in the DFA with shorter labels at level i. Consequently, the number of reachable level-i states in the product DFA is bounded by the number of level-i states in the input DFAs, and the size is $\mathcal{O}(N+M)$.

Example 18. We illustrate here a simple automata-based approach. Say we want to compare the following two GD strings:

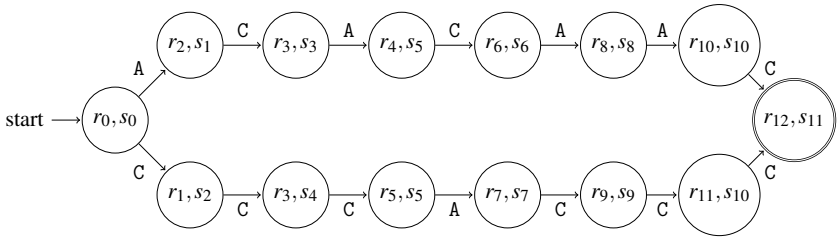
$$\hat{R} = \left\{ \begin{matrix} \texttt{AC} \\ \texttt{CC} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{ACAAC} \\ \texttt{CACCC} \end{matrix} \right\} \qquad \hat{S} = \left\{ \begin{matrix} \texttt{ACA} \\ \texttt{CCC} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{ACC} \\ \texttt{CAA} \end{matrix} \right\} \cdot \left\{ \texttt{C} \right\}.$$

We construct the DFA for $\hat{R}$ and the DFA for $\hat{S}$.





Their product DFA gives their intersection: ACACAAC and CCCACCC.



We observe that computing the product DFA is alphabet-dependent, due to branching (transition function) on the same letter in the states of the two input DFAs.

We observe that the cost of implementing the above ideas has an extra logarithmic factor due to state branching and, moreover, GD string comparisons require building the DFAs each

time. We show, however, that it is possible to obtain $\mathcal{O}(N+M)$ time for integer alphabets, *without* creating DFAs. Specifically, even if the size of $L(\hat{R}) \cap L(\hat{S})$ can be exponential in the total sizes of $\hat{R}$ and $\hat{S}$ (Fact 3), the problem of GD string comparison, i.e. deciding whether $L(\hat{R}) \cap L(\hat{S})$ is non-empty, can be solved in time linear with respect to the sum of the total sizes of the two GD strings (Theorem 3) and is thus of independent interest.

Fact 3. Given two GD strings, $\hat{R}$ and $\hat{S}$, $L(\hat{S}) \cap L(\hat{R})$ can have a size exponential in the total sizes of $\hat{R}$ and $\hat{S}$.

We next show when it is possible to factorise $L(\hat{R}) \cap L(\hat{S})$ into a Cartesian concatenation.

Lemma 2. Consider two GD strings, $\hat{S} = \hat{S}'\hat{S}''$ and $\hat{R} = \hat{R}'\hat{R}''$, such that $w(\hat{S}) = w(\hat{R})$. If $\hat{S}'$ is synchronised with $\hat{R}'$, then $L(\hat{R}) \cap L(\hat{S}) = (L(\hat{R}') \cap L(\hat{S}')) \otimes (L(\hat{R}'') \cap L(\hat{S}''))$.

See Proof B in Appendix B.

By applying Lemma 2 wherever $\hat{R}$ and $\hat{S}$ have synchronised prefixes, we are then left with the problem of intersecting GD strings with no synchronised proper prefixes. We now define an alternative decomposition within such strings (see also Example 19).

Definition 7. Let $\hat{R}$ and $\hat{S}$ be two GD strings of length r and s, respectively, with no synchronised proper prefixes. We define

$$\text{c-chain}(\hat{R}, \hat{S}) = \max_q \{0 \leq q \leq r+s-2 \mid \text{chop}_q \neq \emptyset\},$$

where chop_i denotes the set $\text{chop}_{\hat{A}_i, \hat{B}_i}$, and $(\hat{A}_0, \hat{B}_0), (\hat{A}_1, \hat{B}_1), \ldots, (\hat{A}_q, \hat{B}_q)$, $\text{pos}(\hat{A}_i), \text{pos}(\hat{B}_i)$ are recursively defined as follows:

$\hat{A}_0 = \hat{R}[0], \hat{B}_0 = \hat{S}[0],$ and $\text{pos}(\hat{A}_0) = \text{pos}(\hat{B}_0) = 0$. For $0 < i \leq r+s-2$, if $\text{chop}_{i-1} \neq \emptyset$,

$$\hat{A}_i = \begin{cases} \hat{R}[\text{pos}(\hat{A}_{i-1})+1] \text{ and } \text{pos}(\hat{A}_i) = \text{pos}(\hat{A}_{i-1})+1 & \text{if } w(\text{chop}_{i-1}) = w(\hat{A}_{i-1}) \\ \text{active}_{\hat{A}_{i-1},\hat{B}_{i-1}} \text{ and } \text{pos}(\hat{A}_i) = \text{pos}(\hat{A}_{i-1}) & \text{otherwise} \end{cases}$$

$$\hat{B}_i = \begin{cases} \hat{S}[\text{pos}(\hat{B}_{i-1})+1] \text{ and } \text{pos}(\hat{B}_i) = \text{pos}(\hat{B}_{i-1})+1 & \text{if } w(\text{chop}_{i-1}) = w(\hat{B}_{i-1}) \\ \text{active}_{\hat{A}_{i-1},\hat{B}_{i-1}} \text{ and } \text{pos}(\hat{B}_i) = \text{pos}(\hat{B}_{i-1}) & \text{otherwise} \end{cases}$$

The generation of pairs $(\hat{A}_i, \hat{B}_i)$ stops at $i = q$ either if $q = r+s-2$, or when $\text{chop}_{q+1} = \emptyset$, in which case $\hat{R}$ and $\hat{S}$ only match until $(\hat{A}_q, \hat{B}_q)$. Intuitively, $\hat{A}_i$ (respectively, $\hat{B}_i$) represents suffixes of the current position of $\hat{R}$ (respectively, of $\hat{S}$), while $\text{pos}(\hat{B}_i)$ (respectively, $\text{pos}(\hat{A}_i)$) tells *which* position of $\hat{R}$ (respectively, $\hat{S}$) we are chopping.

Example 19 (Definition 7)**.** Consider the following GD strings, $\hat{R}$ and $\hat{S}$, with no synchronised proper prefixes: chop_0 is the first red set from the left, chop_1 is the first blue one, chop_2 is the second red one, etc. The c-chain$(\hat{R}, \hat{S})$ terminates when $q = 7$.

$\hat{R} = \{\text{CGCAC}, \text{AGCCG}, \underbrace{\underbrace{\underbrace{\text{AAG}}_{\hat{A}_0}\underbrace{\text{TC}}_{\hat{A}_1}}_{\hat{A}_2}}_{\hat{A}_3}\} \cdot \{\text{AAT}, \underbrace{\underbrace{\text{TAG}}_{\hat{A}_4}}_{\hat{A}_5}\} \cdot \{\text{CTCG}, \text{GCAG}, \underbrace{\underbrace{\text{CTCA}}_{\hat{A}_6}}_{\hat{A}_7}\}$

$\hat{S} = \{\underbrace{\text{A}}_{\hat{B}_0}\} \cdot \{\text{GC}, \underbrace{\text{AG}}_{\hat{B}_1}\} \cdot \{\text{TCT}, \text{CGA}, \underbrace{\text{TCA}}_{\hat{B}_2}\} \cdot \{\underbrace{\text{A}}_{\hat{B}_4}\} \cdot \{\text{TCTC}, \text{GCTC}, \underbrace{\underbrace{\text{CGCA}}_{\hat{B}_5}}_{\hat{B}_6}\} \cdot \{\underbrace{\text{G}}_{\hat{B}_7}\}$

Definition 8. Let $\hat{R}$ and $\hat{S}$ be two GD strings of length r and s, respectively, with $w(\hat{R}) = w(\hat{S})$ and no synchronised proper prefixes. We define $G_{\hat{R},\hat{S}}$ as a directed acyclic graph with a

structure of up to $r+s-1$ levels, each node being a set of strings, as follows, where we assume, without loss of generality, that $w(\hat{R}[0]) > w(\hat{S}[0])$:

Level $k=0$**:** consists of a single node:

$n_0 = \{x \in \hat{R}[0] \,|\, x = y_0 \dots y_{q_0} \text{with } y_j \in \text{chop}_j \,\forall j : 0 \leq j \leq q_0\}$, where q_0 is the index of the rightmost chop containing suffixes of $\hat{R}[0]$.

Level $k>0$**:** consists of $\ell = |\text{chop}_{q_{k-1}}|$ nodes. Assuming, without loss of generality, that level $k-1$ has been built with suffixes of $\hat{R}[\text{pos}(\hat{A}_{q_{k-1}})]$, level k contains suffixes of a position of $\hat{S}$. Let $c_0, \dots, c_{\ell-1}$ denote the elements of $\text{chop}_{q_{k-1}}$. Then, for $0 \leq i \leq \ell-1$, the i-th node of level k is:

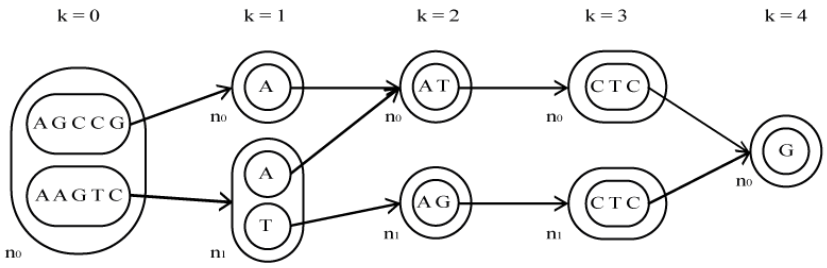
$n_i = \{y_{q_{k-1}+1} \dots y_{q_k} \,|\, c_i y_{q_{k-1}+1} \dots y_{q_k} \in \hat{B}_{q_{k-1}} \text{with } y_j \in \text{chop}_j \,\forall j : q_{k-1}+1 \leq j \leq q_k\}$, where q_k is the index of the rightmost chop containing suffixes of $\hat{S}[\text{pos}(\hat{B}_{q_{k-1}})]$.

Every string in level $k-1$ whose suffix is c_i is the source of an edge having the whole node n_i as a sink.

We define $\text{paths}(G_{\hat{R},\hat{S}})$ as the set of strings spelled by a path in $G_{\hat{R},\hat{S}}$ that starts at n_0 and ends at the last level.

Note that the size of $G_{\hat{R},\hat{S}}$ is at most linear in the sum of the sizes of $\hat{R}$ and $\hat{S}$, since the nodes contain strings either in $\hat{R}$ or in $\hat{S}$ with no duplications, and each node has an out-degree equal to the number of strings it contains.

Example 20 (Definition 8)**.** $G_{\hat{R},\hat{S}}$ for the GD strings $\hat{R}, \hat{S}$ of Example 19 is:



$q_0 = 2$ and the strings in level 0 belong to $(\text{chop}_0 \otimes \text{chop}_1 \otimes \text{chop}_2) \cap \hat{R}[0]$. Level 1 contains suffixes of strings in $\hat{B}_2$ (and of strings in $\hat{B}_3$ as $\text{chop}_3 = \{\texttt{A}, \texttt{T}\}$ and indeed $q_1 = 3$). Level 2 contains suffixes of strings in $\hat{A}_3$ (as $q_2 = 5$). Level 3 contains suffixes of strings in $\hat{B}_5$ ($q_3 = 6$). Level 4 contains suffixes of strings in $\hat{A}_6$ ($q_4 = 7$). The three paths from level 0 to level 4 correspond to the three strings in $L(\hat{R}) \cap L(\hat{S})$: `AGCCGAATCTCG`, `AAGTCAATCTCG`, `AAGTCTAGCTCG`.

Let $G^k_{\hat{R},\hat{S}}$ be $G_{\hat{R},\hat{S}}$ truncated at level k, and let $|G^k_{\hat{R},\hat{S}}|$ be the length of the strings it spells. Let $L_k(\hat{S})$ denote the set of prefixes of length $|G^k_{\hat{R},\hat{S}}|$ of $L(\hat{S})$.

Lemma 3. Let $\hat{R}, \hat{S}$ be two GD strings with $w(\hat{R}) = w(\hat{S}) = W$ and no synchronised proper prefixes. Then $L_k(\hat{S}) \cap L_k(\hat{R}) = \text{paths}(G^k_{\hat{R},\hat{S}})$ for all levels k of $G_{\hat{R},\hat{S}}$ such that $L_k(\hat{S}) \cap L_k(\hat{R}) \neq \emptyset$. See Proof B in Appendix B.

As a special case of Lemma 3, if $L(\hat{S}) \cap L(\hat{R}) \neq \emptyset$, then $G_{\hat{R},\hat{S}}$ is built up to the last level and the following holds:

Theorem 2. Let $\hat{R}, \hat{S}$ be two GD strings having lengths, respectively, r and s, with $w(\hat{R}) = w(\hat{S})$ and no synchronised proper prefixes. Then $G_{\hat{R},\hat{S}}$ has exactly $r + s - 1$ levels, and we have that $L(\hat{S}) \cap L(\hat{R}) = \text{paths}(G_{\hat{R},\hat{S}})$.

$G_{\hat{R},\hat{S}}$ is thus a linear-sized representation of the possibly exponentially-sized (Fact 3) set $L(\hat{S}) \cap L(\hat{R})$.

We now show an $\mathcal{O}(N+M)$-time algorithm for the standard word RAM model, denoted by GDSC, that decides whether $L(\hat{R})$ and $L(\hat{S})$ share at least one string (returns 1) or not (returns 0). GDSC starts by constructing the generalised suffix tree $T_{\hat{R},\hat{S}}$ of all the strings in $\hat{R}$ and $\hat{S}$. Then it scans $\hat{R}$ and $\hat{S}$ starting with $\hat{R}[0]$ and $\hat{S}[0]$ storing in $\text{chop}_{\hat{R},\hat{S}}$ the latest chop_i and in $\text{active}_{\hat{R},\hat{S}}$ the latest $\text{active}_{\hat{A}_i,\hat{B}_i}$ using $T_{\hat{R},\hat{S}}$. For an efficient implementation, suffixes in $\text{active}_{\hat{R},\hat{S}}$ are stored (e.g. for $\text{active}_{\hat{A}_0,\hat{B}_0}$ assuming that $w(\hat{R}[0]) > w(\hat{S}[0])$) as index positions of $\hat{R}[0]$ and the starting position of the suffix as $\text{active}_{\hat{R},\hat{S}}$.suff. The next comparison is made between the corresponding suffixes of $\hat{R}[0]$ of length $w(\hat{R[0]}) - \text{active}_{\hat{R},\hat{S}}$.suff and $\hat{S}[1]$, identifying first the minimum length of the two, and proceeding with the same process. The comparison of letters can be: (i) between $\hat{R}[i]$ and $\hat{S}[j]$; or (ii) between the corresponding strings of $\text{active}_{\hat{R},\hat{S}}$.index and $\hat{R}[i]$; or (iii) between the corresponding strings of $\text{active}_{\hat{R},\hat{S}}$.index and $\hat{S}[j]$. If the two GD strings have a synchronised proper prefix, this will result in $\text{active}_{\hat{R},\hat{S}} = \emptyset$ at positions i in $\hat{R}$ and j in $\hat{S}$. At this point, the comparison is restarted with the immediately following pair of degenerate letters.

Theorem 3. Algorithm GDSC is correct. Given two GD strings, $\hat{R}$ and $\hat{S}$, of total sizes N and M, respectively, derived from an integer alphabet, algorithm GDSC requires $\mathcal{O}(N+M)$ time. See Proof in Appendix B.

$$\hat{S} = \{\underline{\texttt{A}}\} \cdot \left\{\begin{matrix}\underline{\texttt{GC}}\\ \texttt{AG}\end{matrix}\right\} \cdot \left\{\begin{matrix}\texttt{TCT}\\ \underline{\texttt{CGA}}\\ \texttt{TCA}\end{matrix}\right\} \cdot \{\texttt{A}\} \cdot \left\{\begin{matrix}\texttt{TCTC}\\ \underline{\texttt{GCTC}}\\ \texttt{CGCA}\end{matrix}\right\} \cdot \{\underline{\texttt{G}}\}$$

Fig. 4.7 A palindrome at $\hat{S}[0]\dots\hat{S}[2]$ and $\hat{S}[4]\dots\hat{S}[5]$

4.3.2 Computing Palindromes in GD Strings

Armed with the efficient GD string comparison tool, we shift our focus towards our initial motivation, namely, computing palindromes in GD strings.

Definition 9. A GD string $\hat{S}$ is a *GD palindrome* if there exists a string in $L(\hat{S})$ that is a palindrome.

A GD palindrome $\hat{S}[i]\dots\hat{S}[j]$ in $\hat{S}$, whose total width is $w(\hat{S}[i]\dots\hat{S}[j])$, can be encoded as a pair (c,r), where its *centre* is $c = \frac{w(\hat{S}[0]\dots\hat{S}[i-1])+w(\hat{S}[0]\dots\hat{S}[j])-1}{2}$, when $i > 0$, otherwise, $c = \frac{w(\hat{S}[0]\dots\hat{S}[j])-1}{2}$, when $i = 0$; its *radius* is $r = \frac{w(\hat{S}[i]\dots\hat{S}[j])}{2}$. $\hat{S}[i]\dots\hat{S}[j]$ is called *maximal* if no other GD palindrome (c,r') exists in $\hat{S}$ with $r' > r$. Note that we only consider the GD palindromes $\hat{S}[i]\dots\hat{S}[j]$ that start with the first letter of some string $X \in \hat{S}[i]$ and end with the last letter of some string $Y \in \hat{S}[j]$, while the centre can be anywhere: in between or inside degenerate letters. That is, in $\hat{S}$, there are $2 \cdot w(\hat{S}) - 1 = 2W - 1$ possible centres.

Example 21. Consider the GD string $\hat{S}$ in Figure 4.7 where palindromes are underlined; one starts at $\hat{S}[0]$ and ends at $\hat{S}[2]$: it corresponds to $(c,r) = (2.5,3)$. A second palindrome starts at $\hat{S}[4]$ and ends at $\hat{S}[5]$: it corresponds to $(c,r) = (9,2.5)$.

In this section, we consider the following problem. Given a GD string $\hat{S}$ of length n, total size N, and total width W, find all GD strings $\hat{S}[i]\dots\hat{S}[j]$, with $0 \le i \le j \le n-1$, that are

GD palindromes. We give two alternative algorithms: one finds all GD palindromes seeking them for all (i, j) pairs; and the other one finds them starting from all possible centres. The two algorithms have different time complexities: which one is faster depends on W, N and n. In fact, they compute all GD palindromes, but report only the maximal ones.

We first describe algorithm MAXPALPAIRS. For all i, j positions within $\hat{S}$, in order to check whether $\hat{S}[i]\ldots\hat{S}[j]$ is a GD palindrome, we apply the GDSC algorithm to $\hat{S}[i]\ldots\hat{S}[j]$ and its reverse, denoted by $rev(\hat{S}[i]\ldots\hat{S}[j])$. The reverse is defined by reversing the sequence of degenerate letters and also reversing the strings in every degenerate letter. GD palindromes are, finally, sorted per centre, and the maximal GD palindromes are reported. Sorting the (i, j) pairs by their centres can be done in $\mathscr{O}(W)$ time using bucket sort, which is bounded by $\mathscr{O}(N)$ since $N \geq W$.

Since there are $\mathscr{O}(n^2)$ pairs (i, j), and since, according to Theorem 3, the GDSC algorithm takes time proportional to the total size of $\hat{S}[i]\ldots\hat{S}[j]$ to check whether $\hat{S}[i]\ldots\hat{S}[j]$ is a GD palindrome, algorithm MAXPALPAIRS takes $\mathscr{O}(n^2N)$ time in total.

