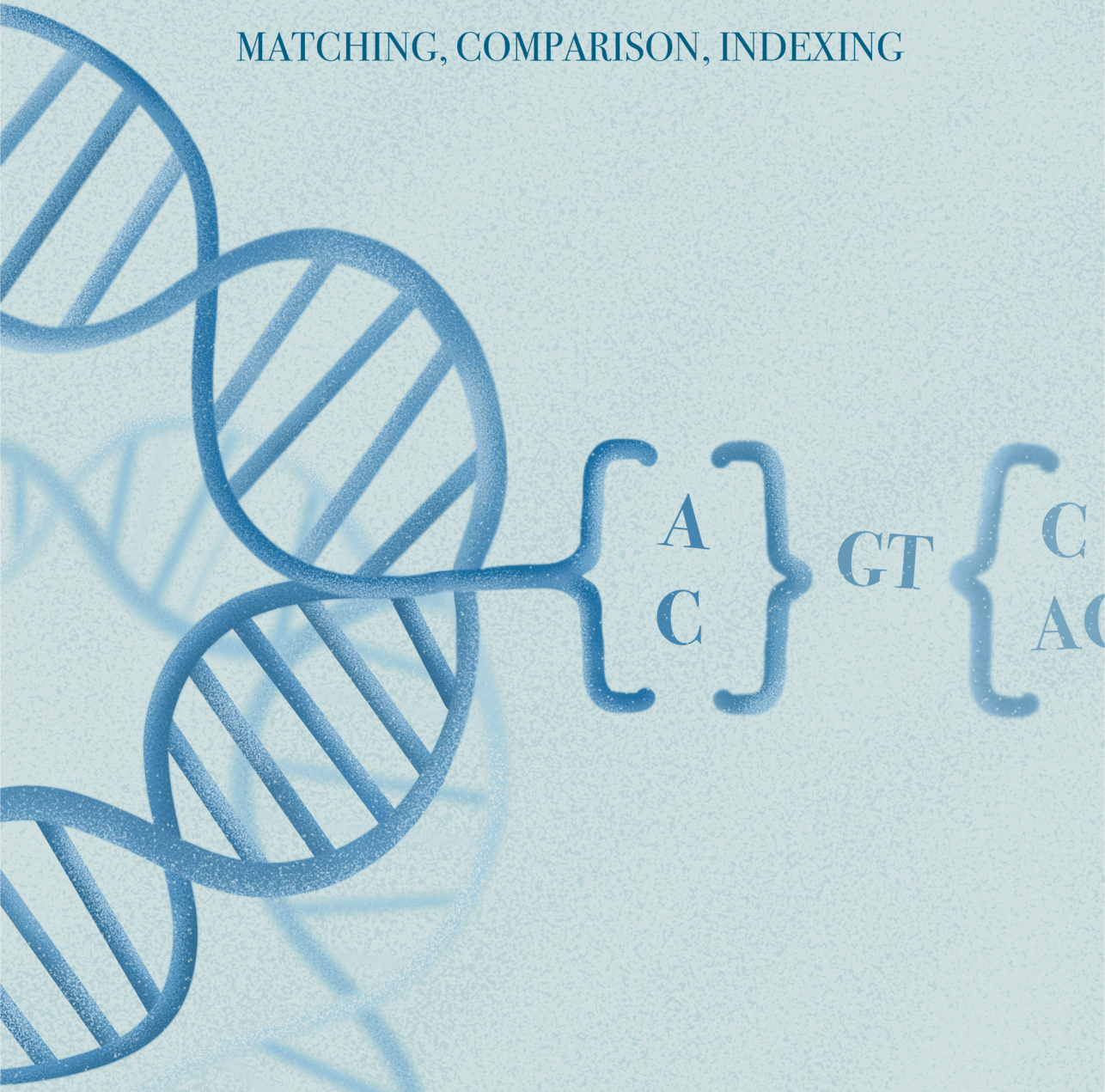
VARIABLE STRINGS FOR PANGENOMES

MATCHING, COMPARISON, INDEXING



Estéban Gabory

VRIJE UNIVERSITEIT

VARIABLE STRINGS FOR PANGENOMES: MATCHING, COMPARISON, INDEXING

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor of Philosophy aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. J.J.G. Geurts,
volgens besluit van de decaan
van de Faculteit der Bètawetenschappen
in het openbaar te verdedigen
op woensdag 21 mei 2025 om 15.45 uur
in de universiteit

door

Estéban Arthur Denis Gabory

geboren te Saint-Herblain, Frankrijk

promotoren:	dr. S. Pissis prof.dr. L. Stougie
promotiecommissie:	prof.dr. V. Mäkinen prof.dr. A. Schoenhuth prof.dr. M. Sciortino prof.dr. W.J. Fokkink dr. L. Iersel

This research has been carried out in the Networks and Optimization group at the Centrum Wiskunde & Informatica in Amsterdam and was partially supported by the ALPACA project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 956229.

ISBN: 978-90-6196-430-8

Co-financed by the Connecting Europe Facility of the European Union

Cover designed by Julia Paulina Gancarz

Contents

Acknowledgements

I would like to express my deepest gratitude to everyone who has supported me throughout this journey. Completing my thesis would not have been possible without their presence, encouragement, and belief in me.

First and foremost, I am sincerely grateful to my supervisor, Solon Pissis, for his invaluable mentorship, insightful advice, and unwavering support. I am equally thankful to my co-supervisor, Leen Stougie, for his guidance, encouragement, and constructive feedback throughout the process.

I would also like to thank the members of my thesis committee—Dick Bulterman, Wan Fokkink, Leo van Iersel, Veli Mäkinen, Alexander Schönhuth, and Marinella Sciortino—for their thoughtful input and for taking the time to participate in my defense.

A heartfelt thank you goes to my M246 colleagues, Michelle and Hilde, as well as to Giulia and Wiktor, to all the members of N&O group. Your stimulating discussions and constant support were a source of motivation throughout.

I am also grateful to the colleagues I had the pleasure of meeting regularly during my PhD: Giuseppe—for our frequent encounters at conferences, and for introducing me to the best arancine, which became essential fuel to finish this thesis —and all the ALPACA students: Adrian, Alessia, Andreas, Andrej, Daria, Fawaz, Francesco, Hugo, Jorge, Khodor, Konstantin, Leonard, Luca, Nicola, Njagi, Pengfei, Pio, and Ragu. Our shared experiences across various countries have truly been a highlight of this journey.

My sincere thanks also go to all the researchers and staff involved in the ALPACA ITN. The network's events and collaborations have made this experience both intellectually enriching and personally rewarding.

I am also deeply thankful to the team at University of Pisa, where I had the opportunity to work for some time—Nadia, Roberto, Alessio, Giulia, and Veronica. Thank you for making my stay so enjoyable and fulfilling, for the support, inspiring conversations, and good moments we shared.

I want to acknowledge all the people who, long before I began this work, helped shape my curiosity and passion for science: many of my teachers—from middle school onwards— and Maxime, who showed me long time ago that mathematics could be more than a fixed set of arbitrary rules.

Merci à Simon pour les conversations sans fin et les glaces a Rome et à Palerme, à Logan pour les autres discussions tout aussi interminables et les soupes étranges en revenant du marché des Lices.

Enfin, un grand merci à mes proches, qui savent déjà combien ils comptent pour moi. Un immense merci à mes parents, pour avoir cru en moi même quand ce n'était pas gagné. Ash, merci d'être toujours là. Tu me rends chaque jour plus fier de t'avoir comme adelphe.

Julia, this thesis would simply never exist without you. Thank you for being my partner and for supporting me—even when I was switching, in the span of an hour, from "I want to give up and start selling cheese" to "I'm about to make the greatest discovery ever," and back again. I hope you know how much you helped me, and how much it means to me to start building a life together. Dziękuję Ci za to, że sprawiłaś, iż moje życie stało się lepsze, Doktorowa.

1

Introduction

1.1 Motivation and outline

1.1.1 Biological Motivation: From molecules to strings, from strings to pangenomes

This thesis focuses on theoretical aspects of sequence analysis for *pangenomes*, hence we start with a brief motivation on the importance of strings in bioinformatics and the paradigm shift towards pangenomes.

A possible starting point is 1871, when Friedrich Miescher first discovered the *DNA* molecule [113, 166]. However, the first accurate description of the DNA structure was published in 1953 by James Watson and Francis Crick [211], building on the earlier work of Rosalind Franklin. Present in the cells of every living organism, DNA is a molecule composed of two complementary strands of successive *nucleotides*. These nucleotides belong to one of four types: *cytosine* (C), *guanine* (G), *adenine* (A), or *thymine* (T). Hence, each DNA molecule can be abstracted to the sequence of its nucleotides, seen as letters. This allowed abstracting DNA molecules to a simple *string* over a 4 letter alphabet, disregarding its molecular complexity. The central importance of DNA in biology stems from its role as the template (via the *genetic code*) for the synthesis of *proteins* (through various complex molecular interactions), which in turn determine most of an organism's characteristics. Similarly, other molecules such as *RNA*, or proteins themselves, can be described as sequences of letters over a finite alphabet. This abstraction motivates the importance of strings, and of sequences algorithms, for bioinformatics.

Genomic data analysis has been facing important challenges that include analyzing an ever-increasing number of genome sequences and choosing which genome should be used as a *reference*. In recent years, these two challenges were merged into the powerful opportunity of using a *pangenome* – rather than a single genome – as a reference. According to [70], a *pangenome* is "any collection of genomic sequences to be analyzed jointly or to be used as a reference". As a consequence, the new -omics science *pangenomics* imposed a paradigm shift: in several analysis tasks, and in particular for species like humans that enjoy a widespread availability of

sequencing data as well as a growing awareness of genomic variants, the simple linear genomic sequence is being replaced by more complex graph-like structures [89, 172]. As opposed to a *linear* reference, a pangenome reference allows a simultaneous representation, in a compact manner, of variations and commonalities among the underlying sequences.

When shifting to pangenomes, a natural theoretical challenge is raised: how can we adapt well-known problems and algorithms, that were studied on classical strings, to those nonlinear sequences? In this thesis, we will focus on those problems. In particular, we will study generalization of 3 classical tasks of sequence analysis: *pattern matching*, *comparison*, and *indexing*.

1.1.2 Classical tasks of sequence analysis

Pattern matching String matching (or pattern matching) is a fundamental task in computer science. Given a longer string (the *text*) and a shorter string (the *pattern*), one asks whether the pattern occurs at least once in the text (in the *decision version*), to report the number of occurrences of the pattern (*counting version*), or to list all the positions of the text where the pattern occurs (*reporting version*). Several algorithms have been designed since the late 1970s, starting with the most standard task of finding a single standard pattern in a single standard string, which can be done in linear time [72]. This was only the starting point of a line of research that extended to, for example, approximate matching [22, 55, 56, 66, 103, 150, 151], pattern matching on different data structures [24, 42, 43, 109, 122, 125, 192]...

In particular, we will study exact and approximate (when occurrences of string at a small *distance* from the input pattern are accepted) pattern matching, on data structures that can be used to model pangenomes.

Comparison Sequence (or string) comparison is a fundamental task in computer science, with numerous applications in computational biology [112], signal processing [80], information retrieval [33], file comparison [114], pattern recognition [27], security [164], and elsewhere [170]. Given two or more sequences and a distance function, the task is to compare the sequences in order to infer or visualize their (dis)similarities [72].

Among classical ways of comparing sequences, one might use the *Hamming distance*, that is the number of mismatches between two string having a same length, or the *edit distance*, that is the minimum number of deletions, insertions or letter replacements needed to turn one given string into another. Hamming distance can be easily computed in linear time, but edit distance, often more fitting real life models, requires quadratic time under a well-established plausible hypothesis [48], which can be inefficient in some applications.

We also mention the *matching statistics* [112] between two strings, which has some sensitivity to local similarities and detects shared arbitrary-length fragments - which is an important property of edit distance, while being much faster to compute, making it widely used in practice, particularly in bioinformatics [154, 204, 206]. The matching statistics between two strings T_1, T_2 is an array that, for each possible

starting position i in T_1, gives the length of the longest fragment of T_1 starting at position i that appears anywhere in T_2.

Indexing Given the ever increasing size of string data, it is crucial to represent them compactly but also to *simultaneously* allow efficient pattern searches. This goal is formalized by the classic *text indexing* problem [72]: preprocess a text into a data structure that supports pattern matching queries.

In text indexing we are interested in four efficiency measures [28, 138]: **(i)** How much space does the final index occupy for a text T of length n (the *index size*)? **(ii)** How fast can we search for a query pattern P of length m (the *query time*)? **(iii)** How much working space do we need to construct the index (the *construction space*)? **(iv)** How fast can we construct the index (the *construction time*)?

The classic indexing solution for standard strings is the *suffix tree* [212]. Given a text T having length n, its suffix tree (written $\mathrm{ST}(T)$) is defined as the compacted trie[1] of all the suffixes of T. The suffix tree occupies $\mathcal{O}(n)$ space and it can be constructed in $\mathcal{O}(n)$ time [91, 212]. It supports queries of a pattern of length m in $\mathcal{O}(m+|\mathbf{Occ}|)$ time, where **Occ** is the set of all the occurrences of the pattern in T. The *suffix array* [163] of T is the lexicographically ordered list of leaf nodes in $\mathrm{ST}(T)$. The suffix array of T can also be computed in $\mathcal{O}(n)$ time.

1.1.3 DATA STRUCTURE FOR PANGENOMES

In the following, we will describe informally the different data structure that we consider in this thesis to encode pangenomes. Among this thesis, we will use the general term *variable string* to mention any of such a data structure.

The most straighforward representation is the Multiple Sequence Alignment (MSA). It comprises a collection of strings, potentially containing gaps introduced to maintain the alignment of similar segments (see Figure 1.1).

While MSAs accurately store pangenomes by preserving each string, they are not space-efficient. Since pangenome strings are typically highly similar, storing multiple copies of the same segments is often undesirable. Thus, we need data structures to efficiently store multiple similar string.

A first simple type of variable strings is obtained by collapsing MSA columns, only keeping the set of letters occurring in each of them. This results in a sequence of sets of letters, that we call a *degenerate string* (1-*D*) (see Figure 1.1), also known as *indeterminate* strings in the bioinformatics literature. They have been extensively employed and investigated [8, 13, 20, 23, 46, 73, 76, 77, 81, 102, 116, 124, 125, 160, 173, 174, 187, 188, 197], and can be used to encode single-gene mutations, known as SNPs (Single Nucleotide Polymorphisms). In practice, they are often implemented via the IUPAC encoding [127] for DNA and RNA, that encodes each possible subset of the alphabet as a special letter.

To accommodate mutations across entire genome segments, *generalized degenerate strings* (*GD*) are defined as sequences of sets, each containing strings of a specified

[1] A trie is *compacted* if every internal node has at least 2 children. Any trie can be compacted in linear time by merging each internal node that has a single child to its child, and updating the labels acordingly. In particular, a compacted trie having n leaf nodes has $\mathcal{O}(n)$ nodes in total.

1

length k. If this length k remains constant across all sets within a *GD* string, we call it a *k-degenerate string* (*k-D*). Notably, the case where $k = 1$ corresponds to the definition of a degenerate string.

While those data structures allow compact representations for multiple similar strings, *elastic degenerate strings* (*ED* strings) [109, 122] provide additional flexibility by allowing representation of strings having varying lengths, and gaps within the alignment. An *ED* string is a sequence of finite sets of strings having any arbitrary finite length. Additionally, sets can also contain the *empty string* ε. *ED* strings are the main focus of the present thesis.

We also mention *Weighted sequences* [130], that are studied in Chapter 4. A weighted sequences is a string in which, at each position, several letters can occur with a given *probability*, depending on the letter and the position. They are also known as *position weight matrices* in bioinformatics [136].

Finally, many other very important (and more general) types of variable strings are *graph-based* [30], such as the *variation graphs* [82, 83, 195] and *sequence graphs* [180]. In this thesis, we will only focus on the graph versions of the variable strings we described, namely *Elastic Founder graphs* [181] (*EF* graphs), that is an *ED* string with an additional graph data structure: each pair of strings from adjacent sets can be connected by an edge, meaning that the corresponding variants can occur simultaneously in a given genome. Note that, in the literature, the convention is that *EF* graphs do not contain the empty string ε. In addition, the described taxonomy of special cases for *ED* strings gives us a symmetric taxonomy for *EF* graphs: A *EF* graph is called a 1-*F*, *k*-*F*, *F* if its underlying string is, respectively, a 1-*D*, *k*-*D*, *GD*.

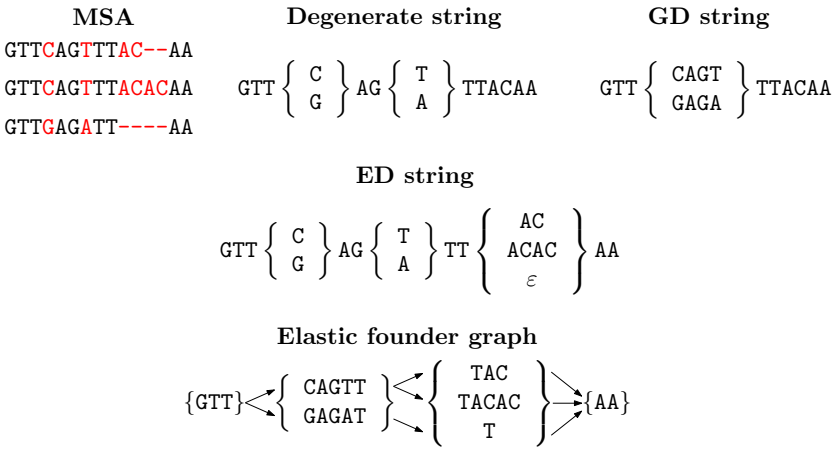


Figure 1.1: An example of an MSA (top left), its (non-unique) corresponding 1-*D* string (top middle), *GD* string (top right), *ED* string (middle row), and *EF* graph (bottom row). For sets containing a single string, the set notation is lifted for simplicity.