Example 22. Figure 4.8 shows the steps of the MAXPALPAIRS of given GD string $\hat{S}$ with $n = 3$. The MAXPALPAIRS will start with $i = 0$ and $j = 1$, then with $i = 1$ and $j = 2$. Finally, with $i = 0$ and $j = 2$. Step 1, MAXPALPAIRS applies the GDSC on $\hat{S}[0]\ldots\hat{S}[1]$ and its reverses. Step 2, MAXPALPAIRS applies the GDSC on $\hat{S}[1]\ldots\hat{S}[2]$ and its reverses. For Step 1 and Step 2, the GDSC reports negatively. Step 3, MAXPALPAIRS applies the GDSC on $\hat{S}[0]\ldots\hat{S}[2]$ and its reverses and GDSC will report positively. Ultimately, the algorithm will report the underline GD string `AAGTCAACTGAA`, in $\hat{S}[0][2]$ as the only maximal GD palindrome pair of $\hat{S}$ while, $\hat{S}[0][1]$ and $\hat{S}[1][2]$ do not hold any palindromes.

$$\hat{S} = \begin{Bmatrix} \texttt{CGCAC} \\ \texttt{AGCCG} \\ \texttt{AAGTC} \end{Bmatrix} \cdot \{\texttt{AA}\} \cdot \{\texttt{CTGAA}\}$$

(a) GD string $\hat{S}$ to be processed by MAXPALPAIRS

$$\hat{S}[0]\hat{S}[1] = \begin{Bmatrix} \texttt{CGCAC} \\ \texttt{AGCCG} \\ \underline{\texttt{AAG}}\texttt{TC} \end{Bmatrix} \cdot \{\texttt{AA}\}, \quad rev(\hat{S}[0]\hat{S}[1]) = \{\underline{\texttt{AA}}\} \cdot \begin{Bmatrix} \texttt{CACGC} \\ \underline{\texttt{G}}\texttt{CCGA} \\ \texttt{CTGAA} \end{Bmatrix}$$

(b) Step 1: Applying the GDSC on $\hat{S}[0]\hat{S}[1]$ pair and $rev(\hat{S}[0]\hat{S}[1])$

$$\hat{S}[1]\hat{S}[2] = \{\underline{\texttt{AA}}\} \cdot \{\texttt{CTGAA}\}, \quad rev(\hat{S}[1]\hat{S}[2]) = \{\underline{\texttt{AA}}\texttt{GTC}\} \cdot \{\texttt{AA}\}$$

(c) Step 2: Applying the GDSC on $\hat{S}[1]\hat{S}[2]$ pair and $rev(\hat{S}[1]\hat{S}[2])$

$$\hat{S}[0]\ldots\hat{S}[3] = \begin{Bmatrix} \texttt{CGCAC} \\ \texttt{AGCCG} \\ \underline{\texttt{AAGTC}} \end{Bmatrix} \cdot \{\underline{\texttt{AA}}\} \cdot \{\underline{\texttt{CTGAA}}\}, \quad rev(\hat{S}[0]\ldots\hat{S}[[2]) = \{\underline{\texttt{AAGTC}}\} \cdot \{\underline{\texttt{AA}}\} \begin{Bmatrix} \texttt{CACGC} \\ \texttt{GCCGA} \\ \underline{\texttt{CTGAA}} \end{Bmatrix}$$

(d) Step 3: Applying the GDSC on $\hat{S}[0]\ldots\hat{S}[3]$ pair and $rev(\hat{S}[0]\ldots\hat{S}[2])$

Fig. 4.8 Steps involved in processing the MaxPalPairs of GD string $\hat{S}$ for each pair

In algorithm MAXPALCENTRES, we consider all possible centres c of $\hat{S}$. In the case when c is in between two degenerate letters we simply try to extend to the left and to the right by applying GDSC. In the case when c is inside a degenerate letter we intuitively split the letter vertically into two letters and try to extend to the left and to the right by applying GDSC. At each extension step of this procedure we maintain two GD strings $\hat{L}$ (left of the centre) and $\hat{R}$ (right of the centre) such that they are of the same total width. We consider the reverse of $\hat{L}$ (similar to algorithm MAXPALPAIRS) for the comparison. In the case where c occurs inside a degenerate letter, in order to make sure we do not identify

palindromes which do not exist, for all j split strings of the degenerate letter, we check that $\hat{L}^R[0][j][0\ldots k-1] = \hat{R}[0][j][0\ldots k-1]$ where $\hat{L}^R = rev(\hat{L})$ and $k = \min(w(L^R[0]), w(\hat{R}[0]))$. If no matches are found, we move onto the next centre. Otherwise, when a match is found, we update $rev(\hat{L})$ and $\hat{R}$ with the remainder of the split degenerate letter (if its length is greater than k), as well as the next degenerate letters. Algorithm GDSC is applied to compare $rev(\hat{L})$ and $\hat{R}$. After a positive comparison, we overwrite $\hat{L}$ and $\hat{R}$ by adding the degenerate letters of the current extension until $w(\hat{L}) = w(\hat{R})$ (or until the end of the string is reached). This process is repeated as long as GDSC returns a positive comparison; that is, until the maximal GD palindrome with centre c is found. The radius reported is then the total sum of all values of $w(\hat{L})$. If GDSC returns a negative comparison at centre c, we proceed with the next centre, because we clearly cannot have a GD palindrome centred at c extended further if $rev(\hat{L}) \cap \hat{R}$ is empty.

Example 23. In Figure 4.9, given the same GD string $\hat{S}$ used in Example 22, the MAXPAL-CENTRES splits $\hat{S}$ at each centre $c = 0\ldots c = 11$ to $\hat{L}$ and $\hat{R}$, and the algorithm maintains the $w(\hat{L})$ and $w(\hat{R})$ to be equal. Since, they are equal at $c = 5.5$ and then it applies the GDSC on $\hat{R}$ and $rev(\hat{L})$. The algorithm will report the palindrome `AAGTCAACTGAA` at $c = 5.5$ which is corresponding to the reported palindrome at $\hat{S[0]}\ldots\hat{S[2]}$ in MAXPALPAIRS.

$$\hat{S} = \left\{ \begin{array}{l} \texttt{CGCAC} \\ \texttt{AGCCG} \\ \underleftarrow{\texttt{AAGTC}} \end{array} \right\} \cdot \left\{ \underleftarrow{\texttt{A}} \mid \underrightarrow{\texttt{A}} \right\} \cdot \left\{ \underrightarrow{\texttt{CTGAA}} \right\}$$

(a) Splitting GD $\hat{S}$ at $c = 5.5$, where $w(\hat{L})$ and $w(\hat{R})$=6

$$rev(\hat{L}) = \left\{ \underline{\texttt{A}} \right\} \cdot \left\{ \begin{array}{l} \texttt{CACGC} \\ \texttt{GCCGA} \\ \underline{\texttt{CTGAA}} \end{array} \right\} \qquad \hat{R} = \left\{ \underline{\texttt{A}} \right\} \cdot \left\{ \underline{\texttt{CTGAA}} \right\}$$

(b) Applying GDSC on $\hat{R}$ and $rev(\hat{L})$

Fig. 4.9 Steps of MaxPalCentres algorithm on GD string $\hat{S}$

According to Theorem 3 and the fact that there are $2W-1$ possible centres, we have that algorithm MAXPALCENTRES takes $\mathcal{O}(WN)$ time in total. We obtain the following result.

Theorem 4. Given a GD string of length n, total size N, and total width W, derived from an integer alphabet, all (maximal) GD palindromes can be computed in time $\mathcal{O}(\min\{W, n^2\}N)$.

4.4 Experimental Results

We present here a proof-of-concept experiment, although, beyond this, we anticipate that the algorithmic tools developed in this paper are applicable in a wide range of biological applications.

We first obtained the amino acid sequences of five immunoglobulins within the human V regions [39] and converted these into mRNA sequences [89]. The letters $\mathtt{X}, \mathtt{S}, \mathtt{T}, \mathtt{Y}, \mathtt{Z}, \mathtt{R}$ and $\mathtt{H}$ were replaced by degenerate letters according to IUPAC [61]. Each other letter, $c \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$, was treated as a single degenerate letter $\{c\}$. An average of 47% of the total number of positions within the five sequences consisted of one of the following: $\mathtt{X}, \mathtt{S}, \mathtt{T}, \mathtt{Y}, \mathtt{Z}, \mathtt{R}$ and $\mathtt{H}$. We then used algorithm MAXPALPAIRS to find all maximal palindromes in the five sequences. Table 4.1 shows the coordinates of maximal palindromes identified within hypervariable regions I and II. Our results are in accordance with Wuilmart et al. [99], who presented a *statistical* (fundamentally different) method to identify the location of palindromes within regions of immunoglobulin genes. The ranges we report are greater than or equal to the ones of [99] due to the *maximality* criterion. A proof-of-concept C++ implementation of the presented algorithms, together with the datasets referred to below, is made available at *https://nms.kcl.ac.uk/mai.alzamel/softwares.html*.

	Hypervariable Region			
	I		II	
V	[99]	**This paper**	[99]	**This paper**
V_kII	18-27	11-36	119-130	118-131
	104-113	104-113	169-180	169-180
V_kIII	18-27	11-30	132-142	131-145
V_λII	63-74	62-81	140-152	140-152
V_λIII	51-74	50-75	132-143	131-144
V_λV	96-104	95-104	134-141	134-141

Table 4.1 Coordinates of maximal palindromes identified within regions I and II.

Example 24. Given the immunoglobulin V_kII region, that is present as follows: Thr Leu Ser Cys Arg Ala Ser Gln Ser, we convert the amino acid to mRNA based on Table 4.2 to TZX UGY SGX GCX TZX CAR. Later, we convert the obtained mRNA to GD string $\hat{S}$ using IUPAC table shown in Table 4.3 which is represented by The International Union of Pure and Applied Chemistry 1970. A complement palindrome is underlined in the three versions of the sequence, as shown below in Figure 4.10.

$$\hat{S} = \left\{ \begin{matrix} \texttt{UCC} \\ \texttt{GU} \\ \texttt{G} \\ \texttt{A} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \underline{\texttt{UGC}} \\ \texttt{U} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \underline{\texttt{CG}}\texttt{U} \\ \texttt{GC} \\ \underline{\texttt{G}} \\ \texttt{A} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{G}\underline{\texttt{CU}} \\ \texttt{C} \\ \texttt{A} \\ \underline{\texttt{G}} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{UUC} \\ \texttt{GU} \\ \underline{\texttt{G}} \\ \texttt{C} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \underline{\texttt{CAA}} \\ \texttt{G} \end{matrix} \right\}$$

Fig. 4.10 A GD string $\hat{S}$ represents the immunoglobulin V_kII region

Amino acid	Compressed	Amino acid	Compressed
Ala	GCN	Leu	YUR, CUN
Arg	SGX	Lys	AAR
Asn	AAY	Met	AUG
Asp	GAY	Phe	UUY
Cys	UGY	Pro	CCN
Gln	CAR	Ser	TZX
Glu	GAR	Thr	ACN
Gly	GGN	Trp	UGG
His	CAY	Tyr	UAY
Ile	AUH	Val	GUN
Asx	GLN or GLU	Glx	ASN or ASP

Table 4.2 Inverse table for the standard genetic code (compressed using IUPAC)

Nucleotide Base	Base	Nucleotide Base	Base
A	A	C	C
G	G	U	U
W	A or (T/U)	S	C or G
M	A or C	K	G or (T /U)
R	A or G	Y	C or (T/U)
N	A,C, G or (T/U)	H	A, C or (T/U)
Z	C or G	X	A, C, G or (T/U)

Table 4.3 The IUPAC table

Chapter 5

Efficient Identification of k-closed Strings

The work presented in this chapter is published as: H. Alamro, M. Alzamel, C. S. Iliopoulos, S. P. Pissis, S. Watts, W. Sung, "Efficient Identification of k-Closed Strings". *Int. J. Found. Comput. Sci.*. Int. J. Found. Comput. Sci. 31(5): 595-610 (2020)

5.1 Background and Contributions

5.1.1 Background

A bordered string x is such that there exists a prefix of x which is also a suffix of x. A closed string (or a closed word) is a bordered string that satisfies an additional property: the border does not occur elsewhere in the string. There are a number of earlier studies dealing with closed strings. Fici in [30] introduced the notion of closed strings in addition to characterisations of this class.

The more practical relevance of closed strings was established via their relationship with palindromic strings. The number of closed factors in a string is minimised if these factors are also palindromic. Additionally it was shown that the upper bound on the number of palindromic factors of a string coincides with the lower bound on the number of closed factors (see [13] and references therein). Thus the study of closed strings shows potential applications in connection with applications of palindromes [5]. On the algorithmic side, Badkobeh et al. in [12] presented (among others) an algorithm for the factorisation of a given string of length n into a sequence of longest closed factors in time and space $\mathcal{O}(n)$, and another algorithm for computing the longest closed factor starting at every position in the string in $\mathcal{O}(n\frac{\log n}{\log\log n})$ time and $\mathcal{O}(n)$ space.

5.1.2 Contributions

Here we extend the definition of closed strings to k-closed strings, for which a level of approximation is permitted up to a number of Hamming distance errors, set by the parameter k. The main contribution is an $\mathcal{O}(kn)$-time and $\mathcal{O}(n)$-space algorithm for deciding whether or not a given string of length n over an integer alphabet is k-closed. We also provide experimental results here as a proof of concept.

5.2 Preliminaries

The algorithm described in this paper makes substantial use of the *kangaroo method*, a well-established method used to perform multiple LCE_k queries on a given string x [45]. This

is done by initially preprocessing the string x to build a *suffix tree* data structure in $\mathcal{O}(n)$ time and space [27]. The suffix tree of x allows us to perform LCE queries in $\mathcal{O}(1)$ time. For a query $\mathrm{LCE}(i,j)$, we first identify two distinct leaf nodes corresponding to the suffixes i and j; then, the lowest common ancestor node in the tree for these two leaf nodes has a string depth equal to $\mathrm{LCE}(i,j)$. The calculation of the lowest common ancestor and its depth may be performed in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ preprocessing time [15], hence the LCE query can be computed in $\mathcal{O}(1)$ time.

The kangaroo method extends this methodology, allowing the calculation of LCE_k queries. Precisely, we have the following lemma:

Lemma 4. Given the suffix tree of x, $\mathrm{LCE}_k(i,j)$ can be computed in $\mathcal{O}(k)$ time.

Proof. $\mathrm{LCE}_k(i,j)$ can be defined recursively as follows:

We denote $l_r = \mathrm{LCE}_r(i,j)$ for any $r \geq 0$. Then, we have $l_0 = \mathrm{LCE}(i,j)$. Also, we have $l_r = l_{r-1} + 1 + \mathrm{LCE}(i + l_{r-1} + 1, j + l_{r-1} + 1)$. By the above recursive formula, $\mathrm{LCE}_k(i,j)$ can be computed by performing LCE queries k times. Since each LCE query requires $\mathcal{O}(1)$ time, the lemma follows. □

Inspect Figure 5.1 for an example.

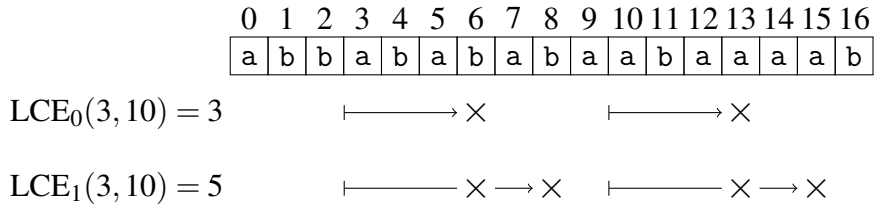


Fig. 5.1 Longest common extensions for $k = 0$ and $k = 1$.

5.3 k-closed Strings

The definition of closed strings can be generalised to *k-closed strings*, where k expresses a Hamming distance error bound. This is useful for dealing with strings where there may be errors or approximations in the data.

Definition 10. A string x of length n is called *k-closed* if, and only if, $n \leq 1$ or the following properties are satisfied for some k' where $0 \leq k' \leq k$:

1. There exists some proper prefix u of x and some proper suffix v of x of length $|u| = |v|$, such that $\delta_H(u,v) \leq k'$.

2. Except for u and v, there exists no factor w of x of length $|w| = |u| = |v|$ such that $\delta_H(u,w) \leq k'$ or $\delta_H(v,w) \leq k'$.

For the above definition, the pair u and v for the smallest k' is called the *k-closed border* of x. In the case where $n \leq 1$ we assign ε as the k-closed border.

Note that this is a generalisation of the closed string problem, which is now known as a 0-closed string problem. It is clear from the definition that a smaller value of k corresponds to k-closed being a stronger statement on the nature of x. Lemma 5 therefore follows trivially from Definition 10.

Lemma 5. A string x that is k-closed is also r-closed for all $r > k$.

It additionally follows from Definition 10 that the k-closed border of a string x is unique by Lemma 6.

Lemma 6. A length-n k-closed string x, $n > 1$, has exactly one k-closed border; i.e. it has exactly one prefix u and one suffix v satisfying the conditions in Definition 10 for the smallest $k' \leq k$. See proof C in Appendix C.

Let x be a non-empty k-closed string of length n. The following properties follow easily from Definition 10 and Lemma 6:

1. x has exactly one k-closed border.

2. If $n > 1$, there exists a string w with $|w| < n$ and a natural number k', with $0 \leq k' \leq k$, such that $w \approx_{k'} x[i..i+|w|-1]$ for exactly two values of i and no others; specifically $i = 0$ and $i = n - |w|$.

3. There exists a natural number k', with $0 \leq k' \leq k$, such that the longest repeated prefix (resp. suffix) of x within k' errors, is equal to u (resp. v), where u and v are the prefix and suffix, respectively, comprising the k-closed border.

4. There exists a natural number k', with $0 \leq k' \leq k$, such that any repeated prefix (resp. suffix) of x within k' errors is necessarily a prefix (resp. suffix) of u (resp. v), where u and v are the prefix and suffix, respectively, comprising the k-closed border.

We display an example in Figure 5.2. Note that, for the string `GTGAGTGGTA`, we illustrate only that a border length of 3 with error 1 is not a possible 1-closed border. To verify fully that it is non 1-closed, all combinations of border lengths and error levels $0 \leq k' \leq 1$ must be considered. It is in fact possible to show that no borders of any length exist that satisfy

the 1-closed criteria, therefore the string is indeed non 1-closed (and by Lemma 5 also non 0-closed).

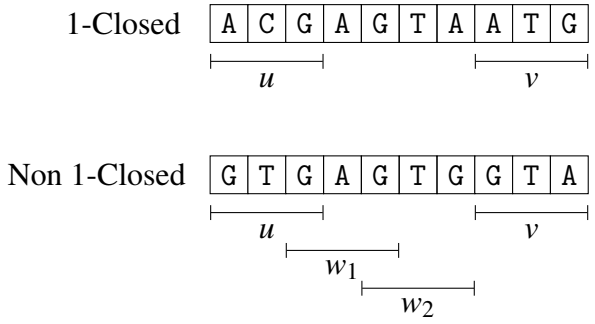


Fig. 5.2 Closed and non-closed strings for $k = 1$.

We are now in a position formally to define the problem solved in this paper. For computation purposes, we focus only on the case when $k > 0$.

k-CLOSED BORDER
Input: A string x of length n and a natural number k, $0 < k < n$
Output: The k-closed border or -1 if x is not k-closed

5.4 Algorithm

In addition to our definition of k-closed strings, we further define some variants of Definition 10 which proved useful in obtaining the main result.

Definition 11. A string x of length n is called *k-weakly-closed* if, and only if, $n \leq 1$ or the following properties are satisfied:

1. Some proper prefix u of x exists, and some proper suffix v of x of length $|u| = |v|$, such that $\delta_H(u,v) \leq k$.

2. Both factors u and v occur only as a prefix and suffix, respectively, within x, i.e. there are no internal occurrences of u or v in x.

We call such a pair u and v a *k-weakly-closed border* of x. In the case where $n \leq 1$, we assign ε as the k-weakly-closed border.

Definition 11 is satisfied in situations where the border may have errors, but internal occurrences are considered not to have errors. Figure 5.3 shows that ACTGTAATTAGT is 1-weakly-closed with a 1-weakly-closed border (ACT, AGT) of length 3, whereas ACTGTAGTTAGT is not 1-weakly-closed.

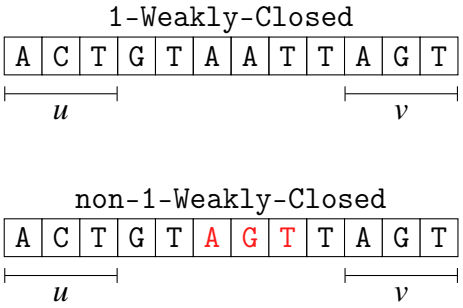


Fig. 5.3 1-weakly-closed border with length 3 and not 1-weakly-closed string

Note that under this definition there may be multiple k-weakly-closed borders.

For example, in Figure 5.4 TATAGAACATAT is 2-weakly-closed with two 2-weakly-closed borders (TAT, TAT) of length 3 and (TATAG, CATAT) of length 5.

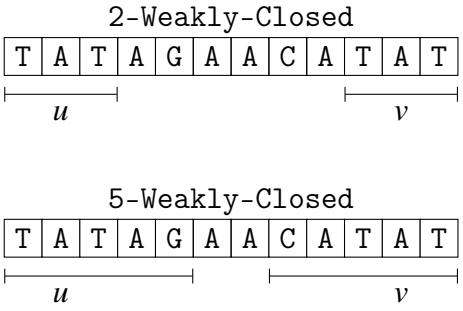


Fig. 5.4 2-weakly-closed border with length 3 and 5-weakly-closed with length 5

Definition 12. A string x of length n is called *k-strongly-closed* if, and only if, $n \leq 1$, or the following properties are satisfied:

1. There exists some non-empty border b of x.

2. There exists no factor w of x of length $|w| = |b|$ such that $\delta_H(b,w) \leq k$, except the prefix and suffix of x.

We call b the *k-strongly-closed border* of x. In the case where $n \leq 1$, we assign ε as the k-strongly-closed border.

Definition 12 is satisfied in situations where the border does not have errors, but internal occurrences may have errors. For example, in Figure 5.5, `ACTGTATCAACT` is 1-strongly-closed with a 1-strongly-closed border `ACT` of length 3, whereas `ACTGTATTAACT` is not 1-strongly-closed. Note that under this definition there is only one k-strongly-closed border.

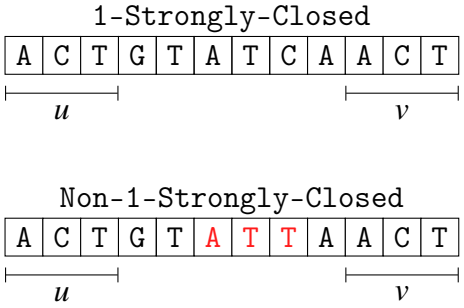


Fig. 5.5 1-strongly-closed border with length 3 and non 1-strongly-closed string

Definition 13. A string x of length n is called *k-pseudo-closed* if, and only if, $n \leq 1$, or the following properties are satisfied:

1. There exists some proper prefix u of x and some proper suffix v of x of length $|u| = |v|$, such that $\delta_H(u,v) \leq k$.

2. Except for u and v, there exists no factor w of x of length $|w| = |u| = |v|$ such that $\delta_H(u,w) \leq k$ or $\delta_H(v,w) \leq k$.

We call such a pair u and v the *k-pseudo-closed border* of x. In the case where $n \leq 1$, we assign ε as the k-pseudo-closed border.

Conditions 1 and 2 of Definition 13 may be regarded as merging of Definition 11 and Definition 12. Both the border and internal occurrences may have errors.

For example in Figure 5.6, ABTCTTACCTAGT is 1-pseudo-closed with a 1-pseudo-closed border (ABT, AGT) of length 3, whereas ABTCTTABCTAGT is not 1-pseudo-closed.

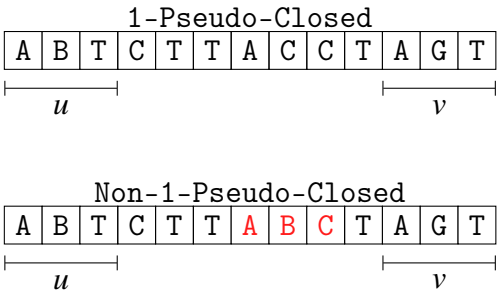


Fig. 5.6 1-pseudo-closed border with length 3 and non 1-pseudo-closed border

Note that Condition 1 is *less selective* and Condition 2 is *more selective*. The requirement to satisfy both conditions therefore implies that a 0-closed string is not necessarily k-pseudo-closed, and a k-pseudo-closed string is not necessarily 0-closed (hence the *pseudo* term). For instance, abac is 1-pseudo-closed with a border (ab, ac), but not 0-closed. In a contrary example, abba is 0-closed with a border a but not 1-pseudo-closed.

Note that there is a similarity in the construction of Definition 10 and Definition 13 that permits us easily to conclude the following crucial lemma:

Lemma 7. x is k-closed $\iff$ $\exists k'$ where $0 \leq k' \leq k$, such that x is k'-pseudo-closed.

We begin by constructing the suffix tree of x. As has been discussed, this is constructible in $\mathcal{O}(n)$ time and space. Recall that once the suffix tree is constructed it can be pre-processed within the same complexity to answer any $\text{LCE}_k(i,j)$ query by applying the Kangaroo method in $\mathcal{O}(k)$ time (see Lemma 4). For the purpose of this algorithm, we draw attention to a specific subset of the possible LCE_k queries and store their values in two related data structures. These structures are the *longest prefix k-match array* and *longest suffix k-match array* of string x, denoted by $\text{LPM}_k(x)$ and $\text{LSM}_k(x)$, respectively. $\text{LPM}_k(x)[j]$ (respectively $\text{LSM}_k(x)[j]$) is defined as the length of the longest factor of x starting (ending) at index j, which matches the prefix (suffix) of x of the same length within k errors, with the exception of the index 0 $(n-1)$ corresponding to the prefix (suffix) itself, for which we set a value of -1. Note that within the literature, the LPM array is similar to the *k-prefix table* [14] with the exception of using the -1 flag.

$$\text{LPM}_k(x)[j] = \begin{cases} \max\{l : \delta_H(x[0..l-1], x[j..j+l-1]) \leq k\} & j \in [1, n-1] \\ \text{-1} & j = 0 \end{cases}$$

$$\text{LSM}_k(x)[j] = \begin{cases} \max\{l : \delta_H(x[n-l..n-1], x[j-l+1..j]) \leq k\} & j \in [0, n-2] \\ \text{-1} & j = n-1 \end{cases}$$

Note that it follows from the definition that the LSM array for a string x is equal to the reverse of the LPM array for the reverse of x, and this logic applies analogously for the prefix array:

$$\mathrm{LSM}_k(x)[j] = \mathrm{LPM}_k(x^R)[n-1-j]$$

$$\mathrm{LPM}_k(x)[j] = \mathrm{LSM}_k(x^R)[n-1-j].$$

Using these identities, we may express the LPM and LSM in terms of the familiar LCE queries, making it possible to apply the Kangaroo method to construct them:

$$\mathrm{LPM}_k(x)[j] = \mathrm{LCE}_k(0,j) \text{ of } x \qquad j \in [1, n-1]$$

$$\mathrm{LSM}_k(x)[j] = \mathrm{LCE}_k(0,n-1-j) \text{ of } x^R \qquad j \in [0, n-2].$$

Using the method for answering LCE_k queries, we can calculate a single value of LPM or LSM in $\mathscr{O}(k)$ time, implying that a total time of $\mathscr{O}(kn)$ would be required to calculate both arrays fully. In fact, the complexity of the full algorithm is bounded by this procedure.

A further set of identities allows us to compute the LPM_{k+1} and LSM_{k+1} arrays from the LPM_k and LSM_k arrays in $\mathscr{O}(1)$ time per entry, such that the arrays are progressively constructed, with each intermediate step yielding valuable information:

$$\mathrm{LPM}_{k+1}(x)[j] = p + 1 + \mathrm{LCE}(p+1, j+p+1) \text{ of } x$$

$$\mathrm{LSM}_{k+1}(x)[j] = s + 1 + \mathrm{LCE}(s+1, n-j+s) \text{ of } x^R$$

$$\text{where } p = \mathrm{LPM}_k(x)[j] \text{ and } s = \mathrm{LSM}_k(x)[n-1-j].$$

After computing $\text{LPM}_{k'}$ and $\text{LSM}_{k'}$, for $0 \leq k' \leq k$, we may determine if a given string x of length $n \geq 2$ is a k-closed string by checking against three conditions for each k', as shown by Lemma 8. Recall that in the case when $n = 0$ or $n = 1$, x is trivially k-closed by definition.

Lemma 8. Given a string x of length $n \geq 2$ and a natural number k, $0 \leq k < n$, x is k-closed if, and only if, there exists some $j \in \{1, \ldots, n-1\}$ and some $k' \in \{0, \ldots, k\}$ such that all the following conditions hold:

1. $j + \text{LPM}_{k'}(x)[j] = n$
2. $\forall i < j,\ \text{LPM}_{k'}(x)[i] < \text{LPM}_{k'}(x)[j]$
3. $\forall i > n-1-j,\ \text{LSM}_{k'}(x)[i] < \text{LSM}_{k'}(x)[n-1-j]$.

See proof C in Appendix C.

Figure 5.7 shows an example of 2-closed border of length $n - j = 10$ found at $j = 5$ for a given string x of length $n = 15$. This corresponds to strings `abbabaabab` and `aababaabab` which are at a Hamming distance of 1.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
$x[j]$	a	b	b	a	b	a	a	b	a	b	a	a	b	a	b	
$\text{LPM}_2[j]$	-1	3	4	7	2	10	4	4	7	2	5	4	3	2	1	
$\text{LSM}_2[j]$	1	2	3	4	5	2	7	6	2	10	2	5	7	2	-1	
$j + \text{LPM}_2[j] = n$	F	F	F	F	F	T	F	F	T	F	T	T	T	T	T	Cond.1
$\text{LPM}_2\_\text{peaks}[j]$	T	T	T	T	F	T	F	F	F	F	F	F	F	F	F	Cond.2
$\text{LSM}_2\_\text{peaks}[n-1-j]$	T	T	T	F	F	T	F	F	F	F	F	F	F	F	F	Cond.3
2-Closed Borders	F	F	F	F	F	T	F	F	F	F	F	F	F	F	F	

▲

Fig. 5.7 2-closed border of length 10 found at $j = 5$ for string x.

Theorem 1. Given a string x of length n over an integer alphabet and a natural number k, $0 < k < n$, the k-closed border of x, if it exists, can be determined in $\mathcal{O}(kn)$ time and $\mathcal{O}(n)$ space.

Proof. By Lemma 8, the time taken to determine whether a string x of length n is k-closed (and determine the k-closed border itself) is bounded by the computation of the $\text{LPM}_{k'}(x)$ and $\text{LSM}_{k'}(x)$ arrays, for all $0 \leq k' \leq k$. For a single k', Condition 1 trivially requires $\mathcal{O}(n)$ time to check across all possible j. Conditions 2 and 3 can be answered for each j in $\mathcal{O}(1)$ time by first preprocessing the $\text{LPM}_{k'}(x)$ and $\text{LSM}_{k'}(x)$ arrays in $\mathcal{O}(n)$ time to determine where the appropriate peaks lie (inspect Figure 5.7 for an example). A total of $\mathcal{O}(kn)$ time is therefore required to check across all possible k' and j, as shown in Lemma 8. The $\text{LPM}_{k'}(x)$ and $\text{LSM}_{k'}(x)$ arrays can be updated from one k' value to the next one and hence the space required is only $\mathcal{O}(n)$.

□

5.5 Implementation

A full implementation of our algorithm was produced and the resulting pseudocode is presented here. The entry point of the algorithm is GETBORDER. This accepts a string x of length n in addition to a parameter k specifying the maximum number of errors. The length that determines the k-closed border is returned. Note that if the string x is not k-closed, the function returns -1. We also make use of additional functions:

REVERSE(x) Standard library function. Accepts a string or array x of length n and returns the reversed string or array, respectively.

LCE(x, i, j) Longest common extension function. Given a string x, this returns the length of the longest common prefix between the ith and jth suffixes of x (details in Chapter 2, section 2.6). Open-source implementations of LCE are available [43].

Algorithm 1 k-Closed Border

```
1:  function GETBORDER(x, n, k)
2:      if n == 0 or n == 1 then                                  ▷ trivial cases
3:          return 0
4:      end if
5:      lpm = GETLPM(x, n, k)                                     ▷ see Algorithm 2 below
6:      lsm = REVERSE(GETLPM(REVERSE(x), n, k))
7:      lpm_peaks = GETPEAKS(lpm)                                 ▷ see Algorithm 3 below
8:      lsm_peaks = GETPEAKS(lsm)
9:      closed_border = −1
10:     for j = 1 to n − 1 do                                     ▷ check 3 conditions for every j
11:         if j + lpm[j] == n and lpm_peaks[j] and lsm_peaks[n − 1 − j] then
12:             closed_border = n − j
13:         end if
14:     end for
15:     return closed_border
16: end function
```

5.6 Experiments

Our k-Closed Border algorithm was implemented as a program to decide whether a given string is indeed a closed string within k Hamming distance errors. We then tested our implementation over numerous input sequences taken from real DNA data.

Algorithm 2 GetLPMx, n, k

```
1:  function GETLPM(x, n, k)
2:      lpm = integer array of length n filled with zeroes
3:      for i = 1 to n − 1 do
4:          for j = 0 to k do
5:              lpm[i] = lpm[i] + LCE(x, lpm[i], i + lpm[i]) + 1        ▷ calculate LCE_k(0, i)
6:              if lpm[i] > n − i then
7:                  break
8:              end if
9:          end for
10:         lpm[i] = lpm[i] − 1                                        ▷ exclude final mismatch
11:     end for
12:     return lpm
13: end function
```

Algorithm 3 GetPeaks*values*

```
1:  function GETPEAKS(values)                                          ▷ values is array of integers
2:      peaks = boolean array with same length as values
3:      max_val = −1
4:      for i = 0 to n − 1 do
5:          if values[i] > max_val then
6:              peaks[i] = True
7:              max_val = values[i]
8:          else
9:              peaks[i] = False
10:         end if
11:     end for
12:     return peaks
13: end function
```

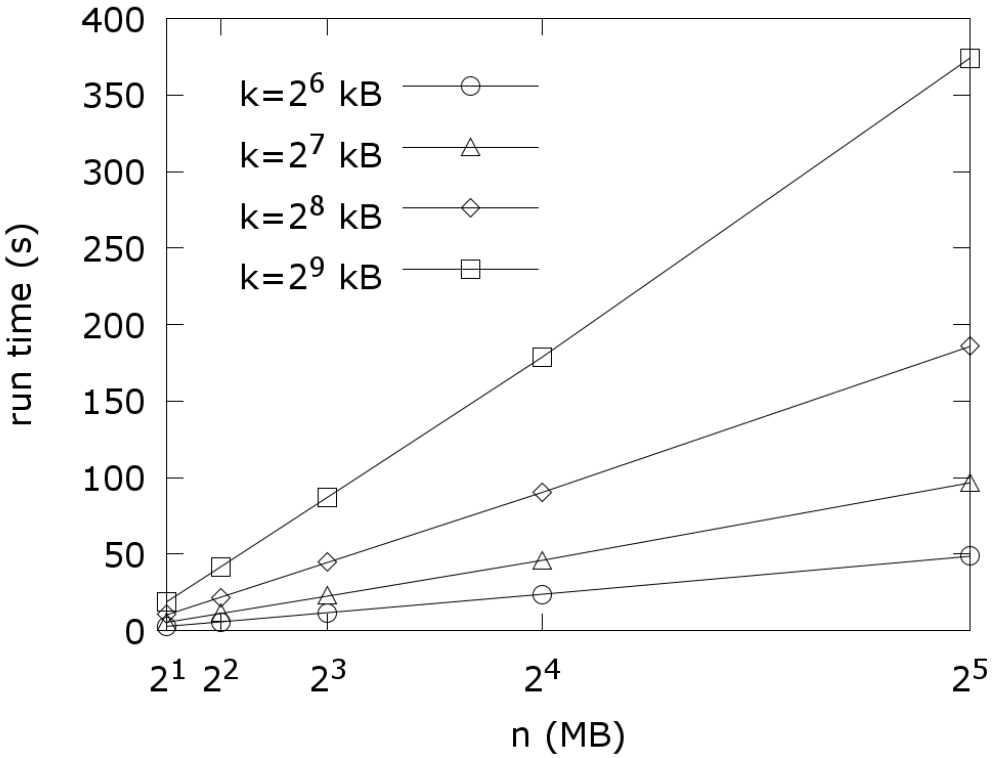
For proof-of-concept experimentation, we made use of the `python` programming language under a GNU/Linux operating system. All experiments were conducted on a Desktop PC using one core of an Intel Core CPU i5-6600K at 3.50GHz.

The task in our experiments was to establish whether the elapsed time of the implemented algorithm does indeed grow linearly with n and linearly with k. As input data-sets for the experiments, we used arbitrarily extracted fragments from Chromosome 1 of the human reference genome GRCh37. The value of k and n were varied exponentially, and the total time taken recorded (we use the standard convention of 1 DNA letter occupying 1 byte of space).

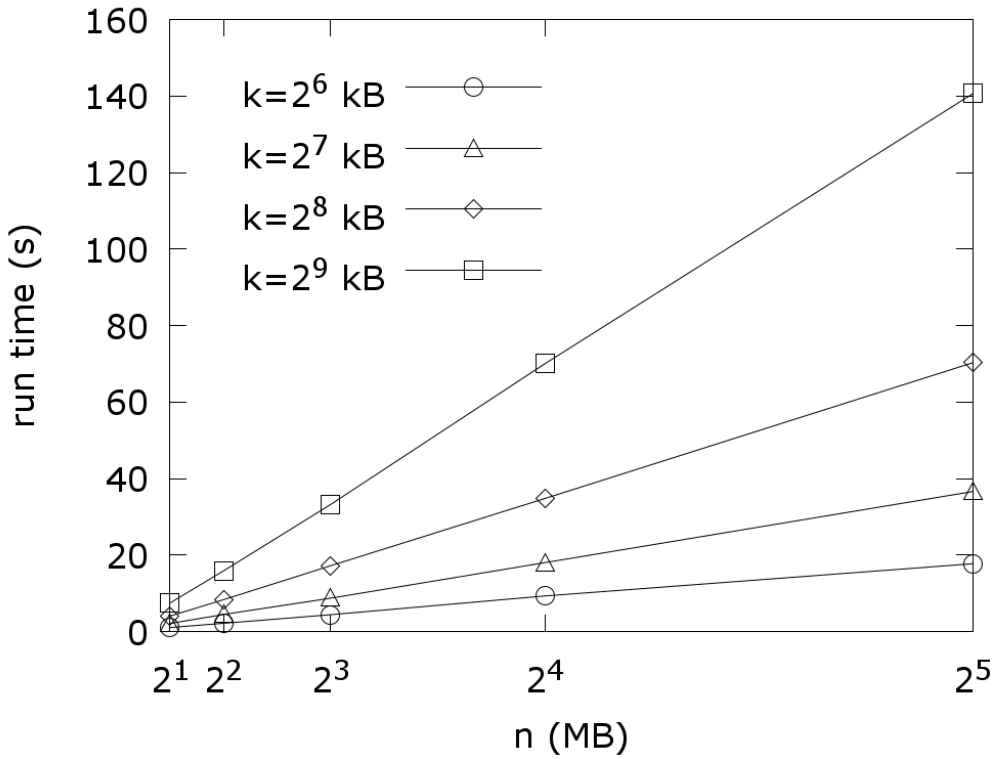
In addition, we found during experimentation that although an $\mathcal{O}(n)$-sized data structure for Range Minimum Queries (RMQs, see [7]) was being used to answer LCE queries, it was slow in practice. We therefore experimented using an $\mathcal{O}(n \log n)$-sized data structure for RMQs [15], which was indeed faster in practice (see also [7] in this regard). In Table 5.1, we provide a guide to the experiments we conducted. The presented experiments (Figures 5.8–5.9) fully confirm our theoretical findings: i.e. the elapsed time of the implemented algorithm grows linearly with n and linearly with k.

	n vs. run time	k vs. run time
$\mathcal{O}(n)$ RMQs	Figure 5.8a	Figure 5.9
$\mathcal{O}(n \log n)$ RMQs	Figure 5.8b	

Table 5.1 Guide for experimental figures.



(a) n vs. run time with $\mathcal{O}(n)$-sized RMQs data structure.



(b) n vs. run time with $\mathcal{O}(n \log n)$-sized RMQs data structure.

Fig. 5.8 n vs. run time with $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$-sized RMQs data structure.

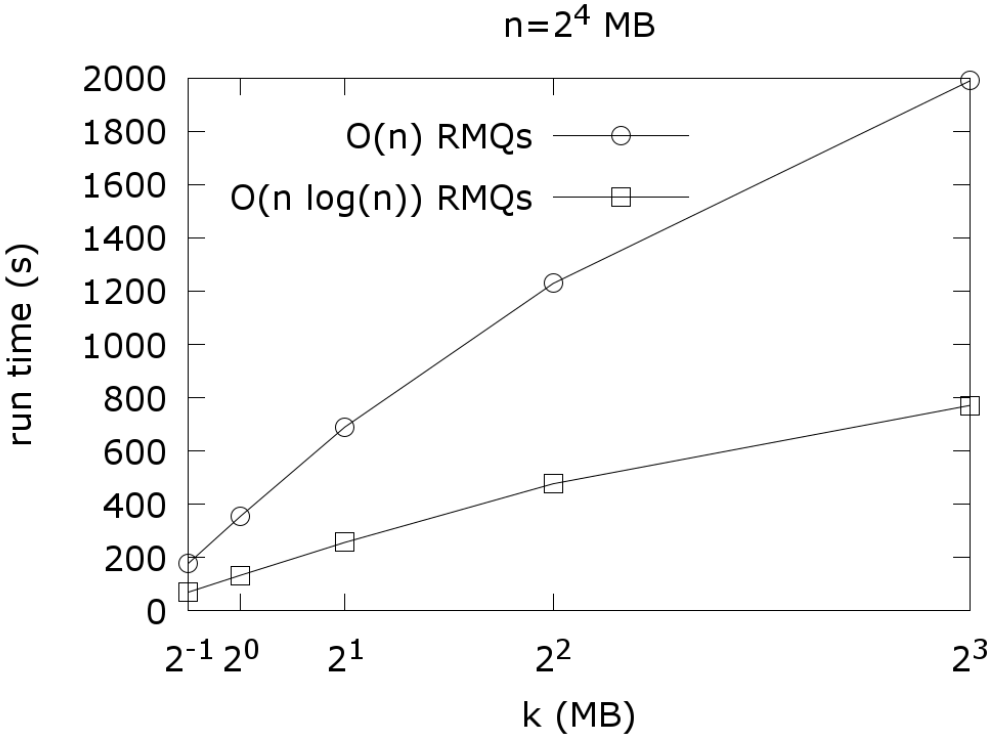


Fig. 5.9 k vs. run time with $\mathscr{O}(n)$- and $\mathscr{O}(n\log n)$-sized RMQs data structures

5.7 Final remarks

We have presented an algorithm and proof-of-concept experiments for finding the k-closed border of a given string x of length n derived from an integer alphabet within Hamming distance k. The proposed algorithm was dependent on building two simple data structures, namely, $\mathrm{LPM}_k(x)$ and $\mathrm{LSM}_k(x)$. Given these data structures, it takes a further $\mathscr{O}(n)$ time to determine the k-closed border. The required space is $\mathscr{O}(n)$.

The main improvement could therefore be in the construction of these two tables, currently requiring $\mathscr{O}(kn)$ time. Decreasing this time complexity appears to be a reasonable, however non-trivial, goal for any future work on this problem, since any faster computation of

$\mathrm{LPM}_k(x)$ and $\mathrm{LSM}_k(x)$ would imply a major breakthrough in approximate string matching under the Hamming distance model.

Chapter 6

The RMQs or LCA Queries in Practice for Small Batch

The work presented in this chapter is published as: M. Alzamel, P. Charalampopoulos, C. Iliopoulos, and S. P. Pissis. How to answer a small batch of rmqs or LCA queries in practice. In *Combinatorial Algorithms - 28th International Workshop, IWOCA 2017, Newcastle, NSW, Australia, July 17-21, 2017,Revised Selected Papers*, pages 343–355, 2017.

6.1 Background and Contributions

6.1.1 Background

The RMQ problem (defined in Section: 2.12) and the linearly equivalent LCA problem (defined in Section: 2.7) [15] have been studied extensively and there are a great number of optimal algorithms for solving them.

In the *Range Minimum Query* (RMQ) problem, we are given an array A of n integers and we are asked to answer queries of the following type: for indices i and j between 0 and $n-1$, query $\mathrm{RMQ}_A(i,j)$ returns the index of a minimum element in the subarray $A[i..j]$.

It was first shown by Harel and Tarjan [51] that a tree can be pre-processed in $\mathscr{O}(n)$ time, so that LCA queries can be answered in $\mathscr{O}(1)$ time per query. A major breakthrough for a good practical constant-time LCA-algorthim was made by Berkman and Vishkin in [18].