Observe Figure 1.1: from the MSA are constructed corresponding 1-*D* string, *GD* string, *ED* string and *EF* graph. Note that only the *ED* string and *EF* graph can represent the gaps in the first and third string of the MSA, since every string

represented by a given 1-*D* or a given *GD* must have the same length. Additionally, observe that in the MSA, the letter G occurs at the 4th position only in strings where the letter A occurs at the 7th position, and in this case, the substring AC does not occur. This information is lost in the 1-*D* string, *GD* string, or *ED* string: for example, observe that one can read GTTGAGTTTACAA in the *ED* string, which does not belong to the MSA. This models mutations happening independently, in which case unobserved recombinations of observed mutations are considered to be biologically relevant. For a more flexible description of the compatibility between mutations that occur at close range, one can use *EF* graphs: in Figure 1.1, the *EF* graph does not encode any recombination that does not occur in the MSA.

In contrast with more general representations, *ED* strings support both theoretically [24, 42, 43] and practically [49, 63, 109, 175, 175, 177] fast *online* exact pattern matching [44, 45] when the pattern is a solid (i.e. non-variable) string. Moreover, it can be efficiently decided whether two *ED* strings share a string [95] Finally, *GD* strings support fast dynamic-programming-based alignment [167, 168] for approximate matching with a solid string, and linear-time answer to standard comparison tasks [15, 16]. On the other hand, *ED* strings cannot be indexed efficiently [105].

For *EF* graphs, efficient off-line pattern matching algorithms are known under specific conditions which can be ensured with a linear-time construction from a gapless MSA (for *F* graphs), or a parameterized linear-time construction from a general MSA (for *EF* graphs) [87, 88, 161, 182, 183, 202].

While supporting provably efficient methods, both *ED* strings and *EF* graphs are acyclic structures that impose a global alignment allowing only matches, mismatches, and short insertions and deletions as variants: they cannot adequately represent structural variations such as repetitions, translocations, or inversions.

Computational pangenomics must balance efficiency, accuracy, and the complexity of representing variable strings as graphs [70].

1.1.4 OUTLINE OF THE THESIS

- Chapter 2 is structured in two sections:

  Section 2.1 is based on [26] and studies exact pattern matching on variable string. We investigate a clear taxonomy of the computational complexity of finding the occurences of a pattern P in a text T, where P and T range through all types of variable strings. In particular, for each possible configuration, our aim is to either find a strongly subquadratic algorithm, either prove that no such algorithm can exist, under standard assumptions.

  Section 2.2 is based on [40], and focuses on approximate pattern matching on *ED* text. When given a pattern and a *ED* text, one wants to find occurences of the pattern with *at most k* edits, or *at most k* mismatches, for a fixed k. While algorithms were already known ([45]), our aim was to design, for the case $k = 1$, an algorithm that depends linearly on the total size of the *ED* string. We present such an algorithm that, after preprocessing the pattern, allows to find its approximate occurences in an *ED* text by reading it linearly.

- Chapter 3 is based on [95, 96, 98], and is dedicated to pangenomes *comparison*.

1

Given two pangenomes, represented as *ED* strings, we ask to find whether a single string is encoded by both *ED* strings. After giving conditional lower bounds on the complexity of this problem, we introduce *intersection graph*, that encode in a compacted way the shared information of both *ED* strings, and can be used to compute their intersection efficiently. We show that intersection graphs can be used for other, more elaborate comparison tasks: in particular, we generalize the notion of *matching statistics*, which is heavily employed on standard strings for practical applications, to *ED* strings, and we show that it can be computed efficiently using intersection graphs.

- Chapter 4 focuses on indexing, and is based on [94]. We study here a different type of data structure, called *weighted sequences*. A weighted sequence encodes *uncertainty* in strings: at each position, several letters can occur with a given probability. Our goal is to, given a fixed *threshold*, preprocess such a sequence into a data structure that can efficiently answer to the following queries: does a pattern P occurs with a probability larger than the threshold? While such index were already known, they require large construction and storing spaces for long texts and small probability thresholds. We were thus motivated to design a space-efficient index at the expense of slower yet competitive pattern matching queries. We describe such an index, that answer queries for patterns larger than a fixed lower bound. We have implemented and evaluated several versions of our index.

1.2 PRELIMINARIES

In this section, we give the main definitions that are relevant to this thesis.

1.2.1 STRINGS AND VARIABLE STRINGS

We start with some arithmetical notations: Given two rational numbers p, q, we denote by $[p,q]$ the interval of rationals between p and q, namely $[p,q] = \{r \in \mathbb{Q},\ p \leq r \leq q\}$. Given two integers $i,\ j$ we denote by $[i\,..\,j]$ the interval of integers between i and j, namely $[i\,..\,j] = [i,j] \cap \mathbb{Z}$. In particular, one has $[i\,..\,j] = \emptyset$ if $j < i$.

We now give some basic definitions and notation on solid strings following [72]. Given an ordered alphabet Σ whose elements are called *letters*, we denote by Σ^n the set of *strings* $T = T[1]\dots T[n]$ (also called *standard* or *solid* strings) of length $|T| = n$ over Σ. The *empty string* is the string of length 0; we denote it by ε. We write Σ^* for the set of strings of arbitrary length (including 0) on Σ.

By T_1T_2 or $T_1 \cdot T_2$ we denote the *concatenation* of two strings T_1 and T_2, i.e., $T_1T_2 = T_1[1]\dots T_1[|T_1|]T_2[1]\dots T_2[|T_2|]$. Given p strings $T_1, .. T_p$ we write $\prod_{i=1}^{p} T_i$ for the concatenation $T_1 \cdot T_2 \cdot \dots \cdot T_p$. Given a string T and an integer p we write $T^p = \prod_{i=1}^{p} T$ for the *pth power* of T, and $T^R = T[|T|]\dots T[1]$ for the *reverse* of T.

For any two positions i and $j \geq i$ of T, $T[i\,..\,j]$ is the *fragment* of T starting at position i and ending at position j. The fragment $T[i\,..\,j]$ is an *occurrence* of the underlying *substring* $P = T[i]\dots T[j]$; we say that P occurs at *position* i in T, and that P is a *factor* of T. As a convention, we set $T[\emptyset] = \varepsilon$. A *prefix* of T is a fragment of the form $T[1\,..\,j]$ and a *suffix* of T is a fragment of the form $T[i\,..\,n]$. Given two strings T_1

and T_2 we write $\mathrm{LCP}(T_1, T_2)$ for the length of their *longest common prefix*, namely for the integer $\max(\{i,\ T_1[1..i] = T_2[1..i]\} \cup \{0\})$. We will make use of the following result in several chapters:

Theorem 1 ([72]). *Given a string T over an integer alphabet, we can construct a data structure over T in $\mathcal{O}(|T|)$ time, so that when $i, j \in [1..|T|]$ are given to us online, we can determine $\mathrm{LCP}(T[i..|T|], T[j..|T|])$ in $\mathcal{O}(1)$ time.*

Given a string T over Σ, we call *suffix tree* of T the compacted trie $\mathrm{ST}(T)$ of all suffixes of $T \cdot \$$ where $\$ \notin \Sigma$ is an extra letter.

Theorem 2 ([91, 212]). *The suffix tree of a string $T[1..n]$ occupies $\mathcal{O}(n)$ space and it can be constructed in $\mathcal{O}(n)$ time.*

Given two strings T_1 and T_2 over an alphabet Σ, we define the *edit distance* $d_E(T_1, T_2)$ between T_1 and T_2 as the length ℓ of a shortest sequence of string operations $\pi_1, \ldots, \pi_\ell$ such that $T_2 = (\Pi_{i=1}^{\ell} \pi_i)(T_1)$, where each π_i (for $1 \le i \le \ell$) is one of the following type:

- *Replacement:* There is $j \in [1..|T_1|]$ and $\sigma \neq T_1[j] \in \Sigma$ s.t. $\pi_i(T_1)[j] = \sigma$ and $\pi_i(T_1)[j'] = T_1[j']$ for $j' \neq j$.
- *Deletion:* One has $|\pi_i(T_1)| = |T_1| - 1$ and there is $j \in [1..|T_1|]$ s.t. $\pi_i(T_1)[j'] = T_1[j']$ for $1 \le j' \le j-1$ and $\pi_i(T_1)[j'] = T_1[j'+1]$ for $j \le j' \le |T_1|-1$.
- *Insertion:* One has $|\pi_i(T_1)| = |T_1|+1$ and there is $j \in [1..|T_1|+1]$ s.t. $\pi_i(T_1)[j'] = T_1[j']$ for $1 \le j' \le j-1$ and $\pi_i(T_1)[j'] = T_1[j'-1]$ for $j+1 \le j' \le |T_1|+1$.

Lemma 3 ([72]). *The function d_E is a distance on Σ^*.*

The following lemma is a simple reformulation of the definition:

Lemma 4. *If T_1, T_2 are two strings with $d_E(T_1, T_2) = 1$, then $T_1 = \pi(T_2)$ where π is a single replacement, deletion, or insertion.*

We now give the formal definitions on variable strings introduced in Subsection 1.1.3:

An *elastic-degenerate string* (*ED* string) [121] $\widetilde{T} = \widetilde{T}[1] \ldots \widetilde{T}[n]$ having *length* n over an alphabet Σ is a sequence of $n = |\widetilde{T}|$ finite sets, called *segments*, such that for every position i of $\widetilde{T}$ we have that $\widetilde{T}[i] \subset \Sigma^*$. By $N = ||\widetilde{T}||$ we denote the total length of all strings in all segments of $\widetilde{T}$, which we call the *size* of $\widetilde{T}$; more formally, $N = \sum_{i=1}^{n} \sum_{j=1}^{|\widetilde{T}[i]|} |\widetilde{T}[i][j]|$, where by $\widetilde{T}[i][j]$ we denote the jth string of $\widetilde{T}[i]$ (to consider the practical data structure size, we also add 1 to account for empty strings: if $\widetilde{T}[i][j] = \varepsilon$, then we have that $|\widetilde{T}[i][j]| = 1$). For a given segment $\widetilde{T}[i]$ of T, we denote by B_i its *cardinality* $B_i = |T_i|$, and by B we denote the *cardinality* of $\widetilde{T}$, namely the total number of strings in all segments, i.e., $B = \sum_{i=1}^{n} B_i$.

If for every i the strings in $\widetilde{T}[i]$ have all the same length k_i (called the *width* of $\widetilde{T}[i]$), we say that $\widetilde{T}$ is a *generalised degenerate (GD) string*. If in addition all segments $\widetilde{T}[i]$ have the same width k, $\widetilde{T}$ is a *k-degenerate* string (k-D, in short). In the special case

1

$k = 1$, then $\widetilde{T}$ is known in the literature as a *degenerate* or *indeterminate* string. Finally, if for every i it holds that $B_i = 1$, then we have a solid string.

Given two sets of strings $\mathcal{S}_1$ and $\mathcal{S}_2$, their *concatenation* is $\mathcal{S}_1 \cdot \mathcal{S}_2 = \{T_1T_2 \mid T_1 \in \mathcal{S}_1, T_2 \in \mathcal{S}_2\}$. For an ED string $\widetilde{T} = \widetilde{T}[1..n]$, we define the *language* of $\widetilde{T}$ as $\mathcal{L}(\widetilde{T}) = \widetilde{T}[1] \cdot \ldots \cdot \widetilde{T}[n]$, that is, the set of all strings that can be obtained by picking one string in each segment of T and concatenating them from left to right. Given a set $\mathcal{S}$ of strings, we write $\mathcal{S}^R$ for the set $\{T^R \mid T \in S\}$. For an ED string $\widetilde{T} = \widetilde{T}[1..n]$ we write $\widetilde{T}^R$ for the ED string $\widetilde{T}[n]^R \ldots \widetilde{T}[1]^R$.

Given a string P and an ED string $\widetilde{T}$, we say that P *matches* the fragment $\widetilde{T}[j..j'] = \widetilde{T}[j] \ldots \widetilde{T}[j']$ of $\widetilde{T}$, or that an *occurrence* of P *starts* at position j and *ends* at position j' in $\widetilde{T}$ if there exist two strings U, V, each of them possibly empty, such that $U \cdot P \cdot V = P_j \cdot \ldots \cdot P_{j'}$, with $P_i \in \widetilde{T}[i]$, for every $j \le i \le j'$, where $P_j = U \cdot P \cdot V \in \widetilde{T}[j]$ if $j = j'$, or $P_j = U \cdot P_p$, and $P_{j'} = P_s \cdot V$ for some prefix P_p (resp. suffix P_s) of P if $i < j$. If U is nonempty (resp. if V is nonempty) we also require P_p (resp. P_s) to be nonempty.

Strings P_i, for every $j \le i \le j'$, specify an *alignment* of P with $\widetilde{T}[j..j']$. For each occurrence of P in $\widetilde{T}$, the alignment is, in general, not unique. We call $|U|$ and $|V|$ the starting and ending *offset* of the alignment, respectively. In Figure 1.1, $P = \mathrm{TTA}$ matches $\widetilde{T}[5..6]$ with two alignments: both have $U = \varepsilon$, $P_5 = \mathrm{TT}$, and P_6 is either AC (with $V = \mathrm{C}$) or ACAC (with $V = \mathrm{CAC}$).

An *elastic founder (EF) graph* is a pair $G = (\widetilde{T}, E)$, where $\widetilde{T}$ is an *ED* string s.t. the empty string ε is not an element of any of its segments, and $E = \bigcup_{i=1}^{n-1} E_i$, where E_i is the set of edges from $\widetilde{T}[i]$ to $\widetilde{T}[i+1]$, which can be identified with a subset of the Cartesian product $[1..B_i] \times [1..B_{i+1}]$ (where $B_i = |T[i]|$ for each $1 \le i \le n$). We say that G is a *founder graph*, F graph in short (resp. a *k-founder graph*, *k-F* in short) if $\widetilde{T}$ is a *GD* (resp a *k-D*) string. We write $G[i..j] = (\widetilde{T}[i..j], \bigcup_{\ell=i}^{j-1} E_\ell)$ for the *fragment* of G between $\widetilde{T}[i]$ and $\widetilde{T}[j]$. The size of G is $N + |E|$, i.e. the sum of the size of the underlying *ED* string $\widetilde{T}$ and the total number of edges of G.

Given a string P and a *EF* graph $G = (\widetilde{T}, E)$ we say that P *matches* the fragment $\widetilde{T}[j..j']$ of G, or that an *occurrence* of P *starts* at position j and *ends* at position j' in G if P matches $\widetilde{T}[j..j']$ in $\widetilde{T}$ with an alignment $P_j..P_{j'}$, and that $(P_i, P_{i+1}) \in E_i$ for every $j \le i \le j' - 1$.

Figure 1.1 shows an *ED* string of length $n = 7$ and size $N = 20$, and an *EF* graph with $n = 4$, $N = 24$, $|E| = 8$, and hence size 32.

1.2.2 USEFUL TECHNIQUES

COMPUTATIONAL GEOMETRY

Some problems from computational geometry have a key role in our solutions. We assume the word RAM model with coordinates on the integer grid $[1..n]^d = \{1, 2, \ldots, n\}^d$ for $d = 2$. In the *2D RECTANGLE STABBING* problem, we are given a set $\mathcal{R}$ of n axis-aligned rectangles to be preprocessed, so that when one gives a point as a query, we report YES if and only if there exists a rectangle from $\mathcal{R}$ containing the point. In the "dual" *2D RECTANGLE EMPTINESS* problem, we are given a set $\mathcal{P}$ of n points to be preprocessed, so that when one gives an axis-aligned rectangle as a

query, we report YES if and only if the rectangle contains a point from $\mathcal{P}$; in the *2D RANGE REPORTING* problem, we report the number of points from $\mathcal{P}$ contained in the rectangle.

Lemma 5 ([57, 191]). *After $\mathcal{O}(n \log n)$-time preprocessing, we can answer 2D RECTANGLE STABBING queries in $\mathcal{O}(\log n)$ time.*

Lemma 6 ([52, 101]). *Assume that the points from $\mathcal{P}$ are obtained from pairing two permutations of $[1..N]$. After $\mathcal{O}(n\sqrt{\log n})$-time preprocessing, we can answer 2D RECTANGLE EMPTINESS queries in $\mathcal{O}(\log \log n)$ time.*

Lemma 7 (Section 2 of [162]). *After a $\mathcal{O}(n)$-time preprocessing, we can answer 2D RANGE REPORTING queries in $\mathcal{O}((1+k)\log n)$ time using $\mathcal{O}(Nn)$ space, where k is the number of points from $\mathcal{P}$ enclosed by the query rectangle.*

ACTIVE PREFIXES EXTENSION

For several problems on variable strings, we want to be able to process strings online, *segment by segment*. To do this, we need a way to extend *partial matches* with new sets that arrive online. This problem is formalized as follows:

Problem 1. ACTIVE PREFIXES EXTENSION (APE) [43]
Input: A string P of length m, a bit vector W of size m, and a set $\mathcal{S}$ of strings of total length N.
Output: A bit vector V of size m with $V[j] = 1$ if and only if there exists $S \in \mathcal{S}$ and $i \in [1..m]$, such that $P[1..i] \cdot S = P[1..j]$ and $W[i] = 1$.

Example 8. Let $m = 9$, $P = \texttt{ACAGGTCAGG}$, $W = \texttt{0110110010}$, and $\mathcal{S} = \{\texttt{GTCA}, \texttt{GGT}, \texttt{TCA}, \texttt{AGG}, \texttt{CA}\}$. Then, we have $V = 0000100100$.

Intuitively, the vector W represents the ending position of previous partial matches with P, and one wants to extend those partial matches with strings from $\mathcal{S}$. The vector returned by APE gives the ending positions of such partial matches.

A first approach is to solve APE is to search for occurrences in P of each string from $\mathcal{S}$, one by one, using an efficient pattern matching algorithm, and report a bit vector obtained from a conjunction of the starting position of those occurrences with W. In general, this would take $\mathcal{O}(N + m + |\mathbf{Occ}|)$, where **Occ** is the set of all the occurrences of a string from $\mathcal{S}$ in P. However, in the general setting, $|\mathbf{Occ}|$ can be as large as $\mathcal{O}(m^2)$ in general.