In the *Lowest Common Ancestor* (LCA) problem, we are given a rooted tree T and a tuple of two nodes (u,v), and we are asked to find the lowest node in T that has both u and v as descendants. Farach and Bender [15] further simplified this problem by showing that the RMQ problem is linearly equivalent to the LCA problem (shown also in [37]). Due to the reduction, the constants remained quite large, however, making these algorithms impractical in most real-life scenarios. To this end, Fischer and Heun [32] presented yet another optimal, but also direct, algorithm for the RMQ problem. The same authors (but also others see [55]) showed that, due to large constants in the pre-processing and querying time, implementations of this algorithm are often slower than implementations of the naive ones. Strenuous efforts to engineer algorithms for these solutions have been made in [29].

6.1.2 Contributions

In this thesis we try to address a variation of the RMQ problem that seeks to answer a relatively small batch of RMQs efficiently. This version of the problem is a core computational task in many real-life applications such as in *object inheritance* during static compilation of code [17] or in several *string matching* problems (see Section 6.4 for some). By a *small*

batch, we mean that the number q of the queries is $o(n)$ *and* we have them all in advance. It is therefore not relevant to build an $\Omega(n)$-sized data structure or spend $\Omega(n)$ time to build a more succinct one. It is well-known, among practitioners and elsewhere, that these data structures carry high hidden constants in both their pre-processing and querying time (note that when, $q = \Omega(n)$ one can use these data structures for this computation). We would thus like to answer this batch efficiently in practice. By *efficiently in practice*, we mean that we (ultimately) want to spend $n + \mathcal{O}(q)$ time and $\mathcal{O}(q)$ space. We write n to stress that the number of operations per entry of A should be a very small constant; e.g. scanning the array just once or twice. In what follows, we show how existing algorithms can be easily modified to satisfy these conditions. Experimental results presented here highlight the practicality of this scheme. The most significant improvement obtained is for answering a small batch of LCA queries.

6.2 Preliminaries and Definitions

A *Euler tour* (see [92]) of a connected, directed graph G = (V, E), is said to be a cycle that traverses each edge of graph G exactly once, although it may visit a vertex more than once. Figure 6.1 shows an example of traversing tree T in the form of a Euler tour.

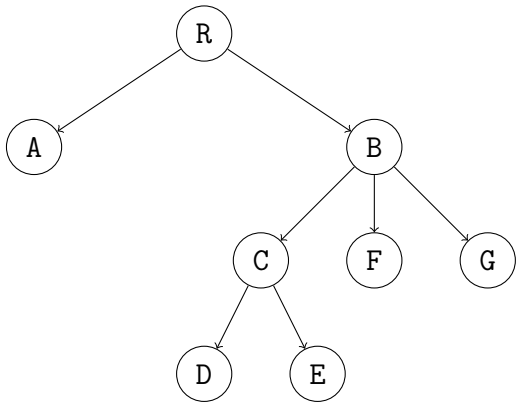
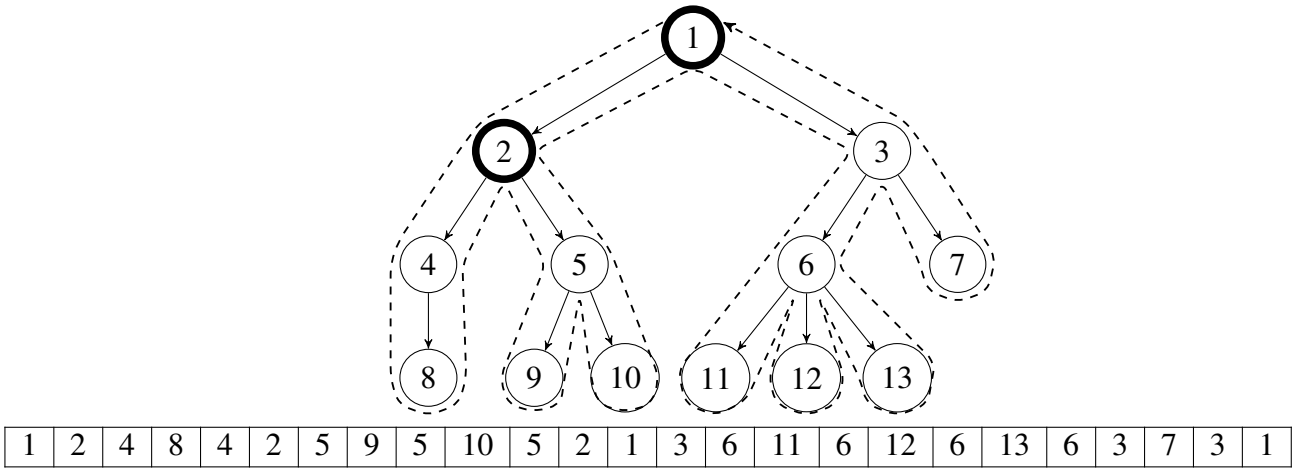


Fig. 6.1 The Euler tour of the tree T is `R A R B C D C E C B F B G B R`.

Reduction of LCA to RMQ: It is well-known that an RMQ instance A can be obtained from an LCA instance on a tree T by writing down the depths of the nodes visited during an *Euler tour* of T. That is, A is obtained by listing all node-visitations in a depth-first search (DFS) traversal of T starting from the root. The LCA of two nodes translates to an RMQ (where we compare nodes based on their level) between the first occurrences of these nodes in A, (see [15] for the details). An example of a translation of a tree T to an array A using a Euler tour to answer $\text{LCA}_T(4,7)$ and LCA_T=(8,10) is shown in Figure 6.2. Given a tree T, a Euler tour is conducted on tree T starting from the root node, leading to an array A as [1,2,4,8,4,2,5,9,5,10,5,2,1,3,6,11,6,12,6,13,6,3,7,1].



$$\mathrm{LCA}_T(4,7) = 1 \text{ and } \mathrm{LCA}_T(8,10) = 2$$

Fig. 6.2 Example of reduction from LCA to RMQ

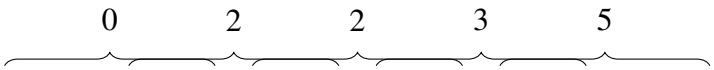
A *Sparse table* is a data structure designed by Bender and Farach-Colton [16] to answer RMQ queries in respect to an array A of length n in $\mathscr{O}(n \log n)$ processing time where queries take $\mathscr{O}(1)$ time. This technique uses dynamic programming, whereby it splits the input array A into 2^j blocks, where $0 \leq j \leq \log n$. It stores the result in a look up table denoted to sparse table M[0- $\log n$] [0-$n-1$], where $M[i][j]$ stores the index of the minimum range starting at position i of length 2^j.

Once sparse table M is completed, this technique uses that table to answer the RMQ(i,j). The idea is to select two blocks that entirely cover the interval $[i \ldots j]$ and then find the minimum between them. Let $k = \lfloor log(j-i+1) \rfloor$. More formally, $\mathrm{RMQ}(i,j) = min = (M[i][k], M[j-2^k+1][k])$.

Example 25. Given an array A, we split the array into blocks with size 2^0, as shown in Table 6.1, then we re-split array A with a 2^1 and 2^2 block size, as shown in Table 6.2 and Table 6.3 respectively. We create a two dimensional array M [0- 2] [0-5] to store the minimum index in the sub array of size 2^j in $M[i][j]$. This process will be repeated for each i in every sub array for Table 6.1, Table 6.2 and Table 6.3 to obtain the sparse table in Table 6.4. To answer RMQ(0,5)= $M[5-2^2+1][2] = 2$, where $A[2] = 1$. Ultimately, RMQ(0,5)=2.

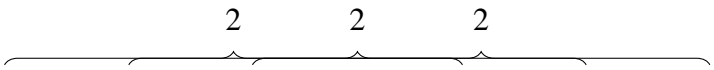
0	1	2	3	4	5
4	6	1	5	7	3

Table 6.1 Splitting array A into blocks with size $2^0 = 1$



0	1	2	3	4	5
4	6	1	5	7	3

Table 6.2 Splitting array A into blocks with size $2^1 = 2$



0	1	2	3	4	5
4	6	1	5	7	3

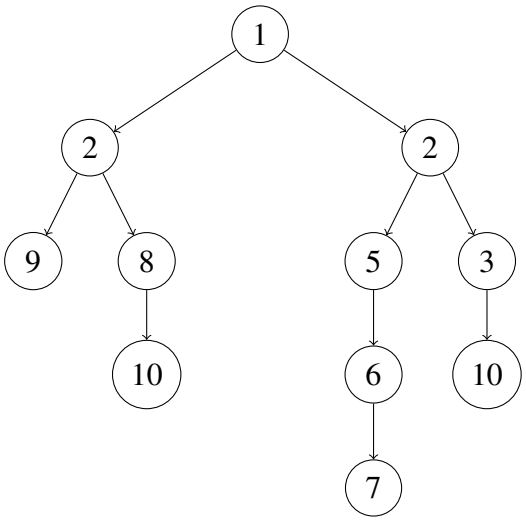
Table 6.3 Splitting array A into blocks with size $2^2 = 4$

i \ j	0	1	2
0	0	0	2
1	1	2	2
2	2	2	2
3	3	3	
4	4	5	
5	5		

Table 6.4 Sparse table M of array A

Given an array A of n numbers, its *Cartesian tree* is defined as follows: the root of the Cartesian tree is $A[i] = \min\{A[0],\ldots,A[n-1]\}$, its left subtree is computed recursively on $A[0],\ldots,A[i-1]$ and its right subtree on $A[i+1],\ldots,A[n-1]$, as introduced by Vuillemin in [95]. An LCA instance can be obtained from an RMQ instance on an array A by letting T be the Cartesian tree of A that can be constructed in $\mathcal{O}(n)$ time, as presented by Gabow and Tarjan [37].

Example 26. Given an array $A[9\ ,2\ ,8\ ,10\ ,1\ ,5\ ,6\ ,7\ ,2\ ,10\ ,3]$, the Cartesian tree T of A is shown below in Figure 6.3:



i	0	1	2	3	4	5	6	7	8	9	10
A[i]	9	2	8	10	1	5	6	7	2	10	3

Fig. 6.3 The Cartesian tree T of array $A[9\ ,2\ ,8\ ,10\ ,1\ ,5\ ,6\ ,7\ ,2\ ,10\ ,3]$

The RMQ Batch problem can be defined as follows.

RMQ Batch

Input: An array A of size n of numbers and a list Q of q pairs of indices (i, j), $0 \leq i \leq j \leq n-1$

Output: $\mathrm{RMQ}_A(i, j)$ for each $(i, j) \in Q$

The LCA Queries Batch problem can be defined as follows.

LCA Queries Batch

Input: A rooted tree T with n labelled nodes $0, 1, \ldots, n-1$ and a list Q of q pairs of nodes (u, v)

Output: $\mathrm{LCA}_T(u, v)$ for each $(u, v) \in Q$

6.3 Algorithm

Our computational model. We assume the word-RAM model with word size $w = \Omega(\log n)$. For the RMQ Batch problem, we assume that we are given a rewritable array A of size n, each entry of which may be increased by n and still fit in a computer word. For the LCA Queries Batch problem, we assume that we are given (an $\mathcal{O}(n)$-sized representation of) a rewritable tree T allowing constant-time access to (at least) the nodes of T that are in some query in Q (see the representation in [41], for instance). All presented algorithms are deterministic.

6.3.1 Contracting the Input Array

Consider any two adjacent array entries $A[i]$ and $A[i+1]$. Observe that if no query in Q starts or ends at i or at $i+1$, then, if $A[i] \neq A[i+1]$, $\max(A[i], A[i+1])$ will never be the answer to any of the queries in Q. Hence, the idea is that we want to contract array A, so that each block

that does not contain the left or right endpoint of any query gets replaced by one element: its minimum. A similar idea, based on sorting the list Q, has been considered in the *External Memory* model [4] (see also [10]). In this section, we present a solution for our computational model, which avoids using $\Omega(n)$ space or time, but also avoids using $\Omega(\text{sort}(Q))$ time.

There are some technical details in order to update the queries for A into queries for the new array using only $\mathcal{O}(q)$ time and extra space. We first scan the array A once and find $\mu = \max_i A[i]$. We also create two auxiliary arrays $Z_0[0..2q-1]$ and $Z_1[0..2q-1]$. For each query $(i, j) \in Q$ we mark positions i (and j) in the array A as follows. If $A[i] \leq \mu$, then i has not been marked before. Let this be the k-th position, $k > 0$, that gets marked (we just store a counter for that). We store $A[i]$ in $Z_0[\mu + k \mod 2q]$ and replace the value that is stored in $A[i]$ by $\mu + k$. We also start a linked list at $Z_1[\mu + k \mod 2q]$, where we insert a pointer to query (i, j), so that we can update it later. If $A[i] > \mu$, then the position has already been marked; we just add a pointer to the respective query in the linked list starting at $Z_1[A[i] \mod 2q]$.

We then scan array A again and create a new array A_Q as follows: for each marked position j (i.e. $A[j] > \mu$), we copy the original value (i.e. $Z_0[A[j] \mod 2q]$) in A_Q, while each maximal block in A that does not contain a marked position is replaced by a single entry—its minimum. When we insert the original entry of a marked position j of A (i.e. $Z_0[A[j] \mod 2q]$) in A_Q at position p, we go through the linked list that is stored in $Z_1[A[j] \mod 2q]$, where we have stored pointers to all the queries of the form (i, j) or (j, k), and in each of them replace j with p . Thus, after we have scanned A, for each query $(i, j) \in Q$ on A, we will have stored the respective pair (i', j') on A_Q.

While creating A_Q, we also store in an auxiliary array the function $f : \{0,1,\ldots,|A_Q|-1\} \rightarrow \{0,1,\ldots,n-1\}$ between positions of A_Q and the respective original positions in A.

Now notice that A_Q and the auxiliary arrays are all of size $\mathscr{O}(q)$, since in the worst case we mark $2q$ distinct elements of A and contract $2q+1$ blocks that do not contain a marked position (we can actually throw away everything before the first marked position and everything after the last marked position and get $4q-1$ instead). The whole procedure takes $n+\mathscr{O}(q)$ time and $\mathscr{O}(q)$ space. Note that if $\mathrm{RMQ}_{A_Q}(i',j') = \ell$ then $\mathrm{RMQ}_A(i,j) = f(\ell)$.

We can finally retrieve the original input array, if required, by replacing $A[f(j)]$ by $A_Q[j]$ for every j in the domain of f in $\mathscr{O}(q)$ time. A full example about how the above contracting array process is shown below in example 27

Example 27. Assume we are given array A and $Q = \{(4,18),(0,6),(6,10)\}$, $q = 3$, and $1 \leq k \leq (2q-1)$, we scan array A to find $\mu = max_i(A[i]) = 38$, and we scan for each i and j $\in Q$ to update array $A[i]$ with $\mu + k$ if it has not been marked before, as shown in Table 6.5. We update $Z_0[\mu + k \mod 2q]$ with $A[Q[k]]$ and we add a pointer in the linked list $Z_1[\mu + k \mod 2q]$ to the corresponding position of $Q(i,j)$ that consists $Z_0[\mu + k \mod 2q]$, Z_0 and Z_1 are presented in Table 6.6 and Table 6.7. Later, we create A_Q and A_F and also, we have Q' to store the respective positions of Q in A_Q, as illustrated in Table 6.8 and Table 6.9, respectively.

We denote $[\mu + k \mod 2q]$ as *index*:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$A[i]$	17	22	38	4	5	8	2	8	9	21	0	12	8	7	13	3	6	14	1
$A[i] = \mu + k$	41	22	38	4	39	8	42	8	9	21	43	12	8	7	13	3	6	14	40

Table 6.5 Scanning array A and updating $A[i]$ with $\mu + k$

Z_0	index 0	index 1	index 2	index 3	index 4	index 5
	2	0		5	1	17

Table 6.6 Storing the original values of $A[Q(i)]$ and $A[Q(j)]$ in auxiliary array Z_0

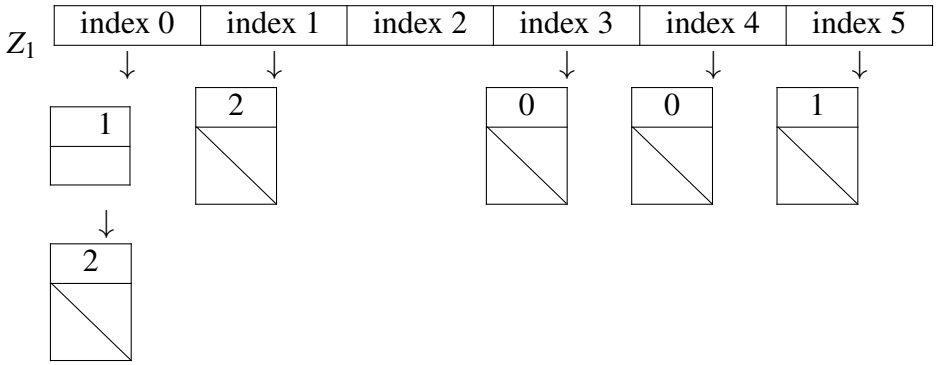


Table 6.7 Storing the corresponding position of Q in auxiliary array Z_1

i	0	1	2	3	4	5	6	7	8
$A_Q[i]$	17	4	5	8	2	8	0	3	1
$A_F[i]$	0	3	4	5	6	7	10	15	18

Table 6.8 The contracted array A_Q and auxiliary array A_F

$Q\prime(i\prime, j\prime)$	$(2,8)$	$(0,4)$	$(4,6)$

Table 6.9 The respective new positions of queries $\in Q$ mapped to $Q\prime$ according to A_Q

6.3.2 Small RMQ Batch

An $n + \mathcal{O}(q \log q)$-time and $\mathcal{O}(q)$-space algorithm

The algorithm presented in this section is a modification of the *Sparse Table* algorithm by Bender and Farach-Colton [15] applied on array A_Q; we denote it by ST-RMQ. The modification is based on the fact that **(i)** we do not want to consume $\Omega(q \log q)$ extra space to answer the q queries; and **(ii)** we do not necessarily want to do all the pre-processing work of the algorithm in [15], which is designed to answer any of the $\Theta(q^2)$ possible queries online. We denote this modified algorithm by ST-RMQ$_{\mathsf{CON}}$ and formalise it below.

Algorithm 4 ST-RMQ$_{\mathsf{CON}}(A, Q)$

$A_Q \leftarrow Contract(A, Q)$
Store function f; store (i', j') for every $(i, j) \in Q$
for each $(i, j) \in Q$ **do**
    if i=j **then**
        Report((i,i),i)
    else
        Add (i, j) in bucket $B_{\lfloor \log(j'-i') \rfloor}$
    end if
end for
$t \leftarrow \max\{r | B_r \neq \emptyset\} + 1$
for $m = 0$ to $|A_Q| - 1$ **do**
    $D[m] \leftarrow (A_Q[m], m)$
end for
for $k = 0$ to $t - 1$ **do**
    for each $(i, j) \in B_k$ **do**
        $(a, p) \leftarrow \min(D[i'], D[j' - 2^k + 1])$
        Report(i, j), $f(p)$
    end for
    for $m = 0$ to $|A_Q| - 1$ **do**
        if $m + 2^k \leq |A_Q| - 1$ **then**
            $D[m] \leftarrow \min(D[m], D[m + 2^k])$
        end if
    end for
end for

The idea is first to put each $(i,j) \in Q$ with $i \neq j$ in a bucket B_k based on the k for which $2^k \leq j' - i' < 2^{k+1}$—we can have at most $\lceil \log(|A_Q| - 1) \rceil$ such buckets. In this process, if we find queries of the form $(i,i) \in Q$, we answer them on the spot. We can do this in $\mathcal{O}(q)$ time. We then create an array D of size $|A_Q|$ where we will store 2-tuples (a,p). In Step k, $D[m]$ will store the minimum value across $A_Q[m..m+2^k-1]$, as well as the position p, $m \leq p < m+2^k$ where it occurs. We initialise it as $D[m] = (A_Q[m], m)$ and we will then update it by utilising the *doubling technique*. At Step 0 we answer all (trivial) queries that are stored in B_0; they are of the form $(i, i+1)$ and the answer can be found by looking at $\min(D[i'], D[i'+1])$—note that we compare elements of D lexicographically. When we are done with B_0 we have to update D by setting $D[m] = \min(D[m], D[m+2^0])$ for all $m < |A_Q| - 1$.

Generally, in Step k, we answer the queries of B_k as follows. For query (i,j), we find the answer by obtaining $\min(D[i'], D[j' - 2^k + 1] = (a,p)$. We then return $f(p)$. The point is that $\{i', \ldots, i' + 2^k - 1\} \cup \{j' - 2^k + 1, \ldots, j'\} = \{i', \ldots, j'\}$. When we are done with B_k we set $D[m] = \min(D[m], D[m+2^k])$ if $m + 2^k \leq |A_Q| - 1$.

We do this until we have gone through all t non-empty buckets (i.e. $t = \max\{r | B_r \neq \emptyset\} + 1$). Updating D takes $\mathcal{O}(q)$ time in each step, and we need in total $\mathcal{O}(q)$ time for the queries. We thus need $\mathcal{O}(qt)$ time for this part of the algorithm. Since $t = \max\{\lfloor \log(j' - i') \rfloor | (f(i'), f(j')) \in Q)\} = \mathcal{O}(\log q)$, this time is $\mathcal{O}(q \log q)$. The overall time complexity of the algorithm is thus $n + \mathcal{O}(q \log q)$. Notably, the space required is only $\mathcal{O}(q)$ because we overwrite D in each step.