The problem was formalized by Grossi et al. [109], followed by a first improvement by Aoyama et al. [24]. Finally, Bernardini et al. proved the following [43]:

Lemma 9 ([43]). *The APE problem can be solved in $\widetilde{\mathcal{O}}(m^{\omega-1}) + \mathcal{O}(N)$ time, where ω is the matrix multiplication exponent*[2].

Surprisingly, the key improvement relies on algebraic techniques, namely fast matrix multiplication (FMM). This was also the case in [24], which used fast Fourier transform (FFT).

[2] In the current state of the art, the matrix multiplication exponent ω is at most 2.371553 [215].

1

CONDITIONAL LOWER BOUNDS

We introduce here the main tools that we use to prove lower bounds for several of the studied problems. They are based on two problems, namely *ORTHOGONAL VECTORS* (OV), and *BOOLEAN MATRIX MULTIPLICATION* (BMM).

Definition 10 (ORTHOGONAL VECTORS (OV))**.** Given two sets $X, Y \subseteq \{0,1\}^d$ such that $|X| = |Y| = n$ and $d = \omega(\log n)$, determine whether there exists $x \in X$ and $y \in Y$ such that x and y are orthogonal, namely, $x \cdot y = \sum_{i=1}^{d} x[i] \cdot y[i] = 0$.

The OV conjecture states the following:

Conjecture 11 (OV hypothesis [214] (OVH))**.** *No (deterministic or randomized) algorithm can solve OV on vector sets $X, Y \subseteq \{0,1\}^d$, $|X| = |Y| = n$, in time $\mathcal{O}(n^{2-\varepsilon}\text{poly}(d))$.*

In fact, the OV hypothesis is based on SETH [126], a popular conjecture in computer science, stating that the sequence s_k of smallest integers such that the k-SAT problem [3] can be solved in $2^{s_k n + o(n)}$ tends toward infinity as k increases. The following was shown by Williams:

Lemma 12 ([214])**.** *SETH implies* OVH*.*

Consider the APE problem. As described above, important improvements over the standard algorithm were made (Lemma 9), but they all rely on *non-combinatorial* techniques, namely, they make use of involved algebraic operations such as FMM or FFT. One natural question is the following: can one design a more efficient algorithm, using only combinatorial techniques? This idea is formalized in the following conjecture:

Conjecture 13 (BMM conjecture [7])**.** *There is no combinatorial algorithm to compute the product of two $D \times D$ Boolean matrices, working in $\mathcal{O}(D^{3-\varepsilon})$ time, for any constant $\varepsilon > 0$.*

Instead of reducing problems to BMM directly, the following problem is often used[4]:

Problem 2. TRIANGLE DETECTION (TD)
Input: Three $D \times D$ Boolean matrices A, B, C
Output: YES if there are three indices $i, j, k \in [1..D-1]$ such that $A[i,j] = B[j,k] = C[k,i] = 1$.

Indeed, the two problems are related as follows:

Lemma 14 ([208])**.** *Problems BMM and TD either both have truly sub-cubic combinatorial algorithms or none of them do.*

In the case of APE, Bernardini et al [43] showed that the existence of a combinatorial $\mathcal{O}(m^{1.5-\varepsilon} + N)$-time algorithm would contradict the BMM conjecture. In particular, an algorithm which is competitive with the algorithm from [43] cannot rely only on combinatorial operations, unless the BMM conjecture is false.

[3] The problem k-SAT asks, given a boolean expression that is written as a sequence of *k-clauses* (namely, disjunctions of k variables or their negation), separated by AND operators, whether there exist a truth assignment of each variable that satisfies the expression.

[4] This is a generalization of triangle detection in a graph, which can be seen by considering A, B, and C as adjacency matrices.

1.3 INVENTORY OF THE MAIN CONTRIBUTIONS

We give here a summary of the main results presented in this thesis.

1.3.1 PATTERN MATCHING

The Chapter 2 is dedicated to pattern matching. In Section 2.1, we study exact pattern matching for various types of variable strings. Formally, we define the following problem:

Problem 3. PATTERN MATCHING ON VARIABLE TEXT (PVART(X,Y))
Input: A solid or variable pattern P of type X and a variable text $\widetilde{T}$ of type Y
Output: All positions j such that a string in $\mathcal{L}(P)$ occurs in $\widetilde{T}$ ending at $\widetilde{T}[j]$

Where X and Y can be any of the string types we defined in Subsection 1.2.1.

In Section 2.1.3 and Section 2.1.4, we give two sub-quadratic upper bounds for PVART(X,Y) when X=SOLID (solid pattern) and Y is either a k-D string or a k-F graph:

Theorem 17. PVART(SOLID,k-D) can be solved in $\mathcal{O}(N+kn\log^2(\frac{m}{k}))=\mathcal{O}(N+N\log^2 m)$ time.

Theorem 18. PVART(SOLID,k-F) can be solved in $\mathcal{O}(\sqrt{m}(|E|+N\log^2 m))$ time.

When the segments of the text do not have a fixed number k of characters, a quadratic term appears in our time complexities, that remain sub-quadratic only when $N/n=\omega(1)$.

Theorem 19. PVART(SOLID,GD) can be solved in $\mathcal{O}(nm+N)$ time.

Theorem 20. PVART(SOLID,F) can be solved in $\mathcal{O}(nm+N+|E|\log m)$ time.

In Section 2.1.4.2, we show the following lower bounds:

Theorem 21. No algorithm can solve PVART(1-D,k-D) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Theorem 22. No algorithm can solve PVART(k-D,1-D) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Theorem 23. No algorithm can solve PVART(1-D,1-F) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Theorem 24. No algorithm can solve PVART(1-F,1-D) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Theorem 25. No algorithm can solve PVART(1-F,1-F) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Since a pattern of type 1-D is a special case of k-D, GD or ED (and k-D is a special case of GD or ED), and since 1-F is a special case of k-F, F, and EF, the lower bounds above propagate along the taxonomies for P and/or $\widetilde{T}$.

In Section 2.2, we focus on *approximate* pattern matching on *ED* text. In particular, we study the following problem, based on the edit distance:

Problem 4. 1-ERROR EDSM
Input: A string P of length m and an ED string $\widetilde{T}$ of length n and size N.
Output: All positions j' in $\widetilde{T}$ such that there is at least one string P' with an occurrence ending at position j' in $\widetilde{T}$, and with $d_E(P,P') \leq 1$ (reporting version); or YES if and only if there is at least one string P' with an occurrence in $\widetilde{T}$, and with $d_E(P,P') \leq 1$ (decision version).

We also define an alternative version, this time based on Hamming distance:

Problem 5. 1-MISMATCH EDSM
Input: A string P of length m and an ED string $\widetilde{T}$ of length n and size N.
Output: All positions j' in $\widetilde{T}$ such that there is at least one string P' with an occurrence ending at position j' in $\widetilde{T}$, and with $d_H(P,P') \leq 1$.

We show the following theorems:

Theorem 39. Given a pattern P of length m and an ED text $\widetilde{T}$ of length n and size N, the reporting version of 1-ERROR EDSM can be solved online in $\mathcal{O}(nm^2 \log m + N \log m)$ or $\mathcal{O}(nm^3 + N)$ time. The decision version of 1-ERROR EDSM can be solved off-line in $\mathcal{O}(nm^2 \sqrt{\log m} + N \log \log m)$ time.

Theorem 40. Given a pattern P of length m and an ED text $\widetilde{T}$ of length n and size N, 1-MISMATCH EDSM can be solved online in $\mathcal{O}(nm^2 + N \log m)$ or $\mathcal{O}(nm^3 + N)$ time.

1.3.2 COMPARISON

Chapter 3 is dedicated to comparison. The main problem we consider is the following:

Problem 6. ED STRING INTERSECTION (EDSI)
Input: Two *ED* strings, $\widetilde{T}_1$ of length n_1, cardinality B_1 and size N_1, and $\widetilde{T}_2$ of length n_2, cardinality B_2 and size N_2.
Output: YES if $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$ have a nonempty intersection, NO otherwise.

In Section 3.2, we give several conditional lower bounds. In particular, we show the following results:

Corollary 57. For any constant $\varepsilon > 0$, there exists no

- $\mathcal{O}((N_1 N_2)^{1-\varepsilon})$-time
- $\mathcal{O}((N_1 B_2 + N_2 B_1)^{1-\varepsilon})$-time
- $\mathcal{O}((N_1 n_2 + N_2 n_1)^{1-\varepsilon})$-time

algorithm for the EDSI problem, unless the OV conjecture is false.

Corollary 58. If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1+N_2)^{1.2-\varepsilon} f(n_1,n_2))$ time, for any constant $\varepsilon > 0$ and any function f, then there exists a truly subcubic combinatorial algorithm for TD.

In Section 3.3, we study EDSI in the special case where the alphabet contains a single letter. In that case, *ED* strings have a *compact representation* where every string in each segment is replaced by its length. We prove the following:

Theorem 61. If $\widetilde{T}_1$ and $\widetilde{T}_2$ are unary *ED* strings and each is given in a compact representation, the problem of deciding whether $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ is nonempty is NP-complete.

Theorem 64. If $\widetilde{T}_1$ and $\widetilde{T}_2$ are unary *ED* strings, then $L(\widetilde{T}_1) \cap L(\widetilde{T}_2)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ time.

In Section 3.4, we study the general case of EDSI, and introduce *intersection graphs*. We first give a combinatorial algorithm solving EDSI in $\mathcal{O}(N_1B_2 + N_2B_1)$ time, and then show a $\widetilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ time algorithm, based on FMM:

Theorem 69. EDSI can be solved in $\mathcal{O}(N_1B_2 + N_2B_1)$ time. If the answer is YES, a witness can be reported within the same time complexity.

Theorem 74. We can solve EDSI in $\widetilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ time, where ω is the matrix multiplication exponent. If the answer is YES, we can output a witness within the same time complexity.

In Section 3.5 and Section 3.6, we show several theoretical applications for the techniques that we develop. In particular, we show that intersection graphs can be used to compute *acronyms* (Theorem 76), compute a shortest/longest wintness of the intersection of $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$ (Lemma 78), compute the size of the intersection (Lemma 81), the longest common substring and subsequences (Lemma 84 and Lemma 85). We generalize *matching statistics* for *ED* strings and show that it can be computed using the intersection graph (Lemma 83). In Section 3.7, we show that the intersection graph can be used for the *approximate* version of EDSI:

Theorem 87. Given a pair $\widetilde{T}_1, \widetilde{T}_2$ of *ED* strings, we can find the pair $T_1 \in \mathcal{L}(\widetilde{T}_1), T_2 \in \mathcal{L}(\widetilde{T}_2)$ minimizing the distance $d_H(T_1,T_2)$ or $d_E(T_1,T_2)$ in $\mathcal{O}(N_1N_2)$ time.

Theorem 88. Given a pair $\widetilde{T}_1, \widetilde{T}_2$ of *ED* strings and an integer $k > 0$, we can check whether a pair $T_1 \in \mathcal{L}(\widetilde{T}_1), T_2 \in \mathcal{L}(\widetilde{T}_2)$ with $d_H(T_1,T_2) \leq k$ (resp. $d_E(T_1,T_2) \leq k$) exists in $\mathcal{O}(k(N_1B_2 + N_2B_1))$ time (resp. in $\mathcal{O}(k^2(N_1B_2 + N_2B_1))$ time) and, if that is the case, return the pair with the smallest distance.

In Section 3.8, we define *breakpoint matching statistics*, a restriction of matching statistics for practical pangenome comparison. We show that it can still be computed efficiently using intersection graphs (Theorem 90). Finally, in Section 3.9, we experimentally evaluate the efficiency of our algorithms.

1

1.3.3 INDEXING

In Chapter 4, we study indexing of a different type of variable string, namely *weighted strings*. Weighted strings model character-level uncertainty, and are defined as a sequence of probability vectors over a given alphabet Σ. The *occurence probability* of a pattern P at a certain position i in a weighted string X is obtained by multiplying for each j the probability of the letter $P[j]$ in $X[i+j-1]$. In general, we are given a probability threshold $\frac{1}{z}$ and we want to report occurences of a pattern only when the *occurence probability* is at least $\frac{1}{z}$. Our problem is formalized as follows:

Problem 7. ℓ-WEIGHTED INDEXING
Input: A weighted string X of length n over an alphabet Σ, a weight threshold $\frac{1}{z} \in (0,1]$, and an integer $\ell > 0$
Output: A *compact* data structure (the index) supporting *efficient* pattern matching queries; i.e., report all positions in X where P occurs with probability at least $\frac{1}{z}$.

Other than the *index size* and the *query time*, we seek to minimize the *construction time* and the *construction space*. We make the following three main contributions:

Theorem 106. Let X be a weighted string of length n over an alphabet of size σ, $\frac{1}{z}$ be a weight threshold, and $\ell > 0$ be an integer. After $\mathcal{O}(nz)$ construction time and using $\mathcal{O}(nz)$ construction space, we can construct an index of $\mathcal{O}(n + \frac{nz}{\ell}\log z)$ expected size answering ℓ-WEIGHTED INDEXING queries of length $m \geq \ell$ in $\mathcal{O}(m + (1 + \frac{nz}{\sigma^m})\log\frac{nz}{\ell})$ expected time.

Theorem 110. For any weighted string X of length n, any weight threshold $\frac{1}{z}$, and any integer $\ell > 0$, we can construct the minimizer solid factor tree in expected time $\mathcal{O}(nz\log\ell + \frac{nz}{\ell}\log\frac{nz}{\ell}\log z)$ and space $\mathcal{O}(n + \frac{nz}{\ell}\log z)$.

1

List of Publications

This thesis is based on the following publications:

- Chapter 2 is based on:

  [26] Rocco Ascone, Giulia Bernardini, Alessio Conte, Massimo Equi, Estéban Gabory, Roberto Grossi, and Nadia Pisanti. A Unifying Taxonomy of Pattern Matching in Degenerate Strings and Founder Graphs. In Solon P. Pissis and Wing-Kin Sung, editors, *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, volume 312 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:21, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik

  A journal extension of [26], to appear.

  [40] Giulia Bernardini, Estéban Gabory, Solon P. Pissis, Leen Stougie, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string matching with 1 error. In Armando Castañeda and Francisco Rodríguez-Henríquez, editors, *15th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 13568 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2022

- Chapter 3 is based on:

  [95] Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Comparing elastic-degenerate strings: Algorithms, lower bounds, and applications. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 259 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023

  [96] Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Pangenome comparison via ED strings. *Frontiers Bioinformatics*, 4 - 2024, 2024

  [98] Estéban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string comparison. *Information and Computation*, 304:105296, 2025

- Chapter 4 is based on:

  [94] Estéban Gabory, Chang Liu, Grigorios Loukides, Solon P. Pissis, and Wiktor Zuba. Space-efficient indexes for uncertain strings. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4828–4842, 2024.

Further set of publications not included in this thesis:

- [39] Giulia Bernardini, Alessio Conte, Estéban Gabory, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, Giulia Punzi, and Michelle Sweering. On Strings Having the Same Length- k Substrings. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, volume 223 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik

- [97] Estéban Gabory, Eric Rivals, Michelle Sweering, Hilde Verbeek, and Pengfei Wang. Periodicity of degenerate strings. In *Prague Stringology Conference 2023*, page 42, 2023

- [41] Giulia Bernardini, Estéban Gabory, Solon P. Pissis, Leen Stougie, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string matching with 1 error or mismatch. *Theory of Computing Systems*, Sep 2024

2 Pattern matching

2.1 Exact pattern matching

2.1.1 Introduction

In this section, we focus on several exact pattern-matching problems on variable texts. The section is entirely based on [26].

Problem 3. Pattern Matching on Variable Text (PvarT(X,Y))
Input: A solid or variable pattern P of type X and a variable text $\widetilde{T}$ of type Y
Output: All positions j such that a string in $\mathcal{L}(P)$ occurs in $\widetilde{T}$ ending at $\widetilde{T}[j]$

Where X and Y can be any of the string types we defined in Subsection 1.2.1.

Following the existing literature [109], the output of PvarT only specifies the ending segments of the occurrences, and gives no information about the specific position within the segment. Note that reporting the starting positions is an equivalent problem, up to text and pattern reversal. We say that PvarT has *constant alphabet* if the alphabet Σ of the strings in both the pattern and the text is of constant size.

For the problem of matching a solid pattern of size M into an ED or EF text of size N, it is known that an algorithm with complexity $O(M^{1-\varepsilon}N)$ or $O(MN^{1-\varepsilon})$ with $\varepsilon > 0$ would contradict SETH [32, 42, 43, 87, 106], and the contradiction holds even if such complexity is achieved at query time after a polynomial-time indexing step [84, 85, 105], while quadratic-time algorithms are known. Moreover, strongly sub-quadratic algorithms are known for PvarT(solid, 1-D):

Theorem 15 ([67])**.** PvarT(solid, 1-D) can be solved in $\mathcal{O}(n\log^2 m)$ time.

The case where both P and $\widetilde{T}$ are 1-D is well-studied in the literature. In [125], the definition of *indeterminate* string coincides with our definition of 1-D string and an $\mathcal{O}(n\log m)$-time algorithm for PvarT(1-D,1-D) for the case of constant-size alphabets is given [125, Lemma 17]. A similar algorithm for constant-size alphabets has also been proposed in [192].

Theorem 16 ([125])**.** PvarT(1-D, 1-D) can be solved in $\mathcal{O}(n\log m)$ time on a constant-size alphabet.

These results are complemented in [125, Theorem 22] with a quadratic conditional lower bound for the cases in which the alphabet size is not bounded by a constant.

In this section, we analyze the complexity of the problem, denoted $\text{PVART}(X,Y)$, of matching a pattern of type X in a text of type Y, where both X and Y can be any of the types of variable strings described above: for instance, $\text{PVART}(1\text{-}D, k\text{-}F)$ is the problem of finding occurrences of a 1-D pattern within a k-F graph.

As a consequence, our reference bound is quadratic: given two types X and Y of strings (variable or solid), either this is proved as a lower bound for $\text{PVART}(X,Y)$, or a better algorithm – an upper bound – should be exhibited. For the purpose of exhaustively performing this task for all types X of patterns, and all types Y of variable texts, our contribution is to complete Table 2.1, where columns correspond to the pattern and rows correspond to the text.

All our results are presented in Subsection 2.1.2 and summarized in Table 2.1. Note that, due to space constraints, in Table 2.1 we write $\Omega((MN)^{1-\varepsilon})$, for every $\varepsilon > 0$, to denote the quadratic lower bound, but in fact, all our lower bound results prove both bounds $\Omega(M^{1-\varepsilon}N)$ and $\Omega(MN^{1-\varepsilon})$ (for every $\varepsilon > 0$).

2.1.2 DEFINITIONS AND SUMMARY OF THE RESULTS

Here we present the results that we will prove in Sections 2.1.3 and 2.1.4, consisting of upper and lower bounds for the problem $\text{PVART}(X,Y)$ for several choices of pattern

Pattern \ Text	1-D	1-F	k-D	k-F	GD	F	ED	EF
SOLID	**Thm. 15** $\mathcal{O}(n\log^2 m)$	**Thm. 18** $\mathcal{O}(\sqrt{m}\cdot(\lvert E\rvert+N\cdot\log^2 m))$	**Thm. 17** $\mathcal{O}(N+N\cdot\log^2 m)$	**Thm. 18** $\mathcal{O}(\sqrt{m}\cdot(\lvert E\rvert+N\cdot\log^2 m))$	**Thm. 19**‡ $\mathcal{O}(nm+N)$	**Thm. 20**‡ $\mathcal{O}(nm+N+\lvert E\rvert\log m)$	[32] $\Omega((mN)^{1-\varepsilon})$	[87] $\Omega((m\lvert E\rvert)^{1-\varepsilon})$
1-D	**Thm. 16** $\mathcal{O}(n\log m)^{\dagger}$	**Thm. 23** $\Omega((MN)^{1-\varepsilon})$	**Thm. 21** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$	**Cor. 26** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$	**Cor. 26** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$
1-F	**Thm. 24** $\Omega((MN)^{1-\varepsilon})$	**Thm. 25** $\Omega((MN)^{1-\varepsilon})$	**Cor. 29** $\Omega((MN)^{1-\varepsilon})$	**Cor. 30** $\Omega((MN)^{1-\varepsilon})$	**Cor. 29** $\Omega((MN)^{1-\varepsilon})$	**Cor. 30** $\Omega((MN)^{1-\varepsilon})$	**Cor. 29** $\Omega((MN)^{1-\varepsilon})$	**Cor. 30** $\Omega((MN)^{1-\varepsilon})$
k-D	**Thm. 22** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$	**Cor. 26** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$	**Cor. 26** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$	**Cor. 26** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$
k-F, F,EF	**Cor. 29** $\Omega((MN)^{1-\varepsilon})$	**Cor. 30** $\Omega((MN)^{1-\varepsilon})$	**Cor. 29** $\Omega((MN)^{1-\varepsilon})$	**Cor. 30** $\Omega((MN)^{1-\varepsilon})$	**Cor. 29** $\Omega((MN)^{1-\varepsilon})$	**Cor. 30** $\Omega((MN)^{1-\varepsilon})$	**Cor. 29** $\Omega((MN)^{1-\varepsilon})$	**Cor. 30** $\Omega((MN)^{1-\varepsilon})$
GD,ED	**Cor. 27** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$	**Cor. 26** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$	**Cor. 26** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$	**Cor. 26** $\Omega((MN)^{1-\varepsilon})$	**Cor. 28** $\Omega((MN)^{1-\varepsilon})$

Table 2.1: Complexity chart for problem $\text{PVART}(X,Y)$, where X ranges over rows and Y over columns. Green cells are for truly sub-quadratic $\mathcal{O}(NM^{1-\varepsilon})$ (for some $\varepsilon > 0$) upper bounds, yellow cells are for sub-quadratic upper bounds under special conditions, and red cells are for a quadratic lower bound $\Omega((MN)^{1-\varepsilon})$ (for every $\varepsilon > 0$) conditioned on SETH, even with a constant-size alphabet (our bounds are even tighter, check the referred results for details). Capital M and N denote the total size of the pattern and the text, respectively, while m and n denote the respective *number of segments* (see Subsection 2.1.2). Note that the reduction for EF [87] implies also a $\Omega((mN)^{1-\varepsilon})$ bound. Symbol † indicates that the bound holds only for constant-size alphabets, as it has an exponential dependency on the alphabet size; symbol ‡ indicates that the bound is sub-quadratic whenever $N/n = \omega(1)$. The sub-quadratic upper bounds for $\text{PVART}(\text{SOLID},1\text{-}D)$, $\text{PVART}(1\text{-}D,1\text{-}D)$ and the quadratic lower bounds for $\text{PVART}(\text{SOLID},ED)$, $\text{PVART}(\text{SOLID},EF)$ were known in the literature; the other results are proven in this section.

type X and text type Y where: m and M are, respectively, the length and the size of the pattern; n, N are, respectively, the length and the size of the text; and finally, E refers to the edges of the text when this is a founder graph.

Upper bounds We give two sub-quadratic upper bounds for PVART(X,Y) when X=SOLID (solid pattern) and Y is either a k-D string or a k-F graph.

Theorem 17. PVART(SOLID,k-D) can be solved in $\mathcal{O}(N+kn\log^2(\frac{m}{k}))=\mathcal{O}(N+N\log^2 m)$ time.

Theorem 18. PVART(SOLID,k-F) can be solved in $\mathcal{O}(\sqrt{m}(|E|+N\log^2 m))$ time.

When the segments of the text do not have a fixed number k of characters, a quadratic term appears in our time complexities, that remain sub-quadratic only when $N/n=\omega(1)$.

Theorem 19. PVART(SOLID,GD) can be solved in $\mathcal{O}(nm+N)$ time.

Theorem 20. PVART(SOLID,F) can be solved in $\mathcal{O}(nm+N+|E|\log m)$ time.

Lower bounds When both the pattern and the text are variable strings, we show conditional lower bounds for PVART(X,Y), which all hold even for constant alphabets (except for PVART(1-D,1-D)). In Subsection 2.1.4.2 we discuss more in detail the case PVART(1-D,1-D), for which a quadratic conditional lower bound exists in the literature for non-constant alphabets.

Theorem 21. No algorithm can solve PVART(1-D,k-D) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Theorem 22. No algorithm can solve PVART(k-D,1-D) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Theorem 23. No algorithm can solve PVART(1-D,1-F) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Theorem 24. No algorithm can solve PVART(1-F,1-D) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Theorem 25. No algorithm can solve PVART(1-F,1-F) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon>0$, unless OVH is false.

Since a pattern of type 1-D is a special case of k-D, GD or ED (and k-D is a special case of GD or ED), and since 1-F is a special case of k-F, F, and EF, the lower bounds above propagate along the taxonomies for P and/or $\widetilde{T}$. Therefore, we have the following corollaries. The first one directly follows from Theorem 21.

Corollary 26. No algorithm can solve PVART(X,Y) for $X = 1$-D, k-D, GD, ED and $Y = k$-D, GD, ED on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

Corollary 27. No algorithm can solve PVART(X,1-D) for $X = k$-D, GD, ED on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

Corollary 27 above follows from Theorem 22, while Corollary 28, 29 and 30 below are from Theorems 23, 24 and 25, respectively.

Corollary 28. No algorithm can solve PVART(X,Y) for $X = 1$-D, k-D, GD, ED and $Y =$ 1-F, k-F, F, EF on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

Corollary 29. No algorithm can solve PVART(X,Y) for $X =$ 1-F, k-F, F, EF and $Y =$ 1-D, k-D, GD, ED on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

Corollary 30. No algorithm can solve PVART(X,Y) for $X =$ 1-F, k-F, F, EF and $Y =$ 1-F, k-F, F, EF on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

2.1.3 MATCHING A SOLID PATTERN IN *GD* AND *F*

In Theorem 19, we show that the online $\mathcal{O}(m^2n+N)$-time algorithm proposed in [109] for pattern matching in ED texts requires only $\mathcal{O}(mn+N)$ time when applied to GD texts.

Theorem 19. PVART(SOLID,GD) can be solved in $\mathcal{O}(nm+N)$ time.

Proof. Consider the pattern-matching algorithm of [109] for ED texts. The algorithm consists of reading the text $\widetilde{T}$ segment by segment, from left to right, after constructing the suffix tree $\mathrm{ST}(P)$ of the pattern P in a preprocessing step. Given a segment $\widetilde{T}[i]$ of width k_i, the 4 following sub-problems are solved: (i) If $k_i \geq |m|$, does P occur in a string from $\widetilde{T}[i]$? (*easy case*) (ii) Compute every suffix of a string $t \in \widetilde{T}[i]$ that matches a prefix of P (*suffix case*); (iii) Compute every prefix of a string $t \in \widetilde{T}[i]$ that matches a suffix of P (*prefix case*); (iv) Find the strings $t \in \widetilde{T}[i]$ that are factors of P (*anchor case*). All occurrences of P are then computed from the solutions to these four problems: it is straightforward to solve the easy and the suffix/prefix case in $\mathcal{O}(N+m)$ time using any standard pattern-matching algorithm (e.g. KMP [141]) and using $\mathrm{ST}(P)$, respectively (see [109] for details).

We now focus on the anchor case for a segment $\widetilde{T}[i]$, which is a special case of APE (Problem 1) and show that, in the case of GD texts, it can be solved in time $\mathcal{O}(N_i+m)$.

To solve APE it suffices to compute all the occurrences in P of all the strings from $\widetilde{T}[i]$ and check whether they extend some active prefixes W stored before. Since the strings in $\widetilde{T}[i]$ are all of the same length k_i, thus no two distinct such strings can occur at the same position in P, we have the following crucial observation.

Observation 31. *The cumulative number of occurrences in P of all the strings from a GD segment $\widetilde{T}[i]$ is bounded by m.*

Observation 31 implies that all starting positions of such occurrences can be computed and stored in an auxiliary bit-vector *OCC* of size m in $\mathcal{O}(N_i+m)$ time using $\mathrm{ST}(P)$. The output vector V can then be obtained by left-shifting *OCC* by one position, taking its bit-wise AND with W, and shifting the resulting vector by k_i positions to the right. Solving the anchor case for every segment thus takes $\mathcal{O}(N+nm)$ time. Note that the difference in the time complexities of APE for *GD* texts and *ED* texts is because, in the latter case, Observation 31 does not hold, and the number of occurrences of the strings from $\widetilde{T}[i]$ in P can only be bounded by m^2. □

We next adapt the algorithm of Theorem 19 to solve pattern matching on founder graphs.

Theorem 20. PVART(SOLID,F) can be solved in $\mathcal{O}(nm+N+|E|\log m)$ time.

Proof. Let $G=(\widetilde{T},E)$ be a founder graph of length n, size N and cardinality B, and let P be a solid pattern of length m. We denote by $\widetilde{T}[i][j]$ the j-th string of $\widetilde{T}[i]$, $j\in[1\,..\,B_i]$, assuming any fixed order; an edge $(j,j')\in E_{i-1}$ thus connects string $\widetilde{T}[i-1][j]$ to $\widetilde{T}[i][j']$. We process separately the occurrences of P that span only one segment (easy case), only two segments, or more. The easy case is trivially solved in $\mathcal{O}(m+N)$ time using any linear-time pattern-matching algorithm. Let us now focus on the other two cases.

Analogously to the suffix and prefix subproblems listed in the proof of Theorem 19, we precompute all proper suffix/prefix and prefix/suffix overlaps between P and each string from each segment, i.e., for each string, we compute the length of all suffixes that are equal to prefixes of P, and of all prefixes that are equal to suffixes of P. While in the case of *GD* text, it suffices to store the length of all such overlaps cumulatively for each segment, in the case of founder graphs, due to the presence of edges, we need to retain this information separately for each string in each segment. We thus compute two binary arrays $b_{i,j}$ and $e_{i,j}$ for each string $\widetilde{T}[i][j]$, each of the same length k_i, s.t. $b_{i,j}[\ell]=1$ if and only if $\widetilde{T}[i][j]$ has a suffix/prefix overlap of length ℓ with P, and $e_{i,j}[\ell]=1$ if and only if $\widetilde{T}[i][j]$ has a prefix/suffix overlap of length ℓ with P. All such arrays can be constructed in $\mathcal{O}(m+N)$ total time using e.g. the suffix trees of P and of its reversal, and occupy total space $\mathcal{O}(N)$.

Occurrences spanning only two segments. We consider all pairs of consecutive segments $\widetilde{T}[i],\widetilde{T}[i+1]$ such that $k_i+k_{i+1}\geq m$ (the only candidates for occurrences of this kind). For each edge $(j,j')\in E_i$, let p_j be the length of the longest suffix of $\widetilde{T}[i][j]$ that overlaps a prefix of P, i.e. the largest index of $b_{i,j}$ set to 1, and let $s_{j'}$ be the length of the longest prefix of $\widetilde{T}[i+1][j']$ that overlaps a suffix of P, i.e. the largest index of $e_{i+1,j'}$ set to 1. The following observation characterises the occurrences of P spanning $\widetilde{T}[i],\widetilde{T}[i+1]$.

Observation 32. *P has an occurrence spanning $\widetilde{T}[i], \widetilde{T}[i+1]$ iff there exists an edge $(j, j') \in E_i$ such that P occurs in the concatenation of the suffix of length p_j of $\widetilde{T}[i][j]$ and the prefix of length $s_{j'}$ of $\widetilde{T}[i+1][j']$, which are equal, respectively, to $P[1\,..\,p_j]$ and $P[m-s_{j'}+1\,..\,m]$.*

The main tool we use to spot these occurrences is the *border tree* [110] of P, a data structure that, given any pair of positions $1 \leq s \leq p \leq m$, returns the set **Occ** of occurrences of P in the concatenation $P[1\,..\,p]P[s\,..\,m]$ of its prefix of length p and suffix of length $m-s+1$. The border tree can be constructed in $\mathcal{O}(m)$ time and answers queries in $\mathcal{O}(\log m + |\mathbf{Occ}|)$ time [110]. After constructing the border tree of P, we thus process each pair $\widetilde{T}[i], \widetilde{T}[i+1]$ such that $k_i + k_{i+1} \geq m$, and apply the following algorithm, whose correctness follows from Observation 32: for each $(j, j') \in E_i$, query the border tree of P with indices $p_j, m - s_{j'} + 1$. If the returned set **Occ** is nonempty, break and return an occurrence of P in G ending at position $i+1$. Each such query takes $\mathcal{O}(\log m + |\mathbf{Occ}|)$ time [110], and $|\mathbf{Occ}| \leq m$ for a given query. Since we stop asking queries as soon as we find a nonempty set of occurrences, the total time for $\widetilde{T}[i], \widetilde{T}[i+1]$ is $\mathcal{O}(|E_i| \log m + m)$, implying time $\mathcal{O}(|E| \log m + mn)$ to process the whole G.

Occurrences spanning at least three segments. This case is analogous to the anchor case of Theorem 19, and can only happen when the second of the (at least three) segments has a width smaller than m. We process the segments of G from left to right and maintain an array V of size m that keeps track of the prefixes of P that match up to a certain segment (*partial occurrences*): however, now V is an array of integers, rather than a simple bit-vector, and it only keeps track of partial occurrences that span at least one full segment (thus excluding prefixes of P that match a proper suffix of some string from some segment: these shorter partial occurrences will be treated differently, as we explain in the following).

Let V_{i-1} denote the state of V after processing a segment $\widetilde{T}[i-1]$ ($V_{i-1} = 0^m$ if $k_{i-1} > m$). When processing segment $\widetilde{T}[i]$ (assuming $k_i < m$), our task is to compute the next state V_i s.t. $V_i[\ell] = j$ iff $P[1\,..\,\ell]$ is a suffix of some string from $\mathcal{L}(G[1\,..\,i])$ that ends with the whole string $\widetilde{T}[i][j]$, and $V_i[\ell] = 0$ otherwise. Note that the values of V_i are uniquely defined: see Observation 33. In particular, this implies that the first $k_i - 1$ positions of V_i, corresponding to partial occurrences that do not contain an entire string from $\widetilde{T}[i]$, are always 0. Further, note that positions between k_i and $\min\{k_i + k_{i-1} - 1, m\}$ of V_i correspond to partial occurrences of P that contain an entire string from $\widetilde{T}[i]$ but not from $\widetilde{T}[i-1]$. We compute $V_i[k_i\,..\,k_i + k_{i-1} - 1]$ and $V_i[k_i + k_{i-1}\,..\,m]$ using two different procedures (the second sub-array is empty in the case $k_i + k_{i-1} > m$).

To process $\widetilde{T}[i]$, we first compute an array OCC of size m such that $OCC[\ell] = j$ iff $\widetilde{T}[i][j]$ occurs in P starting at position ℓ; by Observation 31, OCC is well defined and can be computed in $\mathcal{O}(N_i + m)$ time using the suffix tree $\mathrm{ST}(P)$. Note that OCC contains all potential extensions of partial occurrences of P with whole strings from $\widetilde{T}[i]$. We use the following crucial observation, which is a direct consequence of Observation 31.

Observation 33. *Any partial occurrence of P ending at $\widetilde{T}[i-1]$ can be extended by at most one string from $\widetilde{T}[i]$.*

In particular, the partial occurrence represented by $V_{i-1}[\ell] = j$ can only be extended by the unique string from $\widetilde{T}[i]$ occurring in P at position $\ell+1$, if any. This implies that it suffices to check the following necessary and sufficient conditions to compute $V_i[k_i + k_{i-1} \,..\, m]$: for each position $\ell \in [k_i + k_{i-1}, m]$, we set $V_i[\ell] = j'$ iff $OCC[\ell - k_i + 1] = j'$, $V_{i-1}[\ell - k_i] = j$ and $(j, j') \in E_{i-1}$. To verify these conditions, collect all triplets (j, j', ℓ) such that $V_{i-1}[\ell - k_i] = j$ and $OCC[\ell - k_i + 1] = j'$, for all $\ell \in [k_i + k_{i-1}, m]$, and sort them lexicographically together with all the pairs from E_i, using radix sort [71]. Then, scan the resulting sorted list and set $V_i[\ell] = j'$ iff a triplet (j, j', ℓ) is immediately preceded by the pair (j, j'). Since there is at most one triplet for each value of ℓ, this requires $\mathcal{O}(m + |E_i|)$ total time per segment and thus $\mathcal{O}(nm + |E|)$ time over the whole G.