Example 28. To answer RMQ for all queries in Q we will use arrays A_Q, A_F and Q' in Example 27 to build a bucket B that holds the index of each $i', j' \in Q'$ at B_k where k= $\lfloor \log(j\prime - i\prime) \rfloor$, as shown in Table 6.10 below. Then, for every $0 \leq k \leq 2$ in B_k we will answer the RMQ in $Q\prime[B_k]$ using array $min(D(i', j'))$ and we will retrieve the original index using A_F. Finally, in the step k we update D tuples for every m using doubling technique. Table 6.11 below, consists 2-tuples (a, p) data structure D with size $|A_Q|$ to store in each step k the RMQ$(m, m+2^k-1)$ and reporting the RMQ$(i\prime, j\prime)$ as well as RMQ(i, j).

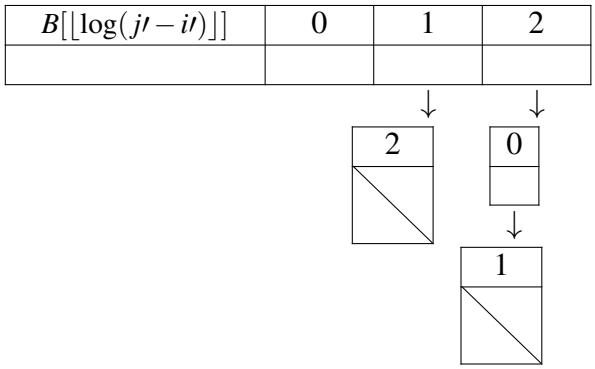


Table 6.10 A data structure B with $\lceil log(|A_Q|-1) \rceil$ buckets

k	m	0	1	2	3	4	5	6	7	8
0	(D_a, D_p)	(17,0)	(4,1)	(5,2)	(8,3)	(2,4)	(8,5)	(0,6)	(3,7)	(1,8)
0	(D_a, D_p)	(4,1)	(4,1)	(5,2)	(2,4)	(2,4)	(0,6)	(0,6)	(1,8)	(1,8)
1	min$(i\prime, j\prime)$=(4,6)		**6**		$A_F(6)$		min(i, j)=(6,10)		**10**	
1	(D_a, D_p)	(4,1)	(2,4)	(2,4)	(0,6)	(0,6)	(0,6)	(0,6)	(1,8)	(1,8)
2	min$(i\prime, j\prime)$=(2,8)		**6**		$A_F(6)$		min(i, j)=(4,18)		**10**	
2	min$(i\prime, j\prime)$=(0,4)		**4**		$A_F(4)$		min(i, j)=(0,6)		**6**	

Table 6.11 A data structure D with size $|A_Q|$

$n + \mathcal{O}(q)$-time and $\mathcal{O}(q)$-space algorithms

Offline-based algorithm: An LCA instance can be obtained from an RMQ instance on an array A by letting T be the Cartesian tree of A that can be constructed in $\mathcal{O}(n)$ time [37]. It is easy to see that $\text{RMQ}_A(i, j)$ in A translates to $\text{LCA}_T(A[i], A[j])$ in T. The first step of this algorithm is to create array A_Q in $n + \mathcal{O}(q)$ time, similarly to algorithm ST-RMQ$_{\text{CON}}$. The second step is to construct the Cartesian tree T_Q of A_Q in $\mathcal{O}(q)$ time and extra space. Finally, we apply the offline algorithm by Gabow and Tarjan [38] to answer q LCA_{T_Q} queries in $\mathcal{O}(q)$ time and extra space. This takes overall $n + \mathcal{O}(q)$ time and $\mathcal{O}(q)$ extra space. We denote this algorithm by OFF-RMQ$_{\text{CON}}$. We denote by OFF-RMQ the same algorithm applied on array A.

Online-based algorithm: The first step of this algorithm is to create array A_Q in $n + \mathcal{O}(q)$ time similarly to algorithm ST-RMQ$_{\text{CON}}$. We can then apply the algorithm by Fischer and Heun [32] on array A_Q to obtain overall an $n + \mathcal{O}(q)$-time and $\mathcal{O}(q)$-space algorithm. We denote this algorithm by ON-RMQ$_{\text{CON}}$. We denote by ON-RMQ the same algorithm applied on array A.

Note that in the case when $q = \Omega(n)$, i.e. the batch is not so small, we can choose to apply algorithm OFF-RMQ or algorithm ON-RMQ on array A directly, thus obtaining an algorithm that always works in $n + \mathcal{O}(q)$ time and $\mathcal{O}(\min\{n, q\})$ extra space. We therefore obtain the following result asymptotically.

Theorem 2. The RMQ Batch problem can be solved in $n + \mathcal{O}(q)$ time and $\mathcal{O}(\min\{n, q\})$ extra space.

6.3.3 Small LCA Queries Batch

In the LCA problem, we are given a rooted tree T having n labelled nodes and we are asked to answer queries of the following type: for nodes u and v, query $\mathrm{LCA}_T(u,v)$ returns the node furthest from the root that is an ancestor of both u and v. Gabow and Tarjan [38] developed a time-optimal algorithm to answer a batch Q of q LCA queries in $\mathscr{O}(n+q)$ time and $\mathscr{O}(n)$ extra space. We denote this algorithm by OFF-LCA. In this section, we present a simple but *non-trivial* algorithm for improving this, for $q = o(n)$, to $n+\mathscr{O}(q)$ time and $\mathscr{O}(q)$ extra space. It is well-known (see [15] for the details) that an RMQ instance A can be obtained from an LCA instance on a tree T by writing down the depths of the nodes visited during a *Euler tour* of T, as described in section 6.2.

We proceed largely as in Section 6.3.1. For each query $(u,v) \in Q$, we mark nodes u (and v) in T as follows. If $u < n$ then u has not been marked before. Let this be the k-th node, $k > 0$, that gets marked (we just store a counter for that). We also create two arrays $Z_0[0\,..\,2q-1]$ and $Z_1[0\,..\,2q-1]$. We store u in $Z_0[n-1+k \mod 2q]$ and replace u by $n-1+k$. We also start a linked list at $Z_1[n-1+k \mod 2q]$, where we insert a pointer to query (u,v), so that we can update it later. If $u > n-1$, the node has already been marked, and we just add a pointer to the respective query in the linked list starting at $Z_1[u \mod 2q]$.

We then do a single DFS traversal on T and create two new arrays E_Q and L_Q as follows. When a marked node v (i.e. $v > n-1$) is visited for the *first time*, we write down in E_Q its original value (i.e. $Z_0[v \mod 2q]$), while for each maximal sequence of visited nodes that are not marked we write down a single entry—the one with the *minimum tree level*. At the same time, we store in $L_Q[v]$ the level of the node added in $E_Q[v]$. While creating E_Q, we also

store in an auxiliary array the function $f : \{0,1,\ldots,|E_Q|-1\} \to \{0,1,\ldots,n-1\}$ between positions of E_Q and the respective node labels in T.

When we insert the original entry of a marked node u of T (i.e. $Z_0[u \mod 2q]$) in E_Q at position p, we go through the linked list that is stored in $Z_1[u \mod 2q]$, where we have stored pointers to all the queries of the form (u,v) or (w,u), and replace u with p in each of these queries. Thus, after we have finished the traversal on T, for each LCA query $(u,v) \in Q$ on T, we will have stored the respective RMQ pair (u',v') on L_Q; where u' (resp. v') corresponds to the *first occurrence* of node u (resp. v) in the traversal. Thus we traverse T only once.

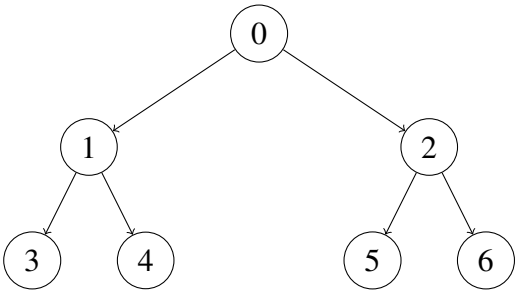
Now notice that E_Q and the auxiliary arrays are all of size $\mathcal{O}(q)$, since in the worst case we mark $2q$ distinct nodes of T and contract $2q+1$ sequences of visited nodes that do not contain a marked node (we can actually throw away everything before the first marked node and everything after the last marked node and get $4q-1$ instead). The whole procedure takes $n+\mathcal{O}(q)$ time and $\mathcal{O}(q)$ space. We are now in a position to apply algorithm ON-RMQ on L_Q to obtain the final bound. To answer the queries, note that if $\mathrm{RMQ}_{L_Q}(u',v') = \ell$ then $\mathrm{LCA}_T(u,v) = E_Q[\ell]$. We denote this algorithm by $\mathsf{ON\text{-}LCA_{CON}}$. Alternatively, we can apply algorithm ST-RMQ on L_Q to solve this problem in $n+\mathcal{O}(q\log q)$ and $\mathcal{O}(q)$ extra space; we denote this algorithm by $\mathsf{ST\text{-}LCA_{CON}}$.

We can finally retrieve the original input tree, if required, by replacing node $f(v)$ by $E_Q[v]$ for every v in the domain of f in $\mathcal{O}(q)$ time.

Note that in the case when $q = \Omega(n)$, i.e. the batch is not so small, we can choose to apply algorithm OFF-LCA on tree T directly, thus obtaining an algorithm that always works

in $n+\mathcal{O}(q)$ time and $\mathcal{O}(\min\{n,q\})$ extra space. We therefore obtain the following result asymptotically.

Example 29. In Figure 6.4, given a rooted tree T and LCA queries $Q=\{(3,4),(0,2),(1,2)\}$, we transform tree T to array A via a DFS in-order traversal and we record the indices of the node labels $0,1,\ldots,6$ in an array *invA*. Also, we create auxiliary Q_{lca} to store the transformation of the Q according to the array A, as shown in tables below. Later, we apply, for example, ST-RMQ$_{\mathsf{CON}}$ to answer the queries $\in Q_{lca}$. Finally, we transform the RMQ answers back to node labels using an array A.



i	0	1	2	3	4	5	6
A	3	1	4	0	5	2	6
invA	3	1	5	0	2	4	6

i	0	1	2
$Q_{lca}(L,R)$	(0,2)	(3,5)	(1,5)
RMQ$_{lca}$	1	3	3
LCA$_{(u,v)}$	1	0	0

Fig. 6.4 Steps involved in answering LCA queries in $n+\mathcal{O}(q)$ time

Theorem 3. The LCA Queries Batch problem can be solved in $n+\mathcal{O}(q)$ time and $\mathcal{O}(\min\{n,q\})$ extra space.

6.4 Applications

We consider the well-known application of answering q LCA queries on the suffix tree of a string. The *suffix tree* $\mathscr{T}(S)$ of a non-empty string S of length n is a compact tree representing all suffixes of S (see [23], for details). The nodes of the tree, which become nodes of the suffix tree, are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes, starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Then, each node of the tree can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. The *path-label* of a node v is the concatenation of the edge labels along the path from the root to v. The nodes whose path-label corresponds to a suffix of S are called *terminal*. Given two terminal nodes, u and v, in $\mathscr{T}(S)$, representing suffixes $S[i..n-1]$ and $S[j..n-1]$, the *string depth* of node $\mathrm{LCA}_{\mathscr{T}(S)}(u,v)$ corresponds to the *length* of their longest common prefix, also known as their longest common extension (LCE) [55].

In many textbook solutions for classical string matching problems (e.g. maximal palindromic factors, approximate string matching with k-mismatches, approximate string matching with k-differences, online string search with the suffix array, etc.) we have that $q = \Omega(n)$ and/or the queries have to be answered *online*. In other algorithms, however, q can be *much smaller* on average (in practice) and the queries can be answered *offline*. We describe a few such solutions here. The common idea, as in many fast average-case algorithms, is to minimise the number of queries by filtering out queries that can never lead to a valid solution.

Text indexing. Suppose we are given the suffix tree $\mathscr{T}(S)$ of a text S of length n, and we are asked to create the suffix links for the internal nodes. This may be necessary if the construction algorithm does not compute suffix links (e.g. construction via suffix array) but they are needed for an application of interest [70]. The *suffix link* of a node, v, with path-label αy is a pointer to the node path-labelled y, where $\alpha \in \Sigma$ is a single letter and y is a string. The suffix link of v exists if v is a non-root internal node of T. The suffix links can be computed as follows. The first step is to mark each internal node, v, of the suffix tree with a pair of leaves (i, j) such that the leaves labelled i and j are in subtrees rooted at different children of v. This can be done by a DFS traversal of the tree. (Note that if an internal node v has only one child then it must be terminal; assume that it represents the suffix $S[t\,..\,n-1]$. We thus create a suffix link to the node representing $S[t+1\,..\,n-1]$). Given an internal node v marked with (i, j), note that $v = \mathrm{LCA}_{\mathscr{T}(S)}(i, j)$, and let αy be its path-label. To create the suffix link from v, node u with path-label y can be obtained by the query $\mathrm{LCA}_{\mathscr{T}(S)}(i+1, j+1)$. We can create a batch of LCA queries consisting of all such pairs. Note that in randomly generated texts, the number of internal nodes of $\mathscr{T}(S)$ is $\mathscr{O}(n/h)$ on average, where h is the alphabet's entropy [87]; thus the standard $\Theta(n)$-time and $\Theta(n)$-space solution to this problem, building the LCA data structure over $\mathscr{T}(S)$ [15], is not satisfactory.

Finding frequent gapped factors in texts. We are given a text S of length n, and positive integers ℓ_1, ℓ_2, d, and $k > 1$. The problem is to find all couples (u, v), such that string uwv, for *any* string w (known as a *gap* or *spacer*), $|w| = d$, occurs in S at least k times, $|u| = \ell_1$, $|v| = \ell_2$ [56, 85]. The first step is to build $\mathscr{T}(S)$. We then locate all subtrees rooted at an

explicit node with a string depth of at least ℓ_1, and whose parent has a string depth less than ℓ_1, corresponding to factors u repeated in S. From these subtrees, we only consider the ones with at least k terminal nodes. Note that if k is large enough, we may have only a few such subtrees. For each subtree with $k' \geq k$ terminal nodes, representing suffixes $S[i_1..n-1], S[i_2..n-1], \ldots, S[i_{k'}..n-1]$, we create a batch of LCA queries between all pairs $(i_j + \ell_1 + d, i_{j'} + \ell_1 + d)$ and report occurrences when LCA queries extend pairwise matches to a length of at least ℓ_2 for a set of at least k such suffixes (this algorithm can be easily generalised for any number of gaps).

Pattern matching on weighted sequences. A *weighted sequence* specifies, for every position, the probability of each letter of the alphabet occurring. A weighted sequence thus represents many different strings, each with the probability of occurrence equal to the product of probabilities of its letters at subsequent positions of the weighted sequence. The problem is to find all occurrences of a (standard) pattern P of length m with probability at least $1/z$ in a weighted sequence S of length n [65]. The first step is to construct the heavy string of S, denoted by $H(S)$, by assigning to $H(S)[i]$ the most probable letter of $S[i]$ (resolving ties arbitrarily). The second step is to build $T(P\$H(S))$, $\$ \notin \Sigma$. We can then compute the first mismatch between P and every substring of $H(S)$. Note that the number of positions in S where two or more letters occur with probability at least $1/z$ can be small, and so we consider only those positions that cause a legitimate mismatch between P and a factor of $H(S)$. We then use $\mathcal{O}(\log z)$ batches of LCA queries per that starting position to extend a match to a

length of at least m. This is because P cannot match a weighted sequence S with probability $1/z$ if more than $\lfloor \log z \rfloor$ mismatches occur between P and $H(S)$ [65].

Pattern matching with don't care letters. We are given a pattern P of length m, with $m-k$ letters from alphabet Σ and k occurrences of a *don't care letter* (i.e. a letter matching itself and any letter from Σ), and a text S of length n. The problem is to find all occurrences of P in S [83]. The first step is to build $T(P'\$S)$, $\$ \notin \Sigma$, where P' is the string obtained from P by replacing don't care letters with a letter $\# \notin \Sigma$. We then locate the subtree rooted at the highest explicit node corresponding to the longest factor f of P' without #'s. We also locate, in the same subtree, all V terminal nodes corresponding to starting positions of f in S. Note that if f is long enough, we may have only a few such nodes. Since we know where the don't care letters occur in P, we can create a batch of kV LCA queries. An occurrence is then reported when LCA queries extend a match to a length of at least m. (This algorithm can be easily generalised for any number of patterns).

Circular string matching. We are given a pattern P of length m and a text S of length n. The problem is to find all occurrences of P, or any of its cyclic shifts in S [11]. The first step is to build $T(PP\$P^RP^R\#S\%S^R)$, where $\$, \#, \% \notin \Sigma$, and X^R denotes the reverse image of string X. We then conceptually split P in two fragments of lengths $\lceil m/2 \rceil$ and $\lfloor m/2 \rfloor$. Any cyclic shift of P contains as a factor at least one of these two fragments. We thus locate the two subtrees rooted at the highest explicit nodes corresponding to the fragments. We also locate in the same subtrees all V terminal nodes corresponding to starting positions of the fragments in S. Note that if m is long enough, we may have only a few such nodes. We create

a batch of at most $2V$ LCA queries in order to extend to the left and to the right and report occurrences when LCA queries extend a match to a length of at least m. (This algorithm can be easily generalised for any number of patterns).

6.5 Implementation

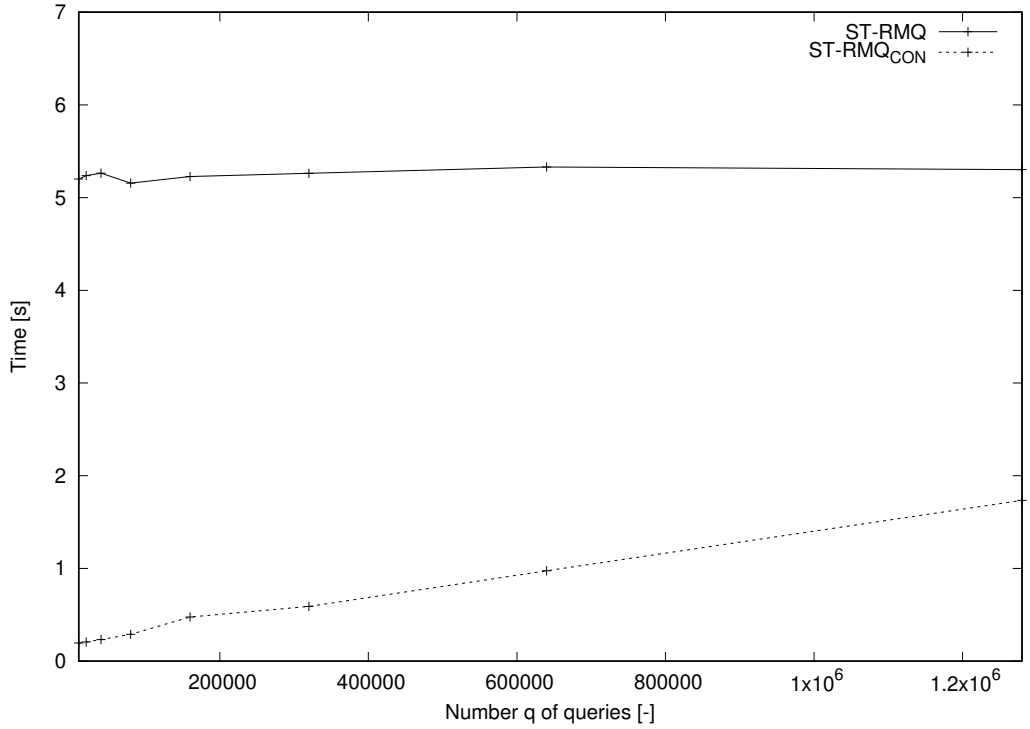
We have implemented algorithms ST-RMQ$_{\mathsf{CON}}$, OFF-RMQ$_{\mathsf{CON}}$ and ON-RMQ$_{\mathsf{CON}}$ in the C++ programming language. We have also implemented the same algorithms applied on the original array A, denoted by ST-RMQ, OFF-RMQ and ON-RMQ, respectively; as well as the brute-force algorithm for answering RMQs in the two corresponding flavours, denoted by BF-RMQ$_{\mathsf{CON}}$ and BF-RMQ. For the implementation of ON-RMQ$_{\mathsf{CON}}$ and ON-RMQ, we used the sdsl-lite library [44]. If an algorithm requires $f(n,q)$ time and $g(n,q)$ extra space, we say that the algorithm has complexity $< f(n,q), g(n,q) >$. Table 6.12 summarises the implemented algorithms. The following experiments were conducted on a Desktop PC using one core of an Intel Core i5-4690 CPU at 3.50GHz and 16GB of RAM. All programs were compiled with g++ version 5.4.0 at optimisation level 3 (-O3).