Let us now focus on computing $V_i[k_i \,..\, k_i + k_{i-1} - 1]$. This portion of V_i corresponds to partial occurrences of P matching a proper suffix of some string from $\widetilde{T}[i-1]$ and extended with an occurrence of some string from $\widetilde{T}[i]$ stored in $OCC[2 \,..\, k_{i-1}]$. Note that we cannot use the same technique as for $V_i[k_i + k_{i-1} \,..\, m]$, because two strings from $\widetilde{T}[i-1]$ can have equal suffixes. Instead, we scan $OCC[2 \,..\, k_{i-1}]$: if $OCC[\ell] = j' \neq 0$, we check whether $b_{i-1,j}[\ell - 1] = 1$ for all $(j, j') \in E_{i-1}$, and set $V_i[\ell + k_i - 1] = j'$ if this is the case. In other words, we check, for each occurrence of $\widetilde{T}[i][j']$ starting at $P[\ell]$, whether the suffix of length $\ell - 1$ of some of the strings from $\widetilde{T}[i-1]$ connected to $\widetilde{T}[i][j']$ is equal to $P[1 \,..\, \ell - 1]$. This procedure has a total cost $\mathcal{O}(N_{i-1} + m)$ because each position of each $b_{i-1,j}$ is accessed at most once; and each time we read an edge, we access exactly one position of one array b, thus we read at most N_{i-1} edges. This implies a total time $\mathcal{O}(N)$ over all segments.

Finally, to check whether some partial occurrence represented by $V_{i-1}[\ell] = j$ for $\ell \in [m - k_i + 2 \,..\, m]$ can be extended to a full occurrence of P with a proper prefix of some string from $\widetilde{T}[i]$, it suffices to check, for each edge $(j, j') \in E_{i-1}$, whether $e_{i,j'}[m - \ell + 1] = 1$. This requires $\mathcal{O}(N + mn)$ total time because each position of each array e is accessed at most once.

We obtain a total time complexity of $\mathcal{O}(N + mn + |E| \log m)$. □

2.1.4 PATTERN MATCHING IN *k-D* AND *k-F* TEXTS

In this section, we investigate the complexity of the pattern-matching problem in cases where the text is either a *k-D* string or a *k-F* graph. In Subsection 2.1.4.1 we consider the cases where the pattern is solid; in Subsection 2.1.4.2, the pattern is a *k-D* string or a *k-F* graph.

SOLID PATTERN

The problem of finding all the occurrences of a solid pattern of length m in a 1-*D* string of length $n > m$ and size N can be seen as a special case of PVART(1-*D*,1-*D*),

(known as the *SUBSET MATCHING* problem, defined by Cole and Hariharan [65]), for which an $\mathcal{O}(n\log^2 m)$-time deterministic algorithm exists [67][1].

In this section, we leverage this result to prove sub-quadratic upper bounds for PVART(SOLID,1-*F*), PVART(SOLID,*k*-*D*) and PVART(SOLID,*k*-*F*) by reducing each of these problems to several instances of PVART(SOLID,1-*D*).

Theorem 17. PVART(SOLID,*k*-*D*) can be solved in $\mathcal{O}(N+kn\log^2(\frac{m}{k})) = \mathcal{O}(N+N\log^2 m)$ time.

Proof. Let P be a solid pattern of length m and $\widetilde{T}$ a *k*-*D* string of length n, cardinality B and total size N. In a preprocessing step, we compute all suffix/prefix and prefix/-suffix overlaps of P and each string in the union of all segments of $\widetilde{T}$. More precisely, for each segment $\widetilde{T}[i]$ we compute two binary arrays b_i and e_i of length k such that $b_i[j] = 1$ if and only if there is a suffix/prefix overlap of length j between one of the strings in $\widetilde{T}[i]$ and P, and $e_i[j] = 1$ if and only if there is a prefix/suffix overlap of length j between a string in $\widetilde{T}[i]$ and P. The arrays b_i and e_i occupy $\mathcal{O}(kn) = \mathcal{O}(N)$ words of space and can be computed in $\mathcal{O}(N)$ time by e.g. building the generalized suffix tree of all the strings in all segments of $\widetilde{T}$.

Consider the case $m \geq 2k-1$, which implies that each occurrence of P contains at least 1 full string from some $\widetilde{T}[i]$. Let P_k denote the set of length-k substrings of P and let $h(s)$ denote the lexicographic rank of $s \in P_k$, to be used as a unique ID for s. The values $h(s)$ can be computed and stored in $\mathcal{O}(m)$ time and space by constructing the suffix tree of P and annotating the nodes at string depth k (possibly making them explicit) with their rank, obtained with a lexicographic traversal of the tree.

Using these values, we construct k instances of PVART(SOLID,1-*D*), one for each possible starting offset of occurrences of P in $\widetilde{T}$. The pattern $P^{(\ell)}$ of the ℓ-th instance, $\ell \in [0, k-1]$, is obtained from P by replacing each non-overlapping fragment of length k by its ID, starting from position $\ell+1$ and ignoring the possible remaining suffix of length $m-\ell-k\lfloor\frac{m-\ell}{k}\rfloor$. The length of $P^{(\ell)}$ is thus $\lfloor\frac{m-\ell}{k}\rfloor$, and they collectively occupy $\mathcal{O}(m)$ space. The text $\widetilde{T}^{(\ell)}$ of the ℓ-th instance is obtained from $\widetilde{T}$ by replacing each string $s \in \widetilde{T}[i]$, $\forall\, i \in [1,n]$, by its ID, as follows. If $s \in P_k$ and $h(s)$ occurs in $P^{(\ell)}$, then s is replaced by $h(s)$. Otherwise, thus if $h(s)$ does not occur in $P^{(\ell)}$ or s does not occur in P, s is discarded[2]. Each $\widetilde{T}^{(\ell)}$ has length n and total size $\mathcal{O}(\min\{B, n\frac{m}{k}\})$, therefore they collectively occupy $\mathcal{O}(kB) = \mathcal{O}(N)$ space. See Figure 2.1 for an example of this construction.

The k patterns can be constructed all at the same time by traversing the suffix tree of P in $\mathcal{O}(m)$ total time. The k texts can be constructed simultaneously by using the suffix tree of P, further annotated as follows. At each node at string depth k (which are all and only the nodes annotated with some ID $h(s)$), we store the list of values ℓ

[1] The upper bound explicitly proved by the authors in the cited paper is $\mathcal{O}(N\log^2 N)$; however, some observations made by the same authors in [65, Section 4] that lead to better bounds apply, and indeed the authors state that SUBSET MATCHING can be solved deterministically in $\mathcal{O}(n\log^2 m)$ time both in the abstract of [67] and in their later work [68].

[2] Note that, applying this procedure, some of the segments of $\widetilde{T}^{(\ell)}$ might be empty. To avoid so, one can use a special symbol \$ different from all the IDs $h(s)$ and place it in the otherwise empty segments of $\widetilde{T}^{(\ell)}$.

such that $\ell = i \mod k$ for some leaf i descending from the node. Since these nodes partition the leaves, and the number of descending leaves bounds the size of each list, such lists collectively occupy $\mathcal{O}(m)$ space and can be computed in $\mathcal{O}(m)$ space with a single tree traversal. Armed with this annotated suffix tree, to construct the k texts we simply search each string of $\widetilde{T}$ in the tree and write the value $h(s)$ (if any) in the corresponding segment of all texts $\widetilde{T}^{(\ell)}$ such that ℓ is in the list of the reached node. This procedure thus requires $\mathcal{O}(N)$ total time.

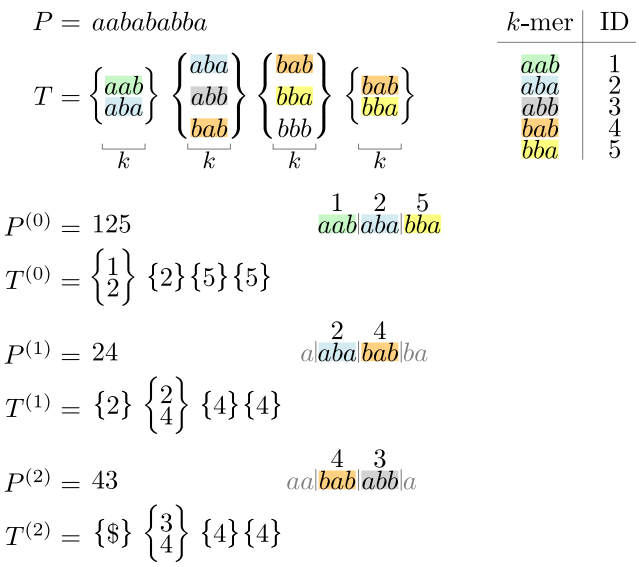


Figure 2.1: Illustration of the reduction described in the proof of Theorem 17. Note that P occurs twice in T, ending at $T[3]$ with starting offset 0, and ending at $T[4]$ with starting offset 2. The first occurrence corresponds to an occurrence of $P^{(0)}$ in $T^{(0)}$ ending at $T^{(0)}[3]$. The second can be retrieved by the occurrence of $P^{(1)}$ in $T^{(1)}$ ending at $T^{(1)}[3]$ using the pre-computed arrays $b_1 = [1,0,1]$ and $e_4 = [0,1,0]$: indeed, $b_1[1] = 1$ and $e_4[2] = 1$, indicating that there is a suffix-prefix overlap of some string from $T[1]$ and P of length 1 and a prefix-suffix overlap of some string from $T[4]$ and P, respectively, which complete the occurrence.

We now show that each occurrence of P in $\widetilde{T}$ (that fully contains at least a string from a segment of $\widetilde{T}$) corresponds to an occurrence of some $P^{(\ell)}$ in $\widetilde{T}'^{(\ell)}$ for some $\ell \in [0..k-1]$. The key observation is that if an occurrence of P in $\widetilde{T}$ starts at offset $q = k - \ell + 1$ in $\widetilde{T}[i]$ and ends at offset r in $\widetilde{T}[j]$, then a string from each of the segments $\widetilde{T}[i+1], \ldots, \widetilde{T}[j-1]$ occurs consecutively in P starting from position $\ell+1$; a prefix of length ℓ of P must match a suffix of some string in $\widetilde{T}[i]$; and a suffix of length r must match a prefix of some string in $\widetilde{T}[j]$. This implies, by construction, an occurrence of $P^{(\ell)}$ in $\widetilde{T}'^{(\ell)}$ starting at position $i+1$; moreover, it must be $b_i[\ell] = e_j[r] = 1$. This gives us the following algorithm. For each possible offset $\ell = 0, 1, \ldots, k-1$:

1. Find the occurrences of $P^{(\ell)}$ in $\widetilde{T}'^{(\ell)}$

2. For each occurrence $\widetilde{T}'^{(\ell)}[i\,..\,j]$ (note that $j = i-1+\lfloor\frac{m-\ell}{k}\rfloor$), check if it corresponds to an occurrence of P in $\widetilde{T}$ by checking whether $b_{i-1}[\ell] = e_{j+1}[r] = 1$ and report an occurrence $\widetilde{T}[i-1\,..\,j+1]$ if this condition holds.

Note that when $\ell = 0$ we only need to check whether $e_{j+1}[r] = 1$ and the corresponding occurrence is $\widetilde{T}[i\,..\,j+1]$; and symmetrically, if $r = 0$, we only check if $b_{i-1}[\ell] = 1$, the occurrence being $\widetilde{T}[i-1\,..\,j]$. For a fixed ℓ, Step (1) can be done in $\mathcal{O}(n\log^2(\frac{m}{k}))$ time using Theorem 15; and Step (2) requires $\mathcal{O}(1)$ time per occurrence. Since each $P^{(\ell)}$ can occur in at most n positions, the total time for step (2) over all $\ell = 0,1,\ldots,k-1$ is $\mathcal{O}(kn) = \mathcal{O}(N)$; and the total time for Step (1) is $k\cdot\mathcal{O}(n\log^2(\frac{m}{k}))$.

Finally, for the cases $k < 2k-1$, we can find all the occurrences of P in $\widetilde{T}$ that do not fully contain a string from some segment in $\mathcal{O}(N)$ total time. These occurrences either (i) span exactly two consecutive segments of $\widetilde{T}$, or (ii) are entirely contained in some string of some segment. To find all occurrences of type (ii) in $\mathcal{O}(N)$ time it suffices to run e.g. KMP [141]. To find the occurrences of type (i), we scan each array b_i: for each $j \in [1\,..\,k]$ s.t. $b_i[j] = 1$, we check whether $e_{i+1}[m-j] = 1$ and report an occurrence if this is the case. This requires $\mathcal{O}(kn) = \mathcal{O}(N)$ total time for all $i \in [1\,..\,n-1]$.

Overall, the preprocessing (constructing arrays b_i and e_i, computing the IDs of the length k substrings of P, and replacing them accordingly in $\widetilde{T}$) takes $\mathcal{O}(N)$ time, and finding all the matches takes $\mathcal{O}(N + kn\log^2\frac{m}{k})$, hence we obtain the claimed result. □

Let us now consider the case where the text is a *k-F* graph. Let P be a solid string of length m over an alphabet Σ, let $\ell < m$ be an integer which divides m, and let us write $P = P_1\,..\,P_d$ where $|P_i| = \ell$ for $i = 1\,..\,d$. We define the *ℓth spread* of P as the string $\mathbf{spr}^\ell(P) = \prod_{i=1}^{d-1} P_i\cdot\$\cdot P_{i+1}$, where $\$\notin\Sigma$. In other words, each fragment P_i of length ℓ is repeated, separated from the other fragments by a gadget letter \$, except for the first and last one, which are repeated once only.

Example 34. Let $P = \mathrm{GTTCGTTATATG}$. One has $\mathbf{spr}^3(P) = \mathrm{GTT}\cdot\$\cdot\mathrm{CGT\ CGT}\cdot\$\cdot\mathrm{TAT\ TAT}\cdot\$\cdot\mathrm{ATG}$.

Let $P' = \mathrm{CGTATTATT}$. One has $\mathbf{spr}^3(P') = \mathrm{CGT}\cdot\$\cdot\mathrm{ATTATT}\cdot\$\cdot\mathrm{ATT}$

Given a *k-F* graph $G = (\widetilde{T}, \bigcup_{i=1}^n E_i)$ of length n, we define its *disentanglement* as the $(2k+1)$-D string $\mathcal{D}(G) = D_1\,..\,D_{n-1}$ where for every $i = 1\,..\,n-1$, the set D_i contains every string $t\cdot\$\cdot t'$ such that $(t,t') \in E_i$. An example is shown in Figure 2.2

We prove the following lemma:

Lemma 35. *Given a k-F graph $G = (\widetilde{T}, E)$ and a solid string P of length $m > k$ such that k divides m, the string $\mathbf{spr}^k(P)$ occurs in $\mathcal{D}(G)$ if and only if there exists i, j with $P \in \mathcal{L}(G[i\,..\,j])$, namely P occurs in G with starting and ending offset 0.*

Proof. If $P \in \mathcal{L}(G[i\,..\,j])$ and $P = P_1\,..\,P_d$ is a factorisation of P in substrings of length k, then for every $\ell = 1\,..\,d-1$ one has $P_\ell \in \widetilde{T}[i+\ell-1]$, $P_{\ell+1} \in \widetilde{T}[i+\ell]$ and $(P_\ell, P_{\ell+1}) \in E_{i+\ell-1}$, which means that the set $\mathcal{D}(G)[i+\ell-1]$ contains the string $P_\ell\cdot\$\cdot P_{\ell+1}$, and the concatenation of those strings for each ℓ is equal to $\mathbf{spr}^k(P)$. Conversely, observe that

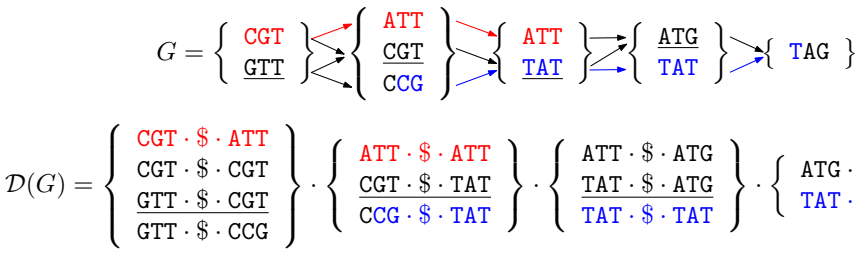


Figure 2.2: A k-F graph G (top) and its disentanglement $\mathcal{D}(G)$ (bottom). Pattern $P = \mathrm{GTTCGTTATATG}$ occurs once in G (underlined), and pattern $\mathbf{spr}^3(P) = \mathrm{GTT \cdot \$ \cdot CGT\ CGT \cdot \$ \cdot TAT\ TAT \cdot \$ \cdot ATG}$ occurs once in $\mathcal{D}(G)$ (also underlined). Pattern $P = \mathrm{CGTATTATT}$ occurs twice in G (in red and blue respectively), and both occurrences be detected in $\mathcal{D}(G)$ as in Theorem 18, from the strings highlighted in red and blue respectively.

for every $\ell = 1..d-1$, one has $\mathbf{spr}^k(P)[(\ell-1)(2k+1)+k+1] = \$$. If $\mathbf{spr}^k(P)$ occurs in $\mathcal{D}(G)$, since by construction the letter \$ occurs only at position $k+1$ of each set in $\mathcal{D}(G)$, the occurrence has to start at the first position of a set, and since k divides m that means that $\mathbf{spr}^k(P) \in \mathcal{L}(\mathcal{D}(G)[i..j])$ for some $1 \le i \le j \le n$. This implies that for such i, j and for each $0 \le \ell \le d-1$ one has $\mathbf{spr}^k(P)[(2k+1)(\ell-1)..(2k+1)\ell] \in \mathcal{D}(G)[i+\ell-1]$. But one notices that $\mathbf{spr}^k(P)[(2k+1)(\ell-1)..(2k+1)\ell] = P_\ell \$ P_{\ell+1}$, which means that $P_\ell \in \widetilde{T}[i+\ell-1]$, $P_{\ell+1} \in \widetilde{T}[i+\ell]$, and $(P_\ell, P_{\ell+1}) \in E_{i+\ell-1}$. Since this holds for every $1 \le \ell \le d-1$, we deduce that $P \in \mathcal{L}(G[i..j])$. □

Example 36. Let $P = \mathrm{GTTCGTTATATG}$, and $P' = \mathrm{CGTATTATT}$ as in Example 34, and let $G = (\widetilde{T}, E)$ and $\mathcal{D}(G)$ be as in Figure 2.2. The pattern P occurs once in G, ending in the fourth segment (underlined occurrence on the figure). This corresponds to a unique occurrence of $\mathbf{spr}^3(P) = \mathrm{GTT \cdot \$ \cdot CGTCGT \cdot \$ \cdot TATTAT \cdot \$ \cdot ATG}$ in $\mathcal{D}(G)$, ending in the third segment. The string P' occurs twice in G (in red and in blue on the figure). However, the string $\mathbf{spr}^3(P') = \mathrm{CGT \cdot \$ \cdot ATTATT \cdot \$ \cdot ATT}$ occurs only once in $\mathcal{D}(G)$ (in red). This is because the blue occurrence has nonzero starting and ending offsets. It can, however, be detected in $\mathcal{D}(G)$, by reading the string $\mathrm{CG \cdot \$ \cdot TATTAT \cdot \$ \cdot TATTAT \cdot \$ \cdot T}$ (also in blue on the figure), as explained in the following.