Experiment I. We generated random (uniform distribution) input arrays of $n = 1,000,000$ and $n = 100,000,000$ entries (integers), and random (uniform distribution) lists of queries of sizes varying from $\sqrt{n}$ to $128\sqrt{n}$, doubling each time. We compared the runtime of the implementations of the algorithms in Table 6.12 on these inputs. In particular, for each algorithm, we compared the standard implementation against the one with the contracted

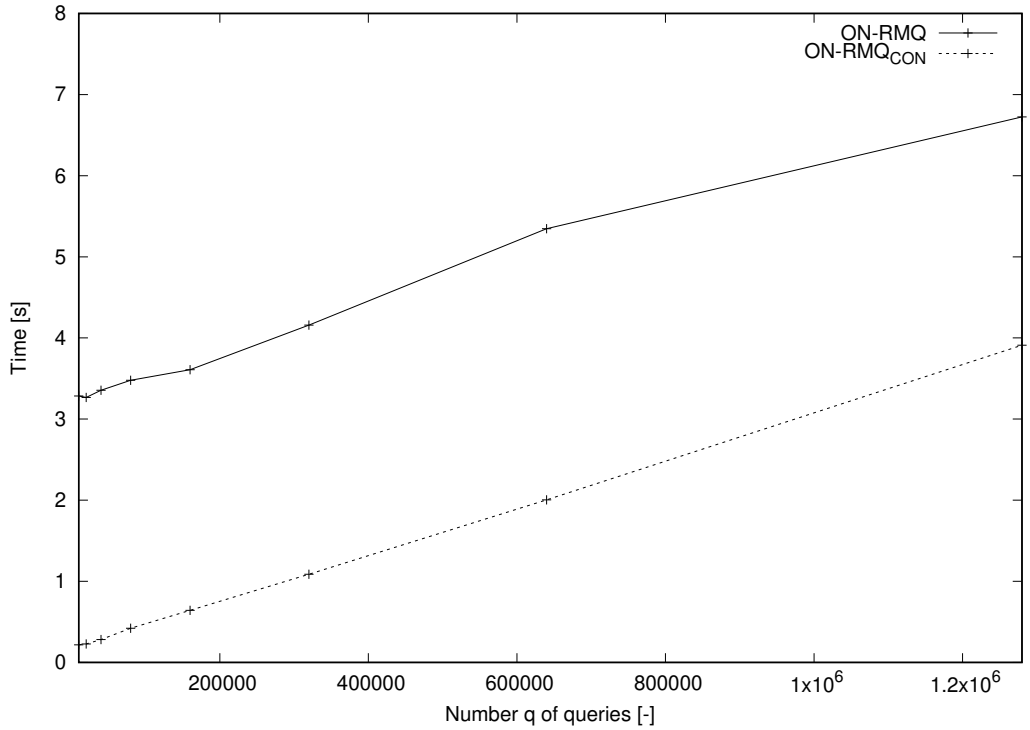
Non-Contracted		**Contracted**	
ST-RMQ	$< \mathscr{O}(n\log n+q), \mathscr{O}(n\log n) >$	ST-RMQ$_{\mathsf{CON}}$	$< n+\mathscr{O}(q\log q), \mathscr{O}(q) >$
ON-RMQ	$< \mathscr{O}(n+q), \mathscr{O}(n) >$	ON-RMQ$_{\mathsf{CON}}$	$< n+\mathscr{O}(q), \mathscr{O}(q) >$
OFF-RMQ	$< \mathscr{O}(n+q), \mathscr{O}(n) >$	OFF-RMQ$_{\mathsf{CON}}$	$< n+\mathscr{O}(q), \mathscr{O}(q) >$
BF-RMQ	$< \mathscr{O}(qn), \mathscr{O}(1) >$	BF-RMQ$_{\mathsf{CON}}$	$< n+\mathscr{O}(q^2), \mathscr{O}(q) >$

Table 6.12 Time and space complexities of algorithms for answering RMQs offline.

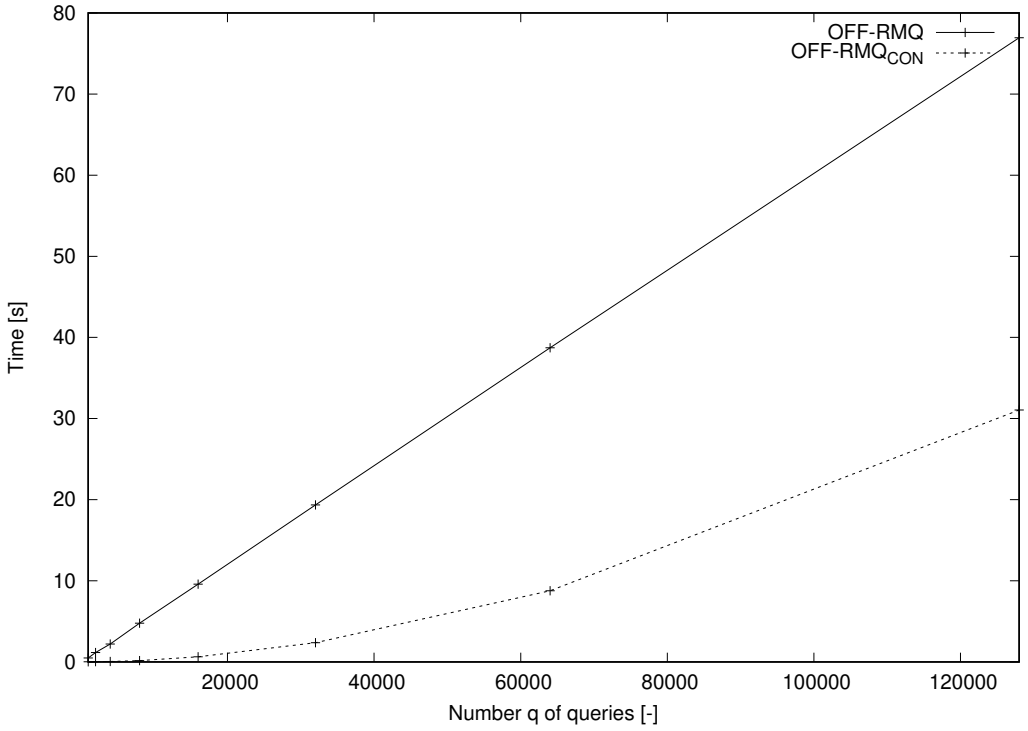
array. We used the large array, $n = 100,000,000$, for ST-RMQ and ON-RMQ because they are significantly faster, and the small one, $n = 1,000,000$, for OFF-RMQ and BF-RMQ. The results plotted in Figure 6.5 show that the proposed scheme of contracting the input array substantially improves the performance for all implementations.



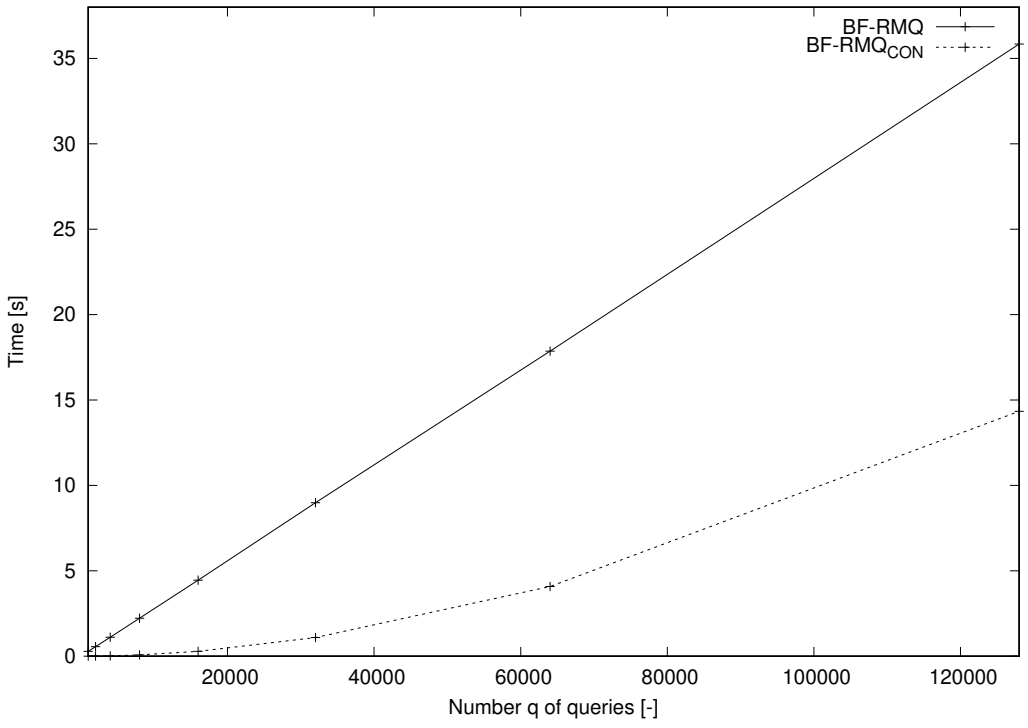
(a) $n = 100,000,000$



(b) $n = 100,000,000$



(c) $n = 1,000,000$

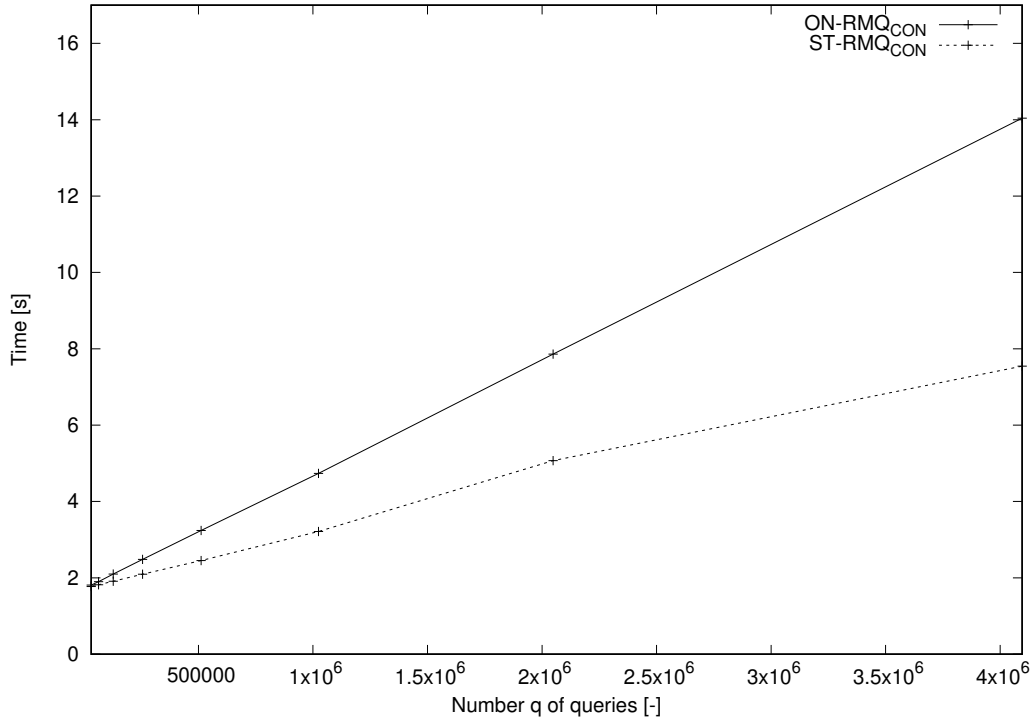


(d) $n = 1,000,000$

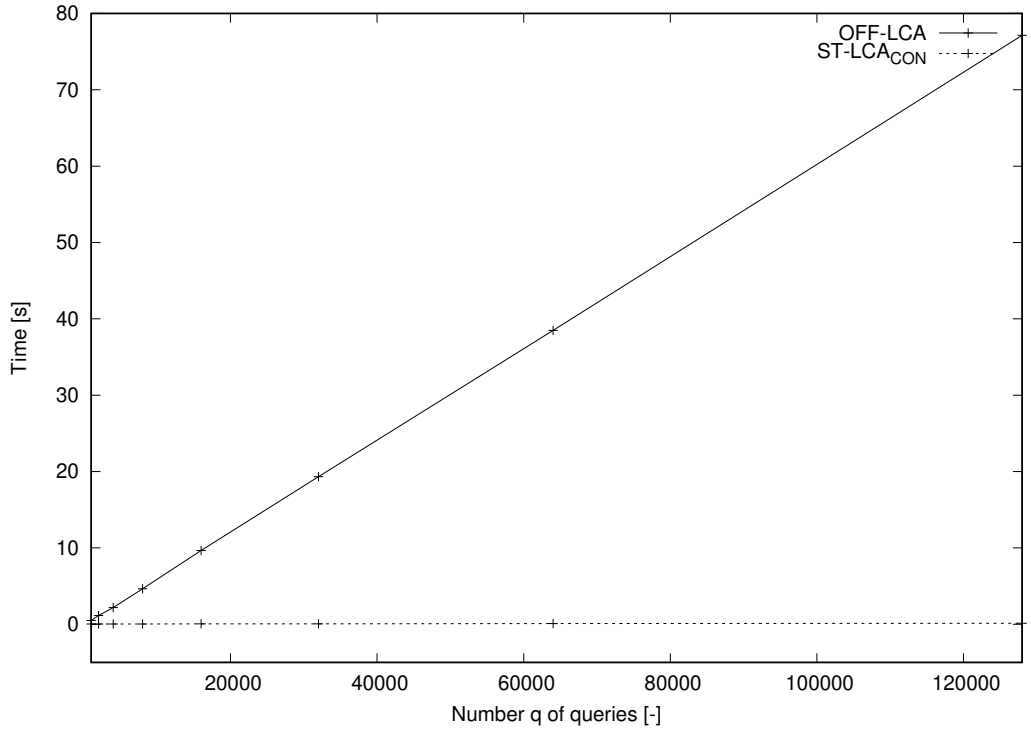
Fig. 6.5 Impact of the proposed scheme on the RMQ algorithms of Table 6.12.

Experiment II. We generated random input arrays of $n = 1,000,000,000$ entries, and random lists of queries of sizes varying from $\sqrt{n}$ to $128\sqrt{n}$, doubling each time. We then compared the runtime of ON-RMQ$_{\mathsf{CON}}$ and ST-RMQ$_{\mathsf{CON}}$ on these inputs. The results are plotted in Figure 6.6a. We observe that ST-RMQ$_{\mathsf{CON}}$ becomes two times faster than ON-RMQ$_{\mathsf{CON}}$ as q grows. Notably, it was not possible to run this experiment with ON-RMQ, which implements a *succinct* data structure for answering RMQs, due to insufficient amount of main memory.

Experiment III. In addition, we have implemented algorithms ST-LCA$_{\mathsf{CON}}$ and OFF-LCA for answering LCA queries. We first generated a random input array of $n = 1,000,000$ entries and used this array to compute its Cartesian tree. Next we generated random lists of LCA queries of sizes varying from $\sqrt{n}$ to $128\sqrt{n}$, doubling each time. We then compared the runtime of OFF-LCA and ST-LCA$_{\mathsf{CON}}$ on these inputs. The results plotted in Figure 6.6b show that the implementation of ST-LCA$_{\mathsf{CON}}$ is more than two orders of magnitude faster than the implementation of OFF-LCA, highlighting the impact of the proposed scheme on LCA queries.



(a) ON-RMQ$_{CON}$ vs ST-RMQ$_{CON}$, $n = 1,000,000,000$



(b) OFF-LCA vs ST-LCA$_{CON}$, $n = 1,000,000$

Fig. 6.6 Elapsed-time of ON-RMQ$_{CON}$ vs ST-RMQ$_{CON}$ and of OFF-LCA vs ST-LCA$_{CON}$.

6.6 Conclusion

In this chapter, we presented a new family of algorithms for answering a small batch of RMQs or LCA queries in practice. The main purpose was to show that if the number, q, of queries is small with respect to n, and we have them all at hand, existing algorithms for RMQs and LCA queries can be easily modified to perform in $n + \mathcal{O}(q)$ time and $\mathcal{O}(q)$ extra space. The presented experimental results indeed show that with this new scheme significant practical improvements can be obtained; in particular, for answering a small batch of LCA queries.

Specifically, algorithms ST-RMQ$_{\mathsf{CON}}$ and ST-LCA$_{\mathsf{CON}}$, our modifications to the *Sparse Table* algorithm whose main catch is $\Theta(n \log n)$ space [15], seem to be the best way to answer in practice a small batch of RMQs and LCA queries, respectively. A library implementation of ST-RMQ$_{\mathsf{CON}}$ is available at *https://github.com/maialzamel/rmqo* under the GNU General Public License.

Recently, Kowaliski and Grabowski in [66] have made further improvements in the computation of the "small batch" RMQs problems. The authors introduced a hybrid approach, which is a slight improvement in practice. Their algorithm does some prepossessing and it creates a data structure, then when it receives the "batch of queries", it augments the data structure, in order to answer them.

Chapter 7

Conclusion and Future Work

This thesis focuses on a number of computational problems, which are linked to applications that analyse molecular sequences and are motivated by real life problems. The aim of this thesis is to design powerful, fast and practical string algorithms, as these are vital to bio-informatics and have applications in other areas like security, privacy and music analysis. Herein, we present a brief summary of the presented work in this thesis and discusses current problems and future work.

In Chapter 3, our main contribution is the design of an $\mathcal{O}(n)$ space and time algorithm for integer alphabets of size σ, if $m = \Omega(\log_\sigma n)$ is average to solve the k-mappability problem. Experimental results are provided for our algorithm in agreement with Theorem 1. We also provide comparative results for our algorithms relative to state-of-the-art implementations [26]. The experiments show that our algorithm requires predicted linear time in n up to a certain value of n, according to Theorem 1. However, our implementation becomes considerably faster with increasing values of m and fixed n (see Theorem

1). Additionally, we demonstrated that the memory usage of our implementation grows linearly with n (see Theorem 1). The k-mappability algorithm is an average-case algorithm with a specific m and n. Additionally, we provided, in Appendix A, an algorithm requiring $\mathcal{O}(\min\{nm, n\log n \log\log n\})$ of time and $\mathcal{O}(n)$ space for the worst case. The same authors in [6] presented an $\mathcal{O}(\min\{nm^k, n\log^{k+1} n\})$ time and $\mathcal{O}(n)$ space, $k = \mathcal{O}(1)$ and constant-sized alphabets to solve the k-mappability problem. Furthermore, our future work is to apply the technique of Thankachan et al. [93] to obtain $\mathcal{O}(\min\{nm, n\log n\})$ time and $\mathcal{O}(n)$ space for when $k = 1$ (for a preliminary exposition of the ideas, see [52]). Additionally, in [86] we present **GenMap**, a more practical algorithm for computing the mappability of genomes up to k errors, which is based on the C++ sequence analysis library SeqAn library [88]. This is significantly faster, often by a magnitude, than the algorithm from the widely used *GEM suite* in [26], while also refraining from approximations. An open question for the k-mappability problem concerns whether k-mappability can be solved over an $o(n\log n)$-time. Another important question to investigate is the k-mappability problem under edit distance. In this approach, all possible factors of length of exactly m, $m-k$ and $m+k$ should be counted.

Chapter 4 sets out a linear algorithm for string comparison to determine whether two GD strings have a non-empty intersection. Furthermore, a string comparison tool has been applied to devise a simple algorithm to compute all palindromes in a GD string. In Section 4.3.1 we sketched how automata can be used to compare two ED strings. Recall that an ED string is a more general conceptualization of a degenerate string, where a degenerate letter generally contains strings of different lengths, as well as an empty string. For GD strings, we demonstrated that a comparison can be done in linear time (Theorem 3). An interesting

open problem relates to whether we can devise a more efficient (than the $\mathcal{O}(NM)$-time automata-based) approach to establish whether the two languages represented by two ED strings of sizes N and M have a non-empty intersection; or, whether more generally, they share a sufficiently long substring.

In Chapter 5, a special type of string, called a closed string was studied. String x is said to be closed if it has a nonempty proper prefix that is also a suffix, and which otherwise occurs nowhere else in x. In this chapter, there is an $\mathcal{O}(kn)$-time and $\mathcal{O}(n)$ space algorithm to decide whether an input string of length n over the integer alphabet describes a closed string of up to k mismatches. The pseudocode for implementation and proof-of-concept experimental results has also been demonstrated. Another approach to investigate is finding an efficient algorithm under the Edit distance model for a k-closed string.

In Chapter 6, a new family of algorithms for answering a small batch of RMQs or LCA queries has been given. The main goal of this study was to find $n+\mathcal{O}(q)$ time and $\mathcal{O}(q)$ extra space algorithm to answer RMQs and LCA queries, when the number of q of queries is small with respect to n, and we have them all available. The efficiency of the algorithms presented has been demonstrated extensively in the experimental results. Especially, to answer a small batch of LCA queries. Later, the "small batch" RMQs problems were improved further by Kowaliski and Grabowski in [66]. The authors introduced a hybrid approach, which represented a slight improvement in practice. Their algorithm performs some prepossessing and creates a data structure, and when it receives the "batch of queries", it augments the data structure to answer them.

References

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

[2] M. Adamczyk, M. Alzamel, P. Charalampopoulos, C. S. Iliopoulos, and J. Radoszewski. Palindromic decompositions with gaps and errors. In *CSR*, volume 10304 of *LNCS*, pages 48–61. Springer International Publishing, 2017.

[3] M. Adamczyk, M. Alzamel, P. Charalampopoulos, and J. Radoszewski. Palindromic decompositions with gaps and errors. *Int. J. Found. Comput. Sci.*, 29(8):1311–1329, 2018.

[4] P. Afshani and N. Sitchinava. I/O-efficient range minima queries. In *SWAT 2014*, volume 8503 of *LNCS*, pages 1–12. Springer, 2014.

[5] Y. Almirantis, P. Charalampopoulos, J. Gao, C. S. Iliopoulos, M. Mohamed, S. P. Pissis, and D. Polychronopoulos. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology*, 12(1):5, 2017.

[6] M. Alzamel, P. Charalampopoulos, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, J. Radoszewski, and J. Straszynski. Efficient computation of sequence mappability. In T. Gagie, A. Moffat, G. Navarro, and E. Cuadros-Vargas, editors, *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, volume 11147 of *Lecture Notes in Computer Science*, pages 12–26. Springer, 2018.

[7] M. Alzamel, P. Charalampopoulos, C. S. Iliopoulos, and S. P. Pissis. How to answer a small batch of rmqs or lca queries in practice. In L. Brankovic, J. Ryan, and W. F. Smyth, editors, *Combinatorial Algorithms*, volume 10765 of *Lecture Notes in Computer Science*, pages 343–355, Cham, 2018. Springer International Publishing.

[8] M. Alzamel, J. Gao, C. S. Iliopoulos, and C. Liu. Efficient computation of palindromes in sequences with uncertainties. *Fundam. Inform.*, 163(3):253–266, 2018.

[9] P. Antoniou, J. W. Daykin, C. S. Iliopoulos, D. Kourie, L. Mouchard, and S. P. Pissis. Mapping uniquely occurring short sequences derived from high throughput technologies to a reference genome. In *2009 9th International Conference on Information Technology and Applications in Biomedicine*, pages 1–4. IEEE Computer Society, 2009.

[10] L. Arge, J. Fischer, P. Sanders, and N. Sitchinava. On (dynamic) range minimum queries in external memory. In *WADS 2013*, volume 8037 of *LNCS*, pages 37–48. Springer, 2013.

[11] T. Athar, C. Barton, W. Bland, J. Gao, C. S. Iliopoulos, C. Liu, and S. P. Pissis. Fast circular dictionary-matching algorithm. *Mathematical Structures in Computer Science*, 27(2):143–156, 2017.

[12] G. Badkobeh, H. Bannai, K. Goto, T. I, C. S. Iliopoulos, S. Inenaga, S. J. Puglisi, and S. Sugimoto. Closed factorization. *Discrete Applied Mathematics*, 212:23–29, 2016. Stringology Algorithms.

[13] G. Badkobeh, G. Fici, and Z. Lipták. On the number of closed factors in a word. In A.-H. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications*, pages 381–390, Cham, 2015. Springer International Publishing.

[14] C. Barton, C. S. Iliopoulos, S. P. Pissis, and W. F. Smyth. Fast and simple computations using prefix tables under hamming and edit distance. In K. Jan, M. Miller, and D. Froncek, editors, *Combinatorial Algorithms*, pages 49–61, Cham, 2015. Springer International Publishing.

[15] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In G. H. Gonnet, D. Panario, and A. Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[16] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN*, volume 1776 of *LNCS*, pages 88–94. Springer, 2000.

[17] 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.

[18] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[19] G. Bernardini, N. Pisanti, S. P. Pissis, and G. Rosone. Pattern matching on elastic-degenerate text with errors. In *SPIRE*, volume 10508 of *LNCS*, pages 74–90. Springer, 2017.

[20] S. Brenner, F. Jacob, and M. Meselson. An unstable intermediate carrying information from genes to ribosomes for protein synthesis. *Nature*, 190(4776):576–581, 1961.

[21] M. Bucci, A. de Luca, and A. De Luca. *Rich and Periodic-Like Words*, pages 145–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[22] T. C. P. Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018.

[23] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

[24] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Covering problems for partial words and for indeterminate strings. *Theoretical Computer Science*, 698:25–39, 2017.

[25] M. Crochemore and G. Tischler. The gapped suffix array: A new index structure for fast approximate matching. In E. Chávez and S. Lonardi, editors, *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2010.

[26] T. Derrien, J. Estellé, S. Marco Sola, D. Knowles, E. Raineri, R. Guigó, and P. Ribeca. Fast computation and applications of genome mappability. *PLoS ONE*, 7(1), 2012.

[27] M. Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97*, pages 137–143. IEEE Computer Society, 1997.

[28] M. Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143. IEEE, 1997.

[29] H. Ferrada and G. Navarro. Improved range minimum queries. *J. Discrete Algorithms*, 43:72–80, 2016.

[30] G. Fici. A classification of trapezoidal words. In Proceedings 8th International Conference *Words 2011,* Prague, volume 63 of *Electronic Proceedings in Theoretical Computer Science*, pages 129–137, 2011.

[31] J. Fischer. Inducing the LCP-array. In F. Dehne, J. Iacono, and J. Sack, editors, *Algorithms and Data Structures - 12th International Symposium, WADS 2011. Proceedings*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer, 2011.

[32] J. Fischer and V. Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *CPM 2006*, volume 4009 of *LNCS*, pages 36–48. Springer Berlin Heidelberg, 2006.

[33] J. Fischer, D. Köppl, and F. Kurpicz. On the benefit of merging suffix array intervals for parallel pattern matching. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, pages 26:1–26:11, 2016.

[34] N. A. Fonseca, J. Rung, A. Brazma, and J. C. Marioni. Tools for mapping high-throughput sequencing data. *Bioinformatics*, 28(24):3169–3177, 2012.

[35] 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.

[36] M. C. Frith, U. Hansen, J. L. Spouge, and Z. Weng. Finding functional sequence elements by multiple local alignment. *Nucleic Acids Res.*, 32(1):189–200, 2004.

[37] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *STOC 1984*, pages 135–143. ACM, 1984.

[38] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[39] J. A. Gally and G. M. Edelman. The genetic control of immunoglobulin synthesis. *Annual Review of Genetics*, 6(1):1–46, 1972.

[40] J. Gao and R. Impagliazzo. Orthogonal vectors is hard for first-order properties on sparse graphs. *Electronic Colloquium on Computational Complexity (ECCC)*, 23:53, 2016.

[41] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[42] E. A. Gladyshev and I. R. Arkhipova. Rotifer rdna-specific r9 retrotransposable elements generate an exceptionally long target site duplication upon insertion. *Gene*, 448(2):145 – 150, 2009. Genomic Impact of Eukaryotic Transposable Elements.

[43] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[44] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, volume 8504 of *LNCS*, pages 326–337, 2014.

[45] G. Gourdel, T. Kociumaka, J. Radoszewski, and T. Starikovskaya. Approximating Longest Common Substring with k mismatches: Theory and Practice. In I. L. Gørtz and O. Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[46] F. Gros, H. Hiatt, W. Gilbert, C. G. Kurland, R. Risebrough, and J. D. Watson. Unstable ribonucleic acid revealed by pulse labelling of escherichia coli. *Nature*, 190(4776):581, 1961.

[47] R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani, and L. Versari. On-line pattern matching on a set of similar texts. In *CPM*, LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[48] D. Gusfield. *Algorithms on strings, trees, and sequences*. Cambridge University Press New York, New York, 1997.

[49] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[50] R. W. Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[51] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[52] S. Hooshmand, P. Abedin, D. Gibney, S. Aluru, and S. V. Thankachan. Faster computation of genome mappability. In A. Shehu, C. H. Wu, C. Boucher, J. Li, H. Liu, and M. Pop, editors, *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB 2018, Washington, DC, USA, August 29 - September 01, 2018*, page 537. ACM, 2018.

[53] J. E. Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[54] L. Ilie, G. Navarro, and L. Tinta. The longest common extension problem revisited and applications to approximate string searching. *Journal of Discrete Algorithms*, 8(4):418–428, 2010.

[55] L. Ilie, G. Navarro, and L. Tinta. The longest common extension problem revisited and applications to approximate string searching. *J. Discrete Algorithms*, 8(4):418–428, 2010.

[56] C. Iliopoulos, J. Mchugh, P. Peterlongo, N. Pisanti, W. Rytter, and M.-F. Sagot. A first approach to finding common motifs with gaps. *International Journal of Foundations of Computer Science*, 16(6):1145–1155, 2005.

[57] C. S. Iliopoulos, R. Kundu, and S. P. Pissis. Efficient pattern matching in elastic-degenerate texts. In *LATA*, volume 10168 of *LNCS*, pages 131–142. Springer International Publishing, 2017.

[58] C. S. Iliopoulos and J. Radoszewski. Truly Subquadratic-Time Extension Queries and Periodicity Detection in Strings with Uncertainties. In *CPM*, volume 54 of *LIPIcs*, pages 8:1–8:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[59] R. Impagliazzo and R. Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001.

[60] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

[61] IUPAC-IUB Commission on Biochemical Nomenclature. Abbreviations and symbols for nucleic acids, polynucleotides, and their constituents. *Biochemistry*, 9(20):4022–4027, 1970.

[62] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages, and Programming*, pages 943–955. Springer, 2003.

[63] D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2-4):126–142, 2005.

[64] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[65] T. Kociumaka, S. P. Pissis, and J. Radoszewski. Pattern Matching and Consensus Problems on Weighted Sequences and Profiles. In *ISAAC 2016*, volume 64 of *LIPIcs*, pages 46:1–46:12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[66] T. M. Kowalski and S. Grabowski. Faster range minimum queries. *Software: Practice and Experience*, 48(11):2043–2060, 2018.

[67] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[68] R. S. Linheiro and C. M. Bergman. Whole genome resequencing reveals natural target site preferences of transposable elements in drosophila melanogaster. *PLOS ONE*, 7(2):1–12, 02 2012.

[69] R. J. Lipton. *On The Intersection of Finite Automata*, pages 145–148. Springer US, Boston, MA, 2010.

[70] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

[71] G. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22(3):346–351, 1975.

[72] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[73] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[74] G. Manzini. Longest common prefix with mismatches. In C. S. Iliopoulos, S. J. Puglisi, and E. Yilmaz, editors, *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2015.

[75] J. C. Martin. *Introduction to Languages and the Theory of Computation*, volume 4. McGraw-Hill NY, 1991.

[76] L. A. McCue, W. Thompson, S. Carmack, M. P. Ryan, J. S. Liu, V. Derbyshire, and C. E. Lawrence. Phylogenetic footprinting of transcription factor binding sites in proteobacterial genomes. *Nucleic Acids Res.*, 29(3):774–782, 2001.

[77] M. L. Metzker. Sequencing technologies – the next generation. *Nat. Rev. Genet.*, 11(1):31–46, 2010.

[78] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[79] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[80] B. M. Muhire, M. Golden, B. Murrell, P. Lefeuvre, J.-M. Lett, A. Gray, A. Y. Poon, N. K. Ngandu, Y. Semegni, E. P. Tanov, et al. Evidence of pervasive biologically functional secondary structures within the genomes of eukaryotic single-stranded DNA viruses. *Journal of virology*, 88(4):1972–1989, 2014.

[81] E. W. Myers. Approximate matching of network expressions with spacers. *Journal of Computational Biology*, 3(1):33–51, 1996.

[82] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In J. A. Storer and M. W. Marcellin, editors, *2009 Data Compression Conference (DCC 2009)*, pages 193–202. IEEE Computer Society, 2009.

[83] R. Y. Pinter. Efficient string matching with don't-care patterns. In *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 11–29. Springer Berlin Heidelberg, 1985.

[84] N. Pisanti, H. Soldano, M. Carpentier, and J. Pothier. A relational extension of the notion of motifs: Application to the common 3d protein substructures searching problem. *Journal of Computational Biology*, 16(12):1635–1660, 2009.

[85] S. P. Pissis. MoTeX-II: structured motif extraction from large-scale datasets. *BMC Bioinformatics*, 15:235, 2014.

[86] C. Pockrandt, M. Alzamel, C. S. Iliopoulos, and K. Reinert. Genmap: Fast and exact computation of genome mappability. *bioRxiv*, page 611160, 2019.

[87] M. Régnier and P. Jacquet. New results on the size of tries. *IEEE Trans. Information Theory*, 35(1):203–205, 1989.

[88] K. Reinert, T. H. Dadi, M. Ehrhardt, H. Hauswedell, S. Mehringer, R. Rahn, J. Kim, C. Pockrandt, J. Winkler, E. Siragusa, et al. The seqan c++ template library for efficient sequence analysis: A resource for programmers. *Journal of biotechnology*, 261:157–168, 2017.

[89] R. T. Schuh. Major patterns in vertebrate evolution. *Systematic Biology*, 27(2):172, 1978.

[90] H. Soldano, A. Viari, and M. Champesme. Searching for flexible repeated patterns using a non-transitive similarity relation. *Pattern Recognition Letters*, 16(3):233–246, 1995.

[91] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[92] R. E. Tarjan and U. Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *25th Annual Symposium onFoundations of Computer Science, 1984.*, pages 12–20. IEEE, 1984.

[93] S. V. Thankachan, A. Apostolico, and S. Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016.

[94] P. Vesely. Molecular biology of the cell. *Scanning: The Journal of Scanning Microscopies*, 26(3):153–153, 2004.

[95] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[96] J. D. Watson, F. H. Crick, et al. Molecular structure of nucleic acids. *Nature*, 171(4356):737–738, 1953.

[97] P. Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

[98] R. Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci*, 348(2-3):357–365, 2005.

[99] C. Wuilmart, J. Urbain, and D. Givol. On the location of palindromes in immunoglobulin genes. *Proceedings of the National Academy of Sciences of the United States of America*, 74(6):2526–2530, 1977.

Appendix A

Efficient Worst-Case Algorithms of k-mappability

A.1 Efficient Worst-Case Algorithms

A.1.1 $\mathscr{O}(mn)$-time and $\mathscr{O}(n)$-space algorithm

In this section we assume that x is a string over an integer alphabet Σ. The main idea is that we want to first find all pairs $x[i_1..i_1+m-1] \approx_1 x[i_2..i_2+m-1]$ that have a mismatch in the first position, then in the second, and so on.

Let us fix $0 \leq j < m$. In order to identify the pairs $x[i_1..i_1+m-1] \approx_1 x[i_2..i_2+m-1]$ with $x[i_1+j] \neq x[i_2+j]$ (i.e. with the mismatch in the j^{th} position), we do the following. For every $i = 0, 1, \ldots, n-m$, we find the explicit or implicit node $u_{i,j}$ in $\mathscr{T}(x)$ that represents $x[i..i+j-1]$ and the node $v_{i,j}$ in $\mathscr{T}(\mathsf{rev}(x))$ that represents $\mathsf{rev}(x[i+j+1..i+m-1]) =$

$\mathsf{rev}(x)[n-i-m\,..\,n-i-j-2]$. In each such node $v_{i,j}$, we create a set $V(v_{i,j})$—if it has not already been created—and insert the triple $(u_{i,j}, x[i+j], i)$.

When we have done this for all possible starting positions of x, we group the triples in each set $V(v)$ by the node variable (i.e., the first component in the triples). For each such group in $V(v)$ we count the number of triples that have each letter of the alphabet and increment array C accordingly. More precisely, if $V(v)$ contains q triples that correspond to the same node u, among which r correspond to the letter $c \in \Sigma$, then for each such triple $(u,c,i) \in V(v)$ we increment $C[i]$ by $q-r$; we subtract r to avoid counting equal factors in C. Before we proceed with the computations for the next index j, we delete all the sets $V(v)$. We formalize this algorithm, denoted by 1-MAP, in the pseudocode presented below and provide an example.

Algorithm 5 $1-Map(x,n,m)$

```
𝒯(x) ← SuffixTree(x)
𝒯(x) ← SuffixTree(rev(x))
for string-depth j = 0 to m − 1 do
    for i = 0 to n − m do
        u_{i,j} ← (Node_{𝒯(x)}(x[i..i+j−1])
        v_{i,j} ← (Node_{𝒯(rev(x))}rev(x)[n−i−m..n−i−j−2])
        Insert(u_{i,j}, x[i+j], i) to V(v_{i,j})
    end for
    for every node v of string-depth m − j − 2 in 𝒯(rev(x)) do
        Group triples in V(v) by the node variable
        for a group corresponding to the node u in V(v) do
            Count number of triples with each letter c ∈ Σ
            Update C[i] accordingly for each triple (u,c,i)
        end for
        Delete V(v)
    end for
end for
```

Example 30. Suppose we have $V(v) = \{(u,\mathtt{A},i_1),(u,\mathtt{A},i_2),(u,\mathtt{A},i_3),(u,\mathtt{C},i_4),(u,\mathtt{C},i_5),(u,\mathtt{C},i_6),(u,\mathtt{G},i_7),(u,\mathtt{G},i_8),(u,\mathtt{T},i_9)\}$, for some distinct positions $i_1,i_2,\ldots,i_9$. We then increment $C[i_1]$, $C[i_2]$, $C[i_3]$, $C[i_4]$, $C[i_5]$, and $C[i_6]$ by 6; $C[i_7]$ and $C[i_8]$ by 7; and $C[i_9]$ by 8.

We now analyze the time complexity of this algorithm. The algorithm iterates j from 0 to $m-1$. In the j^{th} iteration, we need to compute $\{u_{i,j},v_{i,j} \mid i = 0,\ldots,n-m\}$. When $j=0$, $u_{i,0}$ for every i is the root of $\mathscr{T}(x)$ and we can find $v_{i,0}$ for all i naïvely in $\mathscr{O}(mn)$ time. For $j>0$, $v_{i,j}$ can be found in $\mathscr{O}(1)$ time from $v_{i,j-1}$ by moving one letter up in $\mathscr{T}(\mathtt{rev}(x))$ for all i, while $u_{i,j}$ can be obtained from $u_{i,j-1}$ by going down in $\mathscr{T}(x)$ based on letter $x[i+j]$. We then include $(u_{i,j},x[i+j],i)$ in $V(v_{i,j})$.

This requires in total $\mathscr{O}(mn)$ randomized time due to perfect hashing [35] which allows to go down from a node in $\mathscr{T}(x)$ (or in $\mathscr{T}(\mathsf{rev}(x))$) based on a letter in $\mathscr{O}(1)$ randomized time. We can actually avoid this randomization, as queries for a particular child of a node are asked in our solution in a somewhat off-line fashion: we use them only to compute $v_{i,0}$ (m times) and $u_{i,j}$ (from $u_{i,j-1}$).

Observation 1. For an integer alphabet $\Sigma = \{1,\ldots,n\}$, one can answer off-line $\mathscr{O}(n)$ queries in $\mathscr{T}(x)$ asking for a child of an explicit or implicit node u labelled with the letter $c \in \Sigma$ in (deterministic) $\mathscr{O}(n)$ time.

Proof. A query for an implicit node u is answered in $\mathscr{O}(1)$ time, as there is only one outgoing edge to check. All the remaining queries can be sorted lexicographically as pairs (u,c) using radix sort. We can also assume that the children of every explicit node of $\mathscr{T}(x)$ are ordered

by the letter (otherwise we also radix sort them). Finally, all the queries related to a node u can be answered in one go by iterating through the children list of u once. □

Lastly, we use bucket sort to group the triples for each $V(v)$ according to the node variable (recall that the nodes are represented by the edge and the index within the edge) and update the counters in $\mathcal{O}(n)$ time in total (using a global array indexed by the letters from Σ, which is zeroed in $\mathcal{O}(|V(v)|)$ time after each $V(v)$ has been processed). Overall the algorithm requires $\mathcal{O}(mn)$ time. The suffix trees require $\mathcal{O}(n)$ space and we delete the sets $V(v_{i,j})$ after the j^{th} iteration; the space complexity of the algorithm is thus $\mathcal{O}(n)$. We obtain the following result.

Theorem 4. Given a string of length n over an integer alphabet and an integer m, where $1 \leq m < n$, the 1-MAPPABILITY problem can be solved in $\mathcal{O}(mn)$ time and $\mathcal{O}(n)$ space.

Corollary 1 and Theorem 4 imply the following result.

Theorem 5. Given a string x of length n over an integer alphabet Σ of size $\sigma > 1$ with the letters of x being independent and identically distributed random variables, uniformly distributed over Σ, the 1-MAPPABILITY problem can be solved in average-case time $\mathcal{O}(n \log n)$ and space $\mathcal{O}(n)$.

Proof. If $m \geq 3 \cdot \log_\sigma n + 3$, apply Corollary 1. Otherwise, apply Theorem 4. □

Remark 1. Theorem 4 can also be obtained via utilising the gapped suffix array data structure (see [25] for an efficient construction algorithm).

A.1.2 $\mathcal{O}(n \log n \log\log n)$-time and $\mathcal{O}(n)$-space algorithm

In this section we assume that x is a length-n string over an ordered alphabet Σ, where $|\Sigma| = \sigma = \mathcal{O}(1)$. Consider two factors of x represented by nodes u and v in $\mathcal{T}(x)$; we observe that the first mismatch between the two factors is the first letter of the labels of the distinct outgoing edges from the lowest common ancestor of u and v that lie on the paths from the root to u and v. For 1-mappability we require that what follows this mismatch is an exact match.

Definition 14. Let T be a rooted tree. For each non-leaf node u of T, the *heavy edge* (u,v) is an edge for which the subtree rooted at v has the maximal number of leaves (in case of several such subtrees, we fix one of them). The *heavy path of a node v* is a maximal path of heavy edges that passes through v (it may contain 0 edges). The *heavy path of T* is the heavy path of the root of T.

Consider the suffix tree $\mathcal{T}(x)$ and its node u. We say that an (explicit or implicit) node v is a *level ancestor of u at string-depth ℓ* if $\mathcal{D}(v) = \ell$ and $\mathcal{L}(v)$ is a prefix of $\mathcal{L}(u)$. The heavy paths of $\mathcal{T}(x)$ can be used to compute level ancestors of nodes in $\mathcal{O}(\log n)$ time. However, a more efficient data structure is known.

Lemma 9. After $\mathcal{O}(n)$-time preprocessing on $\mathcal{T}(x)$, level ancestor queries of nodes of $\mathcal{T}(x)$ can be answered in $\mathcal{O}(\log\log n)$ time per query.

Definition 15. Given a string x and a factor y of x, we denote by $\mathit{range}(x,y)$ the range in the SA of x that represents the suffixes of x that have y as a prefix.

Every node u in $\mathcal{T}(x)$ corresponds to an SA range $I_u = \mathit{range}(x, \mathcal{L}(u)) = (u_{\min}, u_{\max})$. We can precompute I_u for all explicit nodes u in $\mathcal{T}(x)$ in $\mathcal{O}(n)$ time while performing a depth-

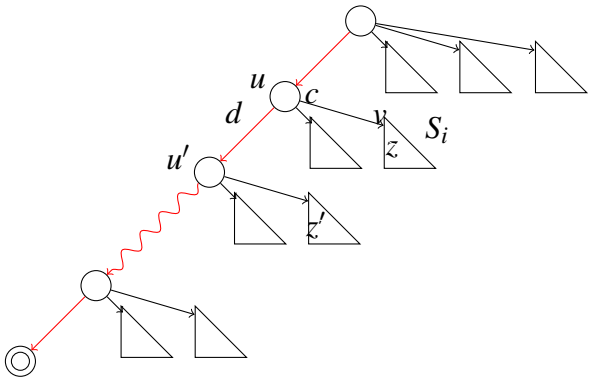


Fig. A.1 Illustration; the heavy path of $\mathscr{T}(x)$ is shown in red.

first traversal of the tree as follows. For a non-terminal node v with children $u^1,\ldots,u^q$, we set $v_{\min} = \min_i\{u^i_{\min}\}$ and $v_{\max} = \max_i\{u^i_{\max}\}$. If v is a terminal node (with children $u^1,\ldots,u^q$), representing the suffix $x[j\,..\,n-1]$, we set $v_{\min} = \mathsf{iSA}[j]$ and $v_{\max} = \max\{\mathsf{iSA}[j], \max_i\{u^i_{\max}\}\}$. When a considered node v is implicit, say along an edge (p,q), then $I_v = I_q$.