As illustrated in the above example, Lemma 35 allows us, given a k-F G and a pattern P whose length is a multiple of k, to use a $(2k+1)$-D string to detect the occurrences of P in G having starting and ending offset 0. We now extend this result to arbitrary pattern length and starting position, in order to prove Theorem 18.

Theorem 18. PVART(SOLID,k-F) can be solved in $\mathcal{O}(\sqrt{m}(|E| + N\log^2 m))$ time.

Proof. Let $G = (\widetilde{T}, E)$ be a k-F graph of length n and $P = P[1..m]$ be a solid pattern. Let us first assume that $k < \sqrt{m}$. We construct the disentanglement of G, which is a $(2k+1)$-D string $\mathcal{D}(G)$. Let us assume that P occurs in G starting at position i and ending at position j (we have $i < j$ since $k < \sqrt{m}$). This means, by definition, that there exist two (possibly empty) strings s, t and $P_1, \ldots, P_{j-i+1}$ such that $s \cdot P_1 \in \widetilde{T}[i]$, $P_{j-i+1} \cdot t \in \widetilde{T}[j]$ and $P_{\ell-i+1} \in \widetilde{T}[\ell]$ for every $i < \ell < j$, and such that $P_1 \cdots P_{j-i+1} = P$.

We notice that one necessarily has $|s|+m+|t| = 0 \mod k$, hence the string $P_1 \cdot \$ \cdot P_2 \cdot \mathbf{spr}^k(P_2..P_{j-i}) \cdot P_{j-i} \cdot \$ \cdot P_{j-i+1}$ is uniquely determined by $|s|$, and we call it $\mathbf{spr}^k_{|s|}(P)$. Writing $\widehat{P} = s \cdot P \cdot t$, we have that $\widehat{P} \in \mathcal{L}(G[i..j])$, which, by Lemma 35, means precisely that $\mathbf{spr}^k(\widehat{P})$ occurs in $\mathcal{D}(G)$. We can rewrite $\mathbf{spr}^k(\widehat{P}) = s \cdot \mathbf{spr}^k_{|s|}(P) \cdot t$, and we deduce that P occurs in G with starting offset $|s|$ in some segment if and only if $\mathbf{spr}^k_{|s|}(P)$ occurs in $\mathcal{D}(G)$ (observing the occurrences of \$ in the constructed strings, every occurrence of $\mathbf{spr}^k_{|s|}(P)$ is always part of an occurrence of $s \cdot \mathbf{spr}^k_{|s|}(P) \cdot t$, for some pair s, t having suitable lengths). To find every occurrence of P in G, we can then search for the k patterns $\mathbf{spr}^k_0(P), .., \mathbf{spr}^k_{k-1}(P)$ in $\mathcal{D}(G)$.

Notice that $\mathcal{D}(G)$ has size $\mathcal{O}(k|E|)$. To search for $\mathbf{spr}^k_0(P), .., \mathbf{spr}^k_{k-1}(P)$ we can apply a modified version of the algorithm from Theorem 17: while in Theorem 17, we need k searches for each pattern (one for each offset), here we are already given k distinct patterns that we each need to search with a known given offset. Hence, we can first compute IDs only for the length $2k+1$ substrings of $\mathbf{spr}^k_0(P), .., \mathbf{spr}^k_{k-1}(P)$ that contain a \$ at their $k+1$th position (as they are the only ones that can match a full string from a segment of $\mathcal{D}(G)$). This can be done simultaneously for all strings in $\mathcal{O}(m)$, by using the suffix tree of P, as such strings are constructed from a $2k$ length substring of P, with an extra central dollar. We then construct arrays e_i and b_i in $\mathcal{O}(k|E|)$ time, as in Theorem 17. Since each of the patterns are searched with a unique offset, the search takes $\mathcal{O}(n \log^2(\frac{m}{k}))$ for each of them. We obtain a total time of $\mathcal{O}(k|E|+kn\log^2(\frac{m}{k})) = \mathcal{O}(\sqrt{m}|E|+N\log^2 m)$ for the cases $k < \sqrt{m}$.

Now consider the case $k \geq \sqrt{m}$. Observe that, in this case, we have $n \leq \frac{N}{\sqrt{m}}$, because it always holds that $n \leq \frac{N}{k}$. By simply applying the algorithm of Theorem 20 in this special case we obtain a time complexity in $\mathcal{O}(N\sqrt{m}+|E|\log m)$. Combining the two cases, the total time becomes $\mathcal{O}(\sqrt{m}(|E|+N\log^2 m))$. □

Example 37. Consider again our running example $P' = \mathrm{CGTATTATT}$, as in Example 34. We can now detect its blue occurrence in G (as in Figure 2.2) by searching for $\mathbf{spr}^3_1(P') = \mathrm{CG} \cdot \$ \cdot \mathrm{TATTAT} \cdot \$ \cdot \mathrm{TATTAT} \cdot \$ \cdot \mathrm{T}$, that occurs in $\mathcal{D}(G)$ (in blue on Figure 2.2).

VARIABLE PATTERN

In this section, we study the complexity of finding all the occurrences of non-solid patterns in a *k-D* or *k-F* text.

Pattern 1-*D* and Text 1-*D* In [125], a $\mathcal{O}(n \log m)$-time algorithm is given for PVART(1-*D*, 1-*D*) when the alphabet size is constant - as we described in Theorem 16. Moreover, a quadratic conditional lower bounds is shown when the alphabet size is not bounded by a constant.

This lower bound clearly applies also to both PVART(1-*D*,*k*-*D*), PVART(*k*-*D*,1-*D*) when the alphabet size is not constant. In Subsection 2.1.4.2, we prove that, in these cases, a quadratic lower bound holds even when the alphabet has only three letters.

In Subsection 2.1.4.2, we consider the pattern-matching problem where at least one between the pattern and the text is in 1-*F* and the other is in 1-*D*. In every

possible case, i.e. PVART(1-*D*,1-*F*), PVART(1-*F*,1-*D*), PVART(1-*F*,1-*F*), we prove a quadratic conditional lower bound for constant-size alphabets. This implies that whenever we are considering the PVART(X,Y) problem with variable patterns, if at least one between pattern and text is a founder, the best that we can hope to achieve is a quadratic-time algorithm.

All the conditional lower bounds here rely on the Orthogonal Vectors Hypothesis (Conjecture 11).

Throughout this chapter, we will use the instance $X = \{x_1 = 010, x_2 = 100, x_3 = 011\}$, $Y = \{y_1 = 001, y_2 = 010, y_3 = 110\}$ of OV as our running example. Note that $x_2 = 100$, $y_2 = 010$ is a valid solution since $x_2 \cdot y_2 = 0$.

Here we summarize the general idea used in the following proofs. We will start with an instance of OV: $X,Y \subseteq \{0,1\}^d$ such that $|X| = |Y| = n$. Then we construct in $\mathcal{O}(nd)$ time a pattern P and a text $\widetilde{T}$ such that there is a match of P in $\widetilde{T}$ if and only if there exists a pair of orthogonal vectors between X and Y. We will ensure that the size of both P and $\widetilde{T}$ is $\mathcal{O}(nd)$. In this way, a sub-quadratic algorithm for matching P in $\widetilde{T}$ would imply an $\mathcal{O}((nd)(nd)^{1-\varepsilon}) = \mathcal{O}(n^{2-\varepsilon}\text{poly}(d))$ time algorithm for OV, which contradicts OVH.

Matching 1-*D* and *k*-*D* We start by introducing a gadget that will be used in several reductions. Given a vector $y \in \{0,1\}^d$, let $Q(y)$ be a 1-*D* string given by d segments $Q(y)[h]$, $1 \le h \le d$, defined as

$$Q(y)[h] = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \quad \text{if } y[h] = 0; \qquad Q(y)[h] = \{0\} \quad \text{if } y[h] = 1.$$

The key property, which is clear by construction, is that a string x matches in $Q(y)$ only if it encodes a vector orthogonal to y.

Lemma 38. *Let $x,y \in \{0,1\}^d$, then the string $x[1]x[2]\cdots x[d]$ occurs in $Q(y)$ if and only if $x \cdot y = 0$.* □

Theorem 21. No algorithm can solve PVART(1-*D*,*k*-*D*) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

Proof. Let $X,Y \subseteq \{0,1\}^d$, $|X| = |Y| = n$ be an instance of OV. We define a 1-*D* pattern P and a *k*-*D* text $\widetilde{T}$ such that P occurs in $\widetilde{T}$ if and only if $\exists x \in X, y \in Y, x \cdot y = 0$. We start by constructing pattern gadgets $Q(y_i)$, for each $y_i \in Y$. We then concatenate such gadgets into a single 1-*D* pattern using an extra character \$ that will force synchronisation with $\widetilde{T}$:

$$P = \{\$\}\, Q(y_1)\, \{\$\} \cdots \{\$\}\, Q(y_n).$$

We remark that the size of P is $M = \mathcal{O}(nd)$. To build the text $\widetilde{T}$, we list all the vectors from X in one segment W, surrounded by $n-1$ segments of the form $Z = \{\$0^d\}$ on

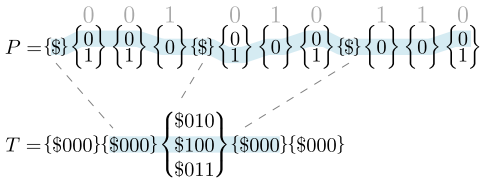


Figure 2.3: Example of the pattern and text constructed from $X = \{010, 100, 011\}$ and $Y = \{001, 010, 110\}$ in the proof of Theorem 21. The highlighted paths represent an occurrence of P in $\widetilde{T}$, which identifies two orthogonal vectors: $x_2 = 100 \in X$ and $y_2 = 010 \in Y$. The dashed lines are drawn to emphasise the synchronisation forced by the symbol \$.

both sides:

$$\widetilde{T} = \underbrace{\{\$\overbrace{0\cdots0}^{d \text{ times}}\}\cdots\{\$0\cdots0\}}_{n-1 \text{ times}} \underbrace{\begin{Bmatrix} \$x_1[1]\cdots x_1[d] \\ \vdots \\ \$x_n[1]\cdots x_n[d] \end{Bmatrix}}_{W} \underbrace{\overbrace{\{\$0\cdots0\}}^{Z}\cdots\{\$0\cdots0\}}_{n-1 \text{ times}}$$

ra Clearly, $\widetilde{T}$ is a $(d+1)$-D string of size $N = \mathcal{O}(nd)$. The idea is that Z can match any gadget $Q(y_i)$, while W allows matches only from gadgets encoding vectors that are orthogonal to a vector in X (Lemma 38). Since P has n gadgets of length d, any match of P in $\widetilde{T}$ must span a string in W (see Figure 2.3). Summing up, if P has a match in $\widetilde{T}$ starting at $\widetilde{T}[i]$, then it must start at the first position of $\widetilde{T}[i]$ because the \$ symbol matches nowhere else. Then i must be less or equal than n, and the intersection of $\mathcal{L}(Q(y_{n-i+1}))$ and $\mathcal{L}(W)$ must be non empty, implying that vector y_{n-i+1} is orthogonal to some vector of X. Therefore, deciding if there exists a pair of orthogonal vectors between X and Y can be reduced in $\mathcal{O}(nd)$ time to an instance of matching a 1-D pattern P of size $\mathcal{O}(nd)$ in a k-D text $\widetilde{T}$ of size $\mathcal{O}(nd)$. If we could find a match for P in $\widetilde{T}$ in $\mathcal{O}(N^{1-\varepsilon}M)$ or $\mathcal{O}(NM^{1-\varepsilon})$ time, then we could solve OV in $\mathcal{O}((nd)(nd)^{1-\varepsilon}) = \mathcal{O}(n^{2-\varepsilon}\,\mathrm{poly}(d))$ time, which contradicts OVH. □

Theorem 22. No algorithm can solve PVART(k-D,1-D) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

Proof. The reduction is entirely analogous to that of Theorem 21, except now we define a 1-D text $\widetilde{T} = \{\$\}\,Q(y_1)\,\{\$\}\cdots\{\$\}\,Q(y_n)$ and $(d+1)$-D pattern $P = W$ of length 1 containing all the vectors from X. We can conclude as before. □

Matching 1-F and 1-D In the following proofs we will use a three-level strategy first introduced in [86, 88].

We start by defining the two underlying 1-D strings P and $\widetilde{T}$ (one for the pattern and one for the text) common to all the 1-F graphs in the reductions of this section. Let $X, Y \subseteq \{0,1\}^d$, $|X| = |Y| = n$ be any instance of OV. $\widetilde{T}$ and P are over the alphabet $\Sigma = \{0, 1, u, b, \$\}$. The role of the letters u and b is to identify parts of the segments

that we will call their upper and bottom level, respectively. We build P as the concatenations of n 1-D string gadgets, one for each vector of X: given $x_i \in X$, we define

$$p(x_i) = \begin{Bmatrix} u \\ x_i[1] \\ b \end{Bmatrix} \cdots \begin{Bmatrix} u \\ x_i[d] \\ b \end{Bmatrix}.$$

We add the symbol \$ at the beginning and end of the pattern to force the occurrences to start/end in some specific parts of the text. The complete 1-D pattern then is

$$P = \{\$\}p(x_1)p(x_2)\cdots p(x_n)\{\$\}, \tag{2.1}$$

thus $|P| = nd+2 = \mathcal{O}(nd)$ and $\|P\| = 3nd+2 = \mathcal{O}(nd)$ (see Figure 2.4 for an example).

To build text $\widetilde{T}$, we consider three different 1-D strings: T_{left}, $\widetilde{T}^{\perp}$ and T_{right}. The 1-D string T_{left} consists of a single segment $\{\$\}$ followed by $n-1$ gadgets $U_{left} = \{u\}^{d-1}\begin{Bmatrix}\$\\ u\end{Bmatrix}$. In a symmetric way, T_{right} consists of $n-1$ gadgets $U_{right} = \begin{Bmatrix} b \\ \$ \end{Bmatrix}\{b\}^{d-1}$ followed by $\{\$\}$.

The 1-D string $\widetilde{T}^{\perp}$ is built using a slight variation of the gadgets $Q(y_i)$ (see proof of Theorem 21) extended with a three-level structure. Given $y_j \in Y$, we define $T_j^{\perp}[1] = \{u\} \cup Q(y_j)[1] \cup \{b,\$\}$, $T_j^{\perp}[h] = \{u\} \cup Q(y_j)[h] \cup \{b\}$ for $2 \le h \le d-1$ and $T_j^{\perp}[d] = \{u,\$\} \cup Q(y_j)[d] \cup \{b\}$. We define $\widetilde{T}^{\perp}$ as the concatenation of $T_1^{\perp}\cdots T_n^{\perp}$. Finally, the full 1-$D$ text $\widetilde{T}$ is the concatenation of T_{left}, $\widetilde{T}^{\perp}$ and T_{right} (see Figure 2.5 for an example):

$$\widetilde{T} = \{\$\}U_{left}^{n-1}T_1^{\perp}\cdots T_n^{\perp}U_{right}^{n-1}\{\$\} \tag{2.2}$$

We remark that the size of $\widetilde{T}$ is $\mathcal{O}(nd)$, since it is built from three 1-D strings of size $\mathcal{O}(nd)$.

Theorem 23. No algorithm can solve PVART(1-D,1-F) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

Proof. Let $X, Y \subseteq \{0,1\}^d$, $|X| = |Y| = n$ and $\Sigma = \{0,1,u,b,\$\}$. We build the pattern P as in Equation (2.1). To build text G_T, we consider $\widetilde{T}$ as in Equation (2.2) and we first separately construct three different 1-F graphs: $G_{left} = (T_{left}, E_{left})$, $G^{\perp} = (\widetilde{T}^{\perp}, E^{\perp})$ and $G_{right} = (T_{right}, E_{right})$. The set of edges E_{left} of G_{left} contains every possible edge between consecutive segments except for that of type $(u,\$)$, which has no incoming edge to \$. Symmetrically, the set E_{right} of G_{right} contains every possible edge between consecutive segments except for the one of the form $(\$,b)$, which has no outgoing edge from \$.