Our algorithm relies heavily on the following auxiliary lemmas.

Lemma 10. Consider a node u in $\mathscr{T}(x)$ with $p = \mathscr{L}(u)$. Let $\mathit{suf}(u,\ell)$ be the node v such that $\mathscr{L}(v) = p[\ell\,..\,|p|-1]$. Given the SA and the iSA of x, v can be computed in $\mathscr{O}(\log\log n)$ time after $\mathscr{O}(n)$-time preprocessing.

Proof. The SA range of the node u is $I_u = (u_{\min}, u_{\max})$; $u_{\min}$ corresponds to the suffix $x[\mathsf{SA}[u_{\min}]\,..\,n-1]$. By removing the first ℓ letters, the suffix becomes $x[\mathsf{SA}[u_{\min}]+\ell\,..\,n-1]$. The corresponding SA value is $v_{\min} = \mathsf{iSA}[\mathsf{SA}[u_{\min}]+\ell]$.

Let v_1 be the node of $\mathscr{T}(x)$ such that $\mathscr{L}(v_1) = x[\mathsf{SA}[v_{\min}]\,..\,n-1]$. The sought node v is the ancestor of v_1 located at string-depth $|p|-\ell$. It can be computed in $\mathscr{O}(\log\log n)$ time using the level ancestor data structure of Lemma 9.

□

Lemma 11. Let u and v be two nodes in $\mathscr{T}(x)$. We denote $\mathscr{L}(u)$ by p_1 and $\mathscr{L}(v)$ by p_2. We further denote by $\text{concat}(u,v)$ the node w such that $\mathscr{L}(w) = p_1p_2$. Given the SA and the iSA of x, as well as $range(x,p_1)$ and $range(x,p_2)$, w can be located in $\mathscr{O}(\log\log n)$ time after $\mathscr{O}(n\log\log n)$-time and $\mathscr{O}(n)$-space preprocessing.

Proof. We can compute $range(x,p_1p_2) = (w_{\min}, w_{\max})$ in $\mathscr{O}(\log\log n)$ time after $\mathscr{O}(n\log\log n)$-time and $\mathscr{O}(n)$-space preprocessing [33]; we can then locate w in $\mathscr{O}(\log\log n)$ time using the level ancestor data structure of Lemma 9. □

We are now ready to present an algorithm for 1-mappability that requires $\mathscr{O}(n\log n\log\log n)$ time and $\mathscr{O}(n)$ space. The first step is to build $\mathscr{T}(x)$. We then make every node u of string-depth m explicit in $\mathscr{T}(x)$ and initialize a counter $Count(u)$ for it. For each explicit node u in $\mathscr{T}(x)$, the SA range $I_u = range(x,\mathscr{L}(u))$ is also stored. We also identify the node v_c with path-label c for each $c \in \Sigma$ in $\mathscr{O}(\sigma) = \mathscr{O}(1)$ time.

Algorithm 6 $PerformCount(T,m))$

```
1:  HP ← HeavyPath(T)
2:  for each side-tree S_i attached to a node u on HP with D(u) < m do
3:      Let (u,v) be the edge that connects S_i to HP
4:      c ←the edge label of (u,v)
5:      d ←the edge label of the heavy edge (u,u')
6:      for each node z in S_i with D(z) = m do
7:          w ←suf(z, D(u)+1)
8:          for each c' ≠ c, label of an outgoing edge from u do
9:              t ← concat(u, concat(v_c', w))
10:             Count(z) ← Count(z) + |I_t|
11:         end for
12:         z' ← concat(u, concat(v_d, w))
13:         Count(z') ← Count(z') + |I_z|
14:     end for
15:     PerformCount(S_i, m − D(u))
16: end for
```

We then call PERFORMCOUNT($\mathscr{T}(x), m$), which does the following (inspect also the pseudocode above and Figure A.1). At first, a heavy path *HP* of $\mathscr{T}(x)$ is computed. Initially, we want to identify the pairs of factors of x of length m at Hamming distance 1 that have a mismatch in the labels of the edges outgoing from a node in *HP*. Given a node u in *HP*, with $\mathscr{L}(u) = p_1$, for every side tree S_i attached to it (say by an edge with label $c \in \Sigma$), we find all nodes of S_i with string-depth m. For every such node z, with path-label p_1cp_2, we use Lemma 10 to obtain the node $w = \text{suf}(z, |p_1| + 1)$; that is, $\mathscr{L}(w) = p_2$. We then use Lemma 11 to compute $\mathit{range}(x, p_1c'p_2)$ for all $c' \neq c$ such that there is an outgoing edge from u with label c' and increment $\mathit{Count}(z)$ by $|\mathit{range}(p_1c'p_2)|$. Let the heavy edge from u have label d; we also increment $\mathit{Count}(z')$, where $z' = \text{concat}(u, \text{concat}(v_d, w))$ is the node with path-label p_1dp_2, by $|I_z|$ while processing node z.

This procedure then recurs on each of the side trees; i.e. for side tree S_i, attached to node u, it calls PERFORMCOUNT($S_i, m - \mathscr{D}(u)$). Finally, we construct array C from array *Count* while performing one more depth-first traversal.

On the recursive calls of PERFORMCOUNT in each of the side trees (e.g. S_i) attached to *HP*, we first compute the heavy paths (in $\mathscr{O}(|S_i|)$ time for S_i) and then consider each node of string-depth m of $\mathscr{T}(x)$ at most once; as above, we process each node in $\mathscr{O}(\log\log n)$ time due to Lemmas 10 and 11. As there are at most n nodes of string-depth m, we do $\mathscr{O}(n\log\log n)$ work in total. This is also the case as we go deeper in the tree. Since the number of leaves of the trees we are dealing with at least halves in each iteration, there at most $\mathscr{O}(\log n)$ steps. Hence, each node of string-depth m will be considered $\mathscr{O}(\log n)$ times and every time we will do $\mathscr{O}(\log\log n)$ work for it. The overall time complexity of the

algorithm is thus $\mathcal{O}(n \log n \log\log n)$. The space complexity is $\mathcal{O}(n)$. By applying Theorem 4 we obtain the following result.

Theorem 6. Given a string of length n over a constant-sized alphabet and an integer m, where $1 \leq m < n$, the 1-MAPPABILITY problem can be solved in $\mathcal{O}(\min\{mn, n \log n \log\log n\})$ time and $\mathcal{O}(n)$ space.

Appendix B

Degenerate String Comparison and Applications

Proof. It is clear that $L(\hat{S}) \cap L(\hat{R}) \supseteq (L(\hat{R}') \cap L(\hat{S}')) \otimes (L(\hat{S}'') \cap L(\hat{R}''))$. Indeed, consider a string $x \in L(\hat{R}') \cap L(\hat{S}')$ and a string $y \in L(\hat{S}'') \cap L(\hat{R}'')$: then, by the definition of Cartesian concatenation, $xy \in L(\hat{R}') \otimes L(\hat{R}'') = L(\hat{R})$ and $xy \in L(\hat{S}') \otimes L(\hat{S}'') = L(\hat{S})$.

We now prove the opposite inclusion. Consider a string $z \in L(\hat{S}) \cap L(\hat{R})$. By definition, $z = x_0 x 1 \dots x_{r-1} = y_0 y_1 \dots y_{s-1}$, with $x_i \in \hat{R}[i], y_j \in \hat{S}[j], \forall\ 0 \leq i \leq r-1, \forall\ 0 \leq j \leq s-1$. Let $\hat{R}' = \hat{R}[0] \dots \hat{R}[\bar{i}]$, $\hat{S}' = \hat{S}[0] \dots \hat{S}[\bar{j}]$. Assume by contradiction that $z \notin (L(\hat{R}') \cap L(\hat{S}')) \otimes (L(\hat{S}'') \cap L(\hat{R}''))$: without loss of generality, $x_0 \dots x_{\bar{i}} \notin L(\hat{S}')$. Since $L(\hat{S}') \otimes L(\hat{S}'') = L(\hat{S})$, it follows that $z = x_0 x_1 \dots x_{r-1} \notin L(\hat{S}) \implies z \notin L(\hat{S}) \cap L(\hat{R})$, that is a contradiction. □

Proof. Again, let us assume without loss of generality that $w(\hat{R}[0]) > w(\hat{S}[0])$. We prove the result by induction on k.

[Level $k = 0$] By construction, n_0 contains strings in $\hat{R}[0] \cap (\text{chop}_0 \otimes \dots \otimes \text{chop}_{q_0})$, which

have length $|G^0_{\hat{R},\hat{S}}|$, and are also in $\hat{S}[0]$, and hence belong to both $L_0(\hat{S})$ and $L_0(\hat{R})$.

[Level $k > 0$] By inductive hypothesis, we have that $L_{k-1}(\hat{S}) \cap L_{k-1}(\hat{R}) = \text{paths}(G^{k-1}_{\hat{R},\hat{S}})$: suppose that $L_k(\hat{S}) \cap L_k(\hat{R}) \neq \emptyset$, otherwise the graph ends at level $k-1$. We first show that $\text{paths}(G^k_{\hat{R},\hat{S}}) \subseteq L_k(\hat{S}) \cap L_k(\hat{R})$: by Definition 8, any $z \in \text{paths}(G^k_{\hat{R},\hat{S}})$ can be written as $z = z'z''$ with z' in $\text{paths}(G^{k-1}_{\hat{R},\hat{S}})$ and with z'' that belongs to some node at level k of $G^k_{\hat{R},\hat{S}}$ reached by an edge leaving a suffix of z'. By inductive hypothesis $z' \in L_{k-1}(\hat{S}) \cap L_{k-1}(\hat{R})$ and, again by Definition 8, $z'' \in \text{chop}_{q_{k-1}+1} \otimes \cdots \otimes \text{chop}_{q_k}$; since $L_k(\hat{S}) \cap L_k(\hat{R}) \neq \emptyset$ these chops are not empty, their concatenation contains the suffix of length $|G^k_{\hat{R},\hat{S}}| - |G^{k-1}_{\hat{R},\hat{S}}|$ of strings in both $L_k(\hat{R})$ and $L_k(\hat{S})$, and hence $z \in L_k(\hat{S}) \cap L_k(\hat{R})$.

We now show that $L_k(\hat{S}) \cap L_k(\hat{R}) \subseteq \text{paths}(G^k_{\hat{R},\hat{S}})$: consider string $u \in L_k(\hat{S}) \cap L_k(\hat{R})$ that can be written as $u = u'u''$ with u' the prefix of u having length $|G^{k-1}_{\hat{R},\hat{S}}|$ which then belongs to $L_{k-1}(\hat{S}) \cap L_{k-1}(\hat{R})$; then, by inductive hypothesis, $u' \in \text{paths}(G^{k-1}_{\hat{R},\hat{S}})$ and, since $u \in L_k(\hat{S}) \cap L_k(\hat{R})$, then there is an edge linking a suffix of u' at level $k-1$ with a node at level k of $G^k_{\hat{R},\hat{S}}$ containing a $|G^k_{\hat{R},\hat{S}}| - |G^{k-1}_{\hat{R},\hat{S}}|$ long suffix u'' of u, and hence $u \in \text{paths}(G^k_{\hat{R},\hat{S}})$. □

Proof. The correctness follows directly from Lemma 2, Lemma 3, and Theorem 2.

Constructing the generalized suffix tree $T_{\hat{R},\hat{S}}$ can be done in time $\mathcal{O}(N+M)$ [28]. For the sets pair $(\hat{A}_i, \hat{B}_i)$ as in Definition 7, such that $w(\hat{A}_i) = k$ and $w(\hat{A}_i) \leq w(\hat{B}_i)$, we query $T_{\hat{R},\hat{S}}$ with the k-length prefixes of strings in $\hat{B}_i$. For integer alphabets, instead of spelling the strings from the root of $T_{\hat{R},\hat{S}}$, we locate the corresponding terminal nodes for $(\hat{A}_i, \hat{B}_i)$. It then suffices to find longest common prefixes between these suffixes to simulate the querying process. Since all suffixes are lexicographically sorted during the construction of $T_{\hat{R},\hat{S}}$, we can also have the suffixes considered by pair $(\hat{A}_i, \hat{B}_i)$ lexicographically ranked with respect

to $(\hat{A}_i, \hat{B}_i)$. Hence we do not perform the longest common prefix operation for all possible suffix pairs, but only for the lexicographically adjacent ones within this group. This can be done in $\mathcal{O}(1)$ time per pair after $\mathcal{O}(N+M)$-time pre-processing over $T_{\hat{R},\hat{S}}$ [16]. chop_i is thus populated with the k-length prefixes of strings in $\hat{B}_i$ found in $\hat{A}_i$. The set $\mathrm{active}_{\hat{A}_i,\hat{B}_i}$ of active suffixes can be found by chopping the suffixes of the string in $\hat{B}_i$ from their prefixes successfully queried in $T_{\hat{R},\hat{S}}$. This requires time $\mathcal{O}(|\hat{A}_i| + |\hat{B}_i|)$ for processing $(\hat{A}_i, \hat{B}_i)$.

Let $\hat{R}$ and $\hat{S}$ be of length r and s, respectively. Assume that $\hat{R}$ and $\hat{S}$ have no synchronized proper prefixes. Then Theorem 2 ensures that the total number of comparisons cannot exceed $r+s-2$: this results in a time complexity of $\mathcal{O}(N+M+\sum_{i=0}^{r+s-2}(|\hat{A}_i| + |\hat{B}_i|)) = \mathcal{O}(N+M)$.

If $\hat{R}$ and $\hat{S}$ have synchronized proper prefixes, we perform the comparison up to the shortest synchronized prefixes (i.e. the set of active suffixes becomes empty) and then restart the procedure from the immediately following pair of degenerate letters. Clearly the total number of comparisons also in this case cannot be more than $r+s-2$. □

B.1 A Conditional Lower Bound under SETH

In this section, we show a conditional lower bound for computing palindromes in degenerate strings. Let us first define the 2-Orthogonal Vectors problem. Given two sets $A = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ and $B = \{\beta_1, \beta_2, \ldots, \beta_n\}$ of d-bit vectors, where $d = \omega(\log n)$, the *2-Orthogonal Vectors* problem asks the following question: is there any pair α_i, β_j of vectors that is orthogonal? Namely, is $\sum_{k=0}^{d-1} \alpha_i[k] \cdot \beta_j[k]$ equal to 0? For the moderate dimension of this problem, we follow [40], assuming $n^{2-\varepsilon} d^{\mathcal{O}(1)} \leq n^2 d$. The following result is known.

Theorem 7 ([40, 59, 60, 98])**.** The 2-Orthogonal Vectors problem cannot be solved in $\mathcal{O}(n^{2-\varepsilon} \cdot d^{\mathcal{O}(1)})$ time, for any $\varepsilon > 0$, unless the *Strong Exponential Time Hypothesis* fails.

We next show that the 2-Orthogonal Vectors problem can be reduced to computing maximal palindromes in degenerate strings thus obtaining a similar conditional lower bound to the upper bound obtained in Theorem 4 for computing all GD palindromes.

Theorem 8. Given a degenerate string of length $4n$ over an alphabet of size $\sigma = \omega(\log n)$, all maximal GD palindromes cannot be computed in $\mathcal{O}(n^{2-\varepsilon} \cdot \sigma^{\mathcal{O}(1)})$ time, for any $\varepsilon > 0$, unless the *Strong Exponential Time Hypothesis* fails.

Proof. Let $d = \sigma$ and consider the alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. We say that two subsets of Σ *match* if they have a common element. Given a d-bit vector α, we define $\mu(\alpha)$ to be the following subset of Σ: $s \in \mu(\alpha)$ if and only if $\alpha[s] = 1$. Thus, two vectors α and β are orthogonal if and only if the sets $\mu(\alpha)$ and $\mu(\beta)$ are disjoint. In the string comparison setting, two degenerate letters $\mu(\alpha)$ and $\mu(\beta)$ *do not match* if and only if α and β are orthogonal. The reduction works as follows. Given $A = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ and $B = \{\beta_1, \beta_2, \ldots, \beta_n\}$, we construct the following simple degenerate string of length $4n$ in time $\mathcal{O}(n\sigma)$:

$$S = \underbrace{\mu(\alpha_1)\mu(\beta_1)\mu(\alpha_2)\mu(\beta_2)\ldots\mu(\alpha_n)\mu(\beta_n)}\mu(\alpha_1)\mu(\beta_1)\mu(\alpha_2)\mu(\beta_2)\ldots\mu(\alpha_n)\mu(\beta_n).$$

$$\ldots$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

Then the 2-Orthogonal Vectors problem for the sets A and B has a positive answer if and only if at any position of S, from 0 to $2n$, there *does not occur* a palindrome of length at least $2n$. All such occurrences can be easily verified from the respective palindrome centers in time $\mathcal{O}(n)$. In other words, if at any position of S there does not occur a palindrome of length at least $2n$, this is because we have a mismatch between a pair $\mu(\alpha_i), \mu(\beta_j)$ of letters, which implies that there exists a pair α_i, β_j of orthogonal vectors. Also, by the construction, all such pairs are to be (implicitly) compared, and thus, if there exists any pair that is orthogonal the corresponding mismatch will result in a palindrome of length less than $2n$. □

Appendix C

Efficient identification of k-closed strings

Proof. Since x is k-closed, it has at least one k-closed border and an associated smallest $k' \leq k$ for which the conditions are satisfied. Let us consider the longest of these k-closed borders, and call u and v the prefix and suffix respectively, comprising the longest k-closed border with length $|u| = |v|$. Let us assume a second k-closed border exists, comprised of the prefix and suffix, u' and v' respectively. We know that $|u'| = |v'| < |u| = |v|$ and $u' = u[0\,..\,|u'|-1]$. Since $u \approx_{k'} v$ it is trivially true that $u[0\,..\,|u'|-1] \approx_{k'} v[0\,..\,|u'|-1]$ and therefore $u' \approx_{k'} v[0\,..\,|u'|-1]$. Thus we see that u' k'-matches the prefix of v of the same length, and this corresponds to an occurrence of u' within x, i.e. $u' \approx_{k'} x[n-|v|\,..\,n-|v|+|u'|-1]$, where n is the length of x, which is an internal occurrence of u' in x. We arrive at a contradiction due to Condition 2 of Definition 10 being violated, therefore no second k-closed border can exist. □

Proof. Recall that $n \geq 2$. The three conditions can be seen to be necessary and sufficient for a string to be k-closed by considering the cases individually.

($\Longrightarrow$) Suppose Conditions 1-3 hold. We need to show that x is k-closed. We first prove that the conditions imply x is k'-pseudo-closed. In other words, we need to find a prefix u of x and a suffix v of x such that:

(I) $u \approx_{k'} v$

(II) Except for u and v, there exists no length-$|u|$ factor w of x such that $w \approx_{k'} u$ or $w \approx_{k'} v$.

First, Condition 1 implies that the longest prefix match within k' errors starting at j terminates at position $n-1$ in x. This implies that $u = x[0..n-1-j] \approx_{k'} x[j..n-1] = v$. Hence, (I) is true.

By contrary of (II), we have either (1) a factor w starting at position $i < j$ such that $w \approx_{k'} u$ or (2) a factor w ending at position $i > n-1-j$ such that $w \approx_{k'} v$.

For (1), this means that $\mathrm{LPM}_{k'}(x)[i] \geq \mathrm{LPM}_{k'}(x)[j]$. However, this contradicts Condition 2.

For (2), this means that $\mathrm{LSM}_{k'}(x)[i] \geq \mathrm{LSM}_{k'}(x)[n-1-j]$. However, this contradicts Condition 3.

Hence, both (I) and (II) are true. This implies that x is k'-pseudo-closed. Since $0 \leq k' \leq k$, we may further imply by Lemma 7 that x is k-closed.

($\Longleftarrow$) If x is k-closed, there must exist some k', where $0 \leq k' \leq k$, such that x is k'-pseudo-closed, by Lemma 7. For such a k', there is an associated k'-pseudo-closed-border consisting of some proper prefix u and some proper suffix v with equal length, such that

$\delta_H(u,v) \leq k'$. We denote j where $v = x[j..n-1]$ and consequently $u = x[0..n-1-j]$. The longest prefix match $\text{LPM}_{k'}(x)[j]$ starting at j must be greater than or equal to $|u|$ as $u \approx_{k'} v$, yet it may not exceed the bounds of x and is therefore less than or equal to $|v|$. Therefore $\text{LPM}_{k'}(x)[j] = |u| = |v| = n-j \implies \text{LPM}_{k'}(x)[j] + j = n$ which implies Condition 1. From the definition of k-closed strings we also conclude that there exists no factor w of x with length $|w| = |u| = |v|$ such that $\delta_H(u,w) \leq k'$ or $\delta_H(v,w) \leq k'$. Therefore if we choose $i < j$ it must be the case that $\text{LPM}_{k'}(x)[i] < \text{LPM}_{k'}(x)[j]$, since otherwise we would have a $w \approx_{k'} v$ starting at i which cannot be the case, and therefore we conclude Condition 2. Similarly if we choose $i > n-1-j$ it must be the case that $\text{LSM}_{k'}(x)[i] < \text{LSM}_{k'}(x)[n-1-j]$, since otherwise we would have a $w \approx_{k'} v$ ending at i which cannot be the case, and therefore we conclude Condition 3. Thus all three conditions are satisfied.

□