To define $E^{\perp}$, we first define the edges for each $T_j^{\perp}$. For these gadgets, we allow all the edges of the form (u,u), (b,b) and (x,y) for any $x,y \in \{0,1\}$. In this way, any matching string in $\mathcal{L}(\mathcal{P})$ passing through $T_j^{\perp}$ must follow the upper level identified by letters u, the bottom level identified by letters b or the orthogonal level corresponding to $Q(y_j)$ (recall that any string $x \in \{0,1\}^d$ matches $Q(y_j)$ if and only if $x \cdot y_j = 0$). To complete $E^{\perp}$, for all consecutive segments $\widetilde{T}_i^{\perp}[d]$ and $\widetilde{T}_{i+1}^{\perp}[1]$ we build two sets of

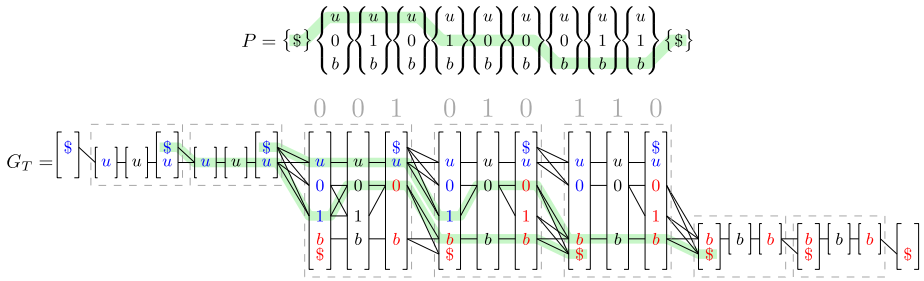


Figure 2.4: Example of a 1-*D* pattern P and 1-*F* text G_T constructed from $X = \{010, 100, 011\}$ and $Y = \{001, 010, 110\}$ following the proof of Theorem 23. Highlighted paths show some occurrences of P in $\widetilde{T}$, given by the string $\$uuu100bbb\$ \in \mathcal{L}(P)$ which belongs also to $\mathcal{L}(\widetilde{T}[4\dots14])$ and $\mathcal{L}(\widetilde{T}[7\dots17])$. This corresponds to the fact that $x_2 = 100$ is orthogonal to $y_1 = 100$ and $y_2 = 010$.

edges: the first connects the upper level of $\widetilde{T}_i^{\perp}[d]$ (i.e. u and \$) to the upper and orthogonal level of $\widetilde{T}_{i+1}^{\perp}[1]$; the second connects the orthogonal and bottom level of $\widetilde{T}_i^{\perp}[d]$ with the bottom level of $\widetilde{T}_{i+1}^{\perp}[1]$ (i.e. b and \$). Finally, $G_T = (\widetilde{T}, E_T)$ is the union of G_{left}, $G^{\perp}$ and G_{right} where we add to E_T all the edges from the last segment of T_{left} to the upper level and orthogonal level of $T_1^{\perp}[1]$ and all the edges from the orthogonal and bottom level of $T_n^{\perp}[d]$ to the first segment of T_{right}. We remark that $|E_T| = \mathcal{O}(nd)$, thus since $\|\widetilde{T}\| = \mathcal{O}(nd)$, the size of G_T is $\mathcal{O}(nd)$.

From the construction of the edges, we have that the only allowed edges from \$ are $(\$, u)$, $(\$, 0)$ and $(\$, 1)$ and the only allowed edges to \$ are $(0, \$)$, $(1, \$)$ and $(b, \$)$. Hence, the pattern must start either on the upper level or in G_{left} and finish either on the bottom level or in G_{right}. Since the bottom level (thus also G_{right}) can be reached only after reading the orthogonal level, we deduce by Lemma 38 that the pattern has a match in G_T if and only if x_i matches with $Q(y_j)$ for some i, j, namely $x_i \cdot y_j = 0$. See Figure 2.4 for an example. □

Theorem 24. No algorithm can solve PVART(1-*F*,1-*D*) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

Proof. Let $X, Y \subseteq \{0,1\}^d$, $|X| = |Y| = n$ and $\Sigma = \{0, 1, u, b, \$\}$. We build the text $\widetilde{T}$ as in Equation (2.2). To build the pattern $G_P = (P, E_P)$, we consider P as in Equation (2.1) and we construct the set of edges E_P with the same criteria used for G_T in the proof of Theorem 23. At the beginning of the pattern, we add the edges $(\$, u)$, $(\$, x_1[1])$, and at the end we add the edges $(x_n[d], \$)$ and $(b, \$)$. For each $x_i \in X$, we add to $p(x_i)$ all the edges of the form (u, u), (b, b) and (x, y) for any $x, y \in \{0, 1\}$. Instead, from $p(x_i)$ to $p(x_{i+1})$ we add the edges: (u, u), $(u, x_{i+1}[1])$, $(x_i[d], b)$, (b, b). Clearly, $|E_P| \leq 4nd = \mathcal{O}(nd)$, thus the size of G_P is $\|P\| + |E_P| = \mathcal{O}(nd)$. Furthermore, we can deduce that the language of G_P is $\mathcal{L}(G_P) = \{\$u^{(i-1)d} \cdot x_i \cdot b^{(n-i)d}\$: i = 1, \dots, n\}$. Remark that each x_i is over the alphabet $\{0, 1\}$. Thus, because of the position of the symbol \$, Lemma 38 applies, that is, there is an occurrence of P in $\widetilde{T}$ if and only if x_i matches with $Q(y_j)$ for some i, j, namely $x_i \cdot y_j = 0$. See Figure 2.5 for an example of this reduction. □

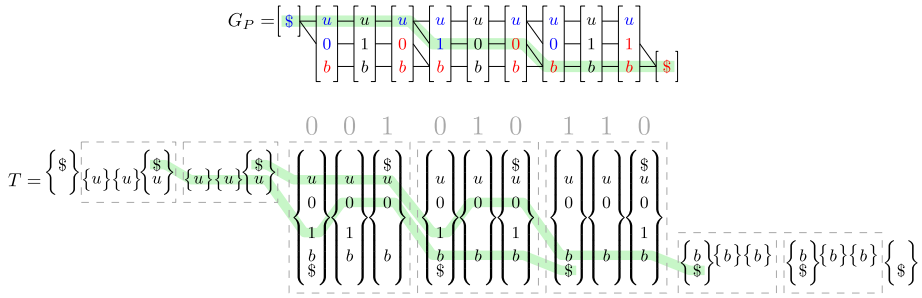


Figure 2.5: Example of a 1-F pattern G_P and a 1-D text $\widetilde{T}$ constructed from $X = \{010, 100, 011\}$ and $Y = \{001, 010, 110\}$ following the proof of Theorem 24, with occurrences of G_P in $\widetilde{T}$ highlighted.

Finally, the following result directly follows from a reduction that uses G_T as in the proof of Theorem 23 and G_P as in the proof of Theorem 24.

Theorem 25. No algorithm can solve PVART(1-F,1-F) on constant alphabet in $\mathcal{O}(M^{1-\varepsilon}N)$ nor in $\mathcal{O}(MN^{1-\varepsilon})$ time for $\varepsilon > 0$, unless OVH is false.

2.2 APPROXIMATE PATTERN MATCHING

2.2.1 INTRODUCTION

This section is based on [40] (which has been extended in [41]). We define the main problem, which is the approximate version of PVART(SOLID,ED), as follows :

Problem 4. 1-ERROR EDSM
Input: A string P of length m and an ED string $\widetilde{T}$ of length n and size N.
Output: All positions j' in $\widetilde{T}$ such that there is at least one string P' with an occurrence ending at position j' in $\widetilde{T}$, and with $d_E(P, P') \leq 1$ (reporting version); or YES if and only if there is at least one string P' with an occurrence in $\widetilde{T}$, and with $d_E(P, P') \leq 1$ (decision version).

An occurrence of a string P' as in the problem definition is called an *occurrence of P with 1 error* (or a *1-error occurrence*).

We also define an alternative version, this time based on Hamming distance:

Problem 5. 1-MISMATCH EDSM
Input: A string P of length m and an ED string $\widetilde{T}$ of length n and size N.
Output: All positions j' in $\widetilde{T}$ such that there is at least one string P' with an occurrence ending at position j' in $\widetilde{T}$, and with $d_H(P, P') \leq 1$.

An occurrence of a string P' as in the problem definition is called an *occurrence of P with 1 mismatch* (or a *1-mismatch occurrence*). We call *mismatch* the single position i with $P[i] \neq P'[i]$.

Approximate EDSM	Features	Running time
Bernardini et al. [45]	k errors	$\mathcal{O}(k^2mB+kN)$
This work	1 error	$\mathcal{O}(nm^3+N)$
This work	1 error	$\mathcal{O}((nm^2+N)\log m)$
This work	1 error (decision)	$\mathcal{O}(nm^2\sqrt{\log m}+N\log\log m)$
Bernardini et al. [45]	k mismatches	$\mathcal{O}(kmB+kN)$
This work	1 mismatch	$\mathcal{O}(nm^3+N)$
This work	1 mismatch	$\mathcal{O}(nm^2+N\log m)$

Table 2.2: The state of the art results for approximate EDSM and our new results for $k=1$. Note that $n \leq B \leq N$. All algorithms underlying these results are combinatorial and all the reporting algorithms are online.

Our Results and Techniques In string matching, a single extra or missing letter in the pattern or in a potential occurrence results in missing (many or all) occurrences. Hence, many works are focused on approximate string matching for standard strings [22, 55, 66, 103, 150, 151]. For approximate k-EDSM, Bernardini et al. [45] presented an online $\mathcal{O}(k^2mB+kN)$-time algorithm under edit distance and an online $\mathcal{O}(kmB+kN)$-time algorithm under Hamming distance, where k is the maximum allowed number of errors (edits) or mismatches, respectively. Unfortunately, B is only bounded by N, and so even for $k=1$, the existing algorithms run in $\Omega(mN)$ time in the worst case.

Let us remark that the special case of $k=1$ is not interesting for approximate string matching on standard strings: the existing algorithms have a polynomial dependency on k and a linear dependency on the length n of the text, and thus for $k=1$ we trivially obtain $\mathcal{O}(n)$-time algorithms under edit or Hamming distance. However, this is not the case for other string problems, such as text indexing with errors, where the first step was to design a data structure for 1 error [21]. The next step, extending it to k errors, required the development of new highly non-trivial techniques and incurred some exponential factor with respect to k [64]. Interestingly, k-EDSM seems to be the same case, which highlights the main theoretical motivation of this work. In Table 2.2, we summarize the state of the art for approximate EDSM and our new results for $k=1$. Note that the reporting algorithms underlying our results are also *online*.

Indeed, to arrive at our main results, we design a rich non-trivial combination of algorithmic techniques. Our algorithms for edit distance rely on non-trivial reductions from 1-EDSM to special instances of classic computational geometry problems (2D RECTANGLE STABBING or 2D RANGE EMPTINESS), which we show how to solve efficiently. In order to obtain an even faster algorithm for Hamming distance, we also rely on employing and adapting the k-errata trees of Cole et al. for text indexing with k errors [64].

The combinatorial algorithms we develop here for approximate EDSM are good in the following sense. First, the running times of our algorithms do not depend on B,

a highly desirable property. Specifically, all of our results replace $m \cdot B$ by an $n \cdot \text{poly}(m)$ factor. Second, our $\widetilde{\mathcal{O}}(nm^2+N)$-time algorithms are at most one $\log m$ factor slower than $\mathcal{O}(nm^2+N)$, the best-known bound obtained by a combinatorial algorithm (not employing fast Fourier transforms) for *exact* EDSM [109]. Notably, for Hamming distance, we show an $\mathcal{O}(nm^2+N\log m)$-time algorithm. Last, our $\mathcal{O}(nm^3+N)$-time algorithms have a linear dependency on N, another highly desirable property (at the expense of an extra m-factor).

2.2.2 PRELIMINARIES

Let P' be an occurrence of P with 1 error starting at position j and ending at position j' in an *ED* string $\widetilde{T}$, equivalently, we say that P *matches* $\widetilde{T}[j\,..\,j']$ *with* 1 *error*. Let $UP'_j,\ldots,P'_{j'}V$ be an alignment of P' with $\widetilde{T}[j\,..\,j']$ and $i \in [j\,..\,j']$ be an integer such that the single replacement, insertion, or deletion required to obtain P from $P' = P'_j \cdot \ldots \cdot P'_{j'}$ occurs on P'_i. We then say that the alignment (and the occurrence) *has the* 1 *error in* $\widetilde{T}[i]$ (it should be clear that for one alignment we may have multiple different i). We show the following theorem.

Theorem 39. Given a pattern P of length m and an ED text $\widetilde{T}$ of length n and size N, the reporting version of 1-ERROR EDSM can be solved online in $\mathcal{O}(nm^2\log m+N\log m)$ or $\mathcal{O}(nm^3+N)$ time. The decision version of 1-ERROR EDSM can be solved off-line in $\mathcal{O}(nm^2\sqrt{\log m}+N\log\log m)$ time.

Theorem 40. Given a pattern P of length m and an ED text $\widetilde{T}$ of length n and size N, 1-MISMATCH EDSM can be solved online in $\mathcal{O}(nm^2+N\log m)$ or $\mathcal{O}(nm^3+N)$ time.

Definition 41. For a string $P = P[1\,..\,m]$, an ED string $\widetilde{T} = \widetilde{T}[1\,..\,n]$, a position $1 \le i \le n$, and a distance on Σ^*, we define three sets:

- $AP_i \subseteq [1\,..\,m]$, such that $j \in AP_i$ if and only if $P[1\,..\,j]$ is an *active prefix* of P in $\widetilde{T}$ ending in the segment $\widetilde{T}[i]$, that is, a prefix of P which is also a suffix of a string in $\mathcal{L}(\widetilde{T}[1\,..\,i])$.
- $AS_i \subseteq [1\,..\,m]$, such that $j \in AS_i$ if and only if $P[j\,..\,m]$ is an *active suffix* of P in $\widetilde{T}$ starting in the segment $\widetilde{T}[i]$, that is, a suffix of P which is also a prefix of a string in $\mathcal{L}(\widetilde{T}[i\,..\,n])$.
- 1-$AP_i \subseteq [1\,..\,m]$, such that $j \in$ 1-AP_i if and only if $P[1\,..\,j]$ is an *active prefix with* 1 *error* of P in $\widetilde{T}$ ending in the segment $\widetilde{T}[i]$, that is, a prefix of P which is also at distance at most 1 from a suffix of a string in $\mathcal{L}(\widetilde{T}[1\,..\,i])$.

For convenience, we also define $AP_0 = AS_{n+1} =$ 1-$AP_0 = \emptyset$.

Computing the sets 1-AP plays the key role in the reporting version of our algorithm for 1-ERROR EDSM (see Figure 2.6). Finding active prefixes (and, up to string reversal, suffixes) reduces to the APE problem (Problem 1)

Given an algorithm for the APE problem working in $f(m)+N$ time, we can find *all* active prefixes for a pattern P of length m in an ED text $\widetilde{T} = \widetilde{T}[1\,..\,n]$ of size N in $\mathcal{O}(nf(m)+N)$ total time. Lemma 9 implies the following:

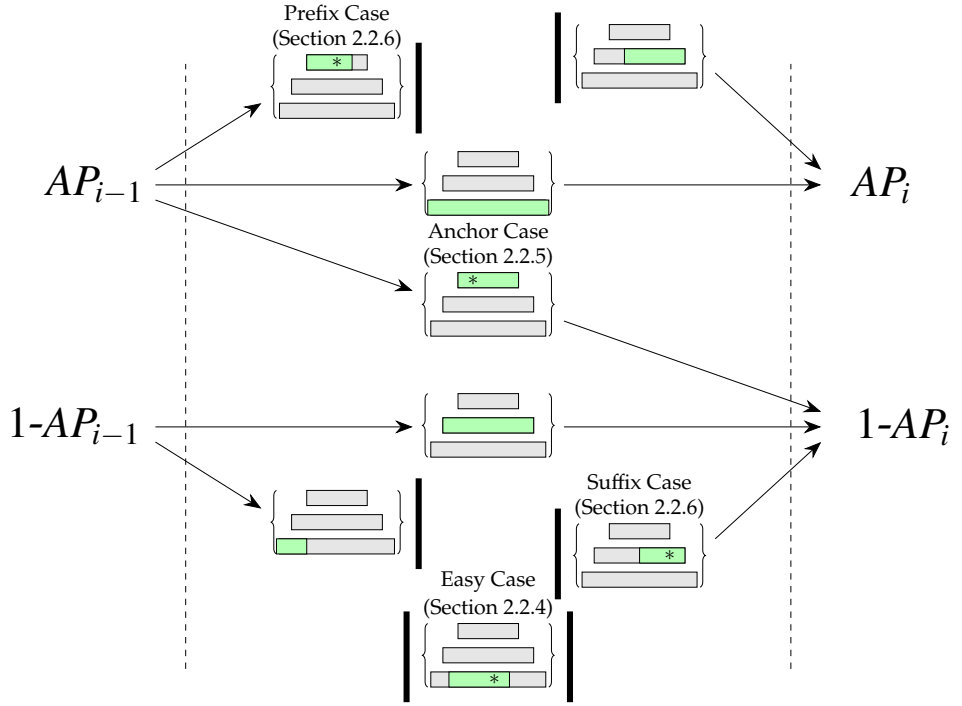


Figure 2.6: The layout of the algorithms for computing AP_i, 1-AP_i, and reporting occurrences. The green areas correspond to the (partial) matches in $\widetilde{T}[i]$, and the symbol $*$ indicates the position of an error. The vertical bold lines indicate the beginning/the end of an occurrence or a 1-error occurrence. The cases without a label allow only exact matches and were already solved by Grossi et al. in [109].

Corollary 42 ([109]). *For a pattern P of length m and an ED text $\widetilde{T} = \widetilde{T}[1..n]$ of total size N, computing the sets AP_i for all $i \in [1..n]$ takes $\mathcal{O}(nm^{\omega-1} + N)$ time.*

As depicted in Figure 2.6, the computation of active prefixes with 1 error (1-AP_i) and the reporting of occurrences with 1 error reduce to a problem where the error can only occur in a single, fixed $\widetilde{T}[i]$. In particular, this problem decomposes into 4 cases, which we formalize in the following proposition.

Proposition 43. *Let $\widetilde{T} = \widetilde{T}[1..n]$ be an ED text and P be a pattern that has an occurrence with 1 error (resp. 1 mismatch) in $\widetilde{T}$. For each alignment corresponding to such occurrence, at least one of the following is true:*

Easy Case: *P matches $\widetilde{T}[i]$ with 1 error (resp. 1 mismatch) for some $1 \le i \le n$.*

Anchor Case: *P matches $\widetilde{T}[j..j']$ with 1 error (resp. 1 mismatch) in $\widetilde{T}[i]$ for some $1 \le j < i < j' \le n$. $\widetilde{T}[i]$ is called the* anchor *of the alignment.*

Prefix Case: *P matches $\widetilde{T}[j..i]$ with 1 error (resp. 1 mismatch) in $\widetilde{T}[i]$ for some $1 \le j < i \le n$, implying an active prefix of P which is a suffix of a string in $\mathcal{L}(\widetilde{T}[j..i-1])$.*

Suffix Case: *P matches $\widetilde{T}[i..j']$ with 1 error (resp. 1 mismatch) in $\widetilde{T}[i]$ for some $1 \leq i < j' \leq n$, implying an active suffix of P which is a prefix of a string in $\mathcal{L}(\widetilde{T}[i+1..j'])$.*

Proof. Suppose P has a 1-error (resp. 1 mismatch) occurrence matching $\widetilde{T}[j..j']$ with $1 \leq j \leq j' \leq n$. If $j = j'$ we are in the Easy Case. Otherwise, each alignment has an error in some $\widetilde{T}[i]$ for $j \leq i \leq j'$. If $j < i < j'$, we are in the Anchor Case; if $j < i = j'$, we are in the Prefix Case; and if $j = i < j'$, we are in the Suffix Case. □

2.2.3 1-ERROR EDSM

In this section, we present algorithms for finding all 1-error occurrences of P given by each type of possible alignment described by Proposition 43 (inspect Figure 2.7). The Prefix and Suffix cases are analogous up to string reversal; the only difference is in that, while the Suffix Case computes new 1-AP, the Prefix Case is used to actually report occurrences. They are jointly considered in Subsection 2.2.6.

We follow two different procedures for the decision and reporting versions. For the decision version, we precompute sets AP_i and AS_i, for all $i \in [1..n]$, using Corollary 42, and we simultaneously compute possible exact occurrences of P. Then we compute 1-error occurrences of P by grouping the alignments depending on the segment i in which the error occurs, and using AP_i and AS_i. For the reporting version, we consider one segment $\widetilde{T}[i]$ at a time (online) and extend partial exact or 1-error occurrences of P to compute sets AP_i and 1-AP_i using just sets AP_{i-1} and 1-AP_{i-1} computed at the previous step. We design different procedures for the 4 cases of Proposition 43. We can sort all letters of P, assign them rank values from $[1..m]$, and construct a perfect hash table over these letters supporting $\mathcal{O}(1)$-time look-up queries in $\mathcal{O}(m \log m)$ time [186]. Any letter of $\widetilde{T}$ not occurring in P can be replaced by the same special letter in $\mathcal{O}(1)$ time. In the rest we thus assume that the input strings are over $[1..m+1]$.

In Subsection 2.2.7, we note that the 2D RECTANGLE STABBING instances arising from 1-ERROR EDSM have a special structure. We show how to solve them efficiently thus shaving logarithmic factors from the time complexity.

2.2.4 EASY CASE

The Easy Case can be reduced to approximate string matching with at most 1 error (1-SM):

Problem 8. 1-SM
Input: A string P of length m and a standard string T of length n.
Output: All positions j in T such that there is at least one string P' ending at position j in T with $d_E(P, P') \leq 1$.

We have the following well-known results.

Lemma 44 ([66, 151])**.** *Given a pattern P of length m, a text T of length n, and an integer $k > 0$, all positions j in T such that the edit distance of $T[i..j]$ and P, for some position $i \leq j$*

on T, is at most k, can be found in $\mathcal{O}(kn)$ time or in $\mathcal{O}(\frac{nk^4}{m}+n)$ time.[3] *In particular, 1-SM can be solved in $\mathcal{O}(n)$ time.*

We find occurrences of P with at most 1 error that are in the Easy Case for segment $\widetilde{T}[i]$ in the following way: we apply Lemma 44 for $k=1$ and every string of $\widetilde{T}[i]$ whose length is at least $m-1$ (any shorter string is clearly not relevant for this case) as text. If, for any of those strings, we find an occurrence of P, we report an occurrence at position i (inspect Figure 2.7a). The time for processing a segment $\widetilde{T}[i]$ is $\mathcal{O}(N_i)$, where N_i is the total length of all the strings in $\widetilde{T}[i]$.

2.2.5 ANCHOR CASE

Let $\widetilde{T}$ be an ED text and P be a pattern with a 1-error occurrence and an alignment in the Anchor Case with anchor $\widetilde{T}[i]$. Further let $L=P[1..\ell]S'$ and $Q=S''P[q..m]$ be a prefix and a suffix of P, respectively, for some $\ell\in AP_{i-1}, q\in AS_{i+1}$, where S',S'' are a prefix and a suffix of some $S\in\widetilde{T}[i]$, respectively (strings S',S'' can be empty). By Lemma 4, a pair L,Q gives a 1-error occurrence of P if one of the following holds:

1 mismatch: $|L|+|Q|+1=m$ *and* $|S'|+|S''|+1=|S|$ (inspect Figure 2.7b).

1 deletion in P: $|L|+|Q|=m-1$ *and* $|S'|+|S''|=|S|$.

1 insertion in P: $|L|+|Q|=m$ *and* $|S'|+|S''|+1=|S|$.

We show how to find such pairs with the use of a geometric approach. For convenience, we only present the Hamming distance (1 mismatch) case. The other cases are handled similarly.

Let $\lambda\in AP_{i-1}$ be the length of an active prefix, and let ρ be the length of an active suffix, that is, $m-\rho+1\in AS_{i+1}$. Note that AP_{i-1} and AS_{i+1} can be precomputed, for all i, in $\mathcal{O}(nm^{\omega-1}+N)=\mathcal{O}(nm^2+N)$ total time by means of Corollary 42. (In particular, AS_{i+1} is required only for the decision version; for the reporting version, we explain later on how to avoid the precomputation of AS_{i+1} to obtain an online algorithm.) We will exhaustively consider all pairs (λ,ρ) such that $\lambda+\rho<m$. Clearly, there are $\mathcal{O}(m^2)$ such pairs.

Consider the length $\mu=m-(\lambda+\rho)>0$ of the substring of P still to be matched for some prefix and suffix of P of lengths (λ,ρ), respectively. We group together all pairs (λ,ρ) such that $m-(\lambda+\rho)=\mu$ by sorting them in $\mathcal{O}(m^2)$ time. We construct, for each such group μ, the compacted trie $\mathcal{T}_\mu$ of the fragments $P[\lambda+1..m-\rho]$, for all (λ,ρ) such that $m-(\lambda+\rho)=\mu$, and analogously the compacted trie $\widetilde{T}_\mu^R$ of all fragments $P^R[\rho+1..m-\lambda]$. For each group μ, this takes $\mathcal{O}(m)$ time [169]. We enhance all nodes with a perfect hash table in $\mathcal{O}(m)$ total time to access edges by the first letter of their label in $\mathcal{O}(1)$ time [93].

We also group all strings in segment $\widetilde{T}[i]$ of length less than m by their length μ. The group for length μ is denoted by G_μ. This takes $\mathcal{O}(N_i)$ time. Clearly, the strings

[3] Charalampopoulos et al. have announced an improvement on the exponent of k from 4 to 3.5; specifically, they presented an $\mathcal{O}(\frac{nk^{3.5}\sqrt{\log m\log k}}{m}+n)$-time algorithm [56].

in G_μ are the only candidates to extend pairs (λ,ρ) such that $m-(\lambda+\rho)=\mu$. Note that the mismatch can be at any position of any string of G_μ: its position determines a prefix S' of length h and a suffix S'' of length k of the same string S, with $h+k=\mu-1$, that must match a prefix and a suffix of $P[\lambda+1\,..\,m-\rho]$, respectively. We will consider all such pairs of positions (h,k) whose sum is $\mu-1$ (intuitively, the minus one is for the mismatch). This guarantees that $L=P[1\,..\,\lambda]S'$ and $Q=S''P[m-\rho+1\,..\,m]$ are such that $|L|+|Q|+1=m$. The pairs are $(0,\mu-1),(1,\mu-2),\ldots,(\mu-1,0)$. This guarantees that L and Q are *one position apart* ($|S'|+|S''|+1=|S|$).

The number of these pairs is $\mathcal{O}(\mu)=\mathcal{O}(m)$. Consider one such pair (h,k) and a string $S\in G_\mu$. We treat every such string S separately. We spell $S[1\,..\,h]$ in $\mathcal{T}_\mu$. If the whole $S[1\,..\,h]$ is successfully spelled ending at a node u, this implies that all the fragments of P corresponding to nodes descending from u share $S[1\,..\,h]$ as a prefix. We also spell $S^R[1\,..\,k]$ in $\widetilde{T}_\mu^R$. If the whole of $S^R[1\,..\,k]$ is successfully spelled ending at a node v, then all the fragments of P corresponding to nodes descending from v share $(S^R[1\,..\,k])^R$ as a suffix. Nodes u and v identify an interval of leaves in $\mathcal{T}_\mu$ and $\widetilde{T}_\mu^R$, respectively. We need to check if these intervals both contain a leaf corresponding to the same fragment of P. If they do, then we obtain an occurrence of P with 1 mismatch (see Figure 2.8). We now have two different ways to proceed, depending on whether we need to solve the off-line decision version or the online reporting version.

Decision Version Let us recall that $\mathcal{T}_\mu,\widetilde{T}_\mu^R$ by construction are ordered based on lexicographic ranks. For every pair $(\mathcal{T}_\mu,\widetilde{T}_\mu^R)$, we construct a data structure for 2D RANGE EMPTINESS queries on the grid $[1\,..\,\ell]^2$, where ℓ is the number of leaves of $\mathcal{T}_\mu$ (and of $\mathcal{T}_\mu^R$), for the set of points (x,y) such that x is the lexicographic rank of the leaf of $\mathcal{T}_\mu$ representing $P[\lambda+1\,..\,m-\rho]$ and y is the rank of the leaf of $\mathcal{T}_\mu^R$ representing $P^R[\rho+1\,..\,m-\lambda]$ for the same pair (λ,ρ). This denotes that the two leaves correspond to *the same fragment* of P. For every $(\mathcal{T}_\mu,\widetilde{T}_\mu^R)$, this preprocessing takes $\mathcal{O}(m\sqrt{\log m})$ time by Lemma 6, since ℓ is $\mathcal{O}(\mu)=\mathcal{O}(m)$. For all μ groups (they are at most m), the whole preprocessing thus takes $\mathcal{O}(m^2\sqrt{\log m})$ time.

We then ask 2D RANGE EMPTINESS queries that take $\mathcal{O}(\log\log m)$ time each by Lemma 6. Note that all rectangles for S can be collected in $\mathcal{O}(|S|)=\mathcal{O}(\mu)$ time by spelling S through $\mathcal{T}_\mu$ and S^R through $\widetilde{T}_\mu^R$, one letter at a time. Thus the total time for processing all G_μ groups of segment i is $\mathcal{O}(m^2\sqrt{\log m}+N_i\log\log m)$. If any of the queried ranges turns out to be non-empty, then P' such that $d_H(P,P')\le 1$ appears in $\mathcal{L}(\widetilde{T})$ with anchor in $\widetilde{T}[i]$; we do not have sufficient information to output its ending position however.

Reporting Version For this version, we do the dual. We construct a data structure for 2D RECTANGLE STABBING queries on the grid $[1\,..\,\ell]^2$ for the set of rectangles collected for all strings $S\in G_\mu$. By Lemma 5, for all μ groups, the whole preprocessing thus takes $\mathcal{O}(N_i\log N_i)$ time.

For every $(\mathcal{T}_\mu,\widetilde{T}_\mu^R)$, we then ask the following queries: (x,y) is queried if and only if x is the rank of a leaf representing $P[\lambda+1\,..\,m-\rho]$ and y is the rank of a

leaf representing $P^R[\rho+1\,..\,m-\lambda]$. For every $(\mathcal{T}_\mu, \widetilde{T}^R_\mu)$, this takes $\mathcal{O}(m\log N_i)$ time by Lemma 5 and by the fact that for each group G_μ there are $\mathcal{O}(m)$ pairs (λ,ρ) such that $m-(\lambda+\rho)=\mu$. For all groups G_μ (they are at most m), all the queries thus take $\mathcal{O}(m^2\log N_i)$ time. Thus the total time for processing all G_μ groups of segment i is $\mathcal{O}((m^2+N_i)\log N_i)$.

We are not done yet. By performing the above algorithm for active prefixes and active suffixes, we find out which pairs can be completed to a full occurrence of P with at most 1 error. This information is not sufficient to compute where such an occurrence ends (and storing additional information together with the active suffixes may prove costly). To overcome this, we use some ideas from the decision algorithm, appropriately modified to preserve the online nature of the reporting algorithm. Instead of iterating ρ over the lengths of precomputed active suffixes, we iterate it over *all* possible lengths in $[0\,..\,m]$ (including 0 because we may want to include m in 1-AP_i). A suffix of P of length ρ completes a partial occurrence computed up to segment i exactly when $m-\rho \in$ 1-AP_i (a pair $x \in$ 1-$AP_i, x+1 \in AS_{i+1}$ corresponds to an occurrence). We thus use the reporting algorithm to compute the part of 1-AP_i coming from the extension of AP_{i-1} (see Figure 2.6), and defer the reporting to the no-error version of the Prefix Case for the right j'; which was solved by Grossi et al. [109] in linear time.

2.2.6 PREFIX CASE

Let $\widetilde{T}$ be an ED text and P be a pattern with a 1-error occurrence and an alignment in the Prefix Case with active prefix ending at $\widetilde{T}[i-1]$. Let $L = P[1\,..\,\ell]S'$, with $\ell \in AP_{i-1}$, be a prefix of P that is extended in $\widetilde{T}[i]$ by S'; and Q be a suffix of P occurring in some string of $\widetilde{T}[i]$ (strings S', Q can be empty). By Lemma 4, we have 3 possibilities for any alignment of a 1-error occurrence of P in the Prefix Case:

1 mismatch: $|L|+|Q|+1=m$, S' is a prefix of the same string in which Q occurs, and they are one position apart (inspect Figure 2.7c).

1 deletion in P: $|L|+|Q|=m-1$, S' is a prefix of the same string in which Q occurs, and they are consecutive.

1 insertion in P: $|L|+|Q|=m$, S' is a prefix of the same string in which Q occurs, and they are one position apart.

For convenience, we only present the method for Hamming distance (1 mismatch). The other possibilities are handled similarly.

The techniques are similar to those for the Anchor Case (Subsection 2.2.5). We group the prefixes of all strings in $\widetilde{T}[i]$ according to their length $\mu \in [1\,..\,m-1]$. The total number of these prefixes is $\mathcal{O}(N_i)$. The group for length μ is denoted by G_μ. We construct the compacted trie $\mathcal{T}_{G_\mu}$ of the strings in G_μ, and the compacted trie $\widetilde{T}^R_{G_\mu}$ of the reversed strings in G_μ. This can be done in $\mathcal{O}(N_i)$ total time for all compacted tries. To achieve this, we employ the following lemma by Charalampopoulos et al. [53]. (Recall that we have already sorted all letters of P. In what follows, we

assume that $N_i \geq m$; if this is not the case, we can sort all letters of $\widetilde{T}[i]$ in $\mathcal{O}(m+N_i)$ time.)

Lemma 45 ([53]). *Let X be a string of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$. Let I be a collection of intervals $[i\,..\,j] \subseteq [1\,..\,n]$. We can lexicographically sort the substrings $X[i\,..\,j]$ of X, for all intervals $[i\,..\,j] \in I$, in $\mathcal{O}(n+|I|)$ time.*

We concatenate all the strings of $\widetilde{T}[i]$ to obtain a single string X of length N_i, to which we apply, for each μ, Lemma 45, with a set I consisting of the intervals over X corresponding to the strings in G_μ. By sorting, in this way, all strings in G_μ (for all μ), and by constructing [91] and preprocessing [37] the generalized suffix tree of the strings in $\widetilde{T}[i]$ in $\mathcal{O}(N_i)$ time to support answering lowest common ancestor (LCA) queries in $\mathcal{O}(1)$ time, we can construct all $\mathcal{T}_{G_\mu}$ in $\mathcal{O}(N_i)$ total time. We handle $\widetilde{T}^R_{G_\mu}$, for all μ, analogously. Similar to the Anchor Case we enhance all nodes with a perfect hash table within the same complexities [93].

In contrast to the Anchor Case, we now only consider the set AP_{i-1}: namely, we do not consider AS_{i+1}. Let $\lambda \in AP_{i-1}$ be the length of an active prefix. We treat every such element separately, and they are $\mathcal{O}(m)$ in total. Let $\mu = m - \lambda > 0$ and consider the group G_μ whose strings are all of length μ. The mismatch being at position $h+1$ in one such string S determines a prefix S' of S of length h that must extend the active prefix of P of length λ, and a fragment Q of S of length $k = \mu - h - 1$ that must match a suffix of P. We will consider all such pairs (h,k) whose sum is $\mu - 1$. The pairs are again $(0,\mu-1),(1,\mu-2),\ldots,(\mu-1,0)$, and there are clearly $\mathcal{O}(\mu) = \mathcal{O}(m)$ of them.

Consider (h,k) as one such pair. We spell $P[\lambda+1\,..\,\lambda+h]$ in $\mathcal{T}_{G_\mu}$. If the whole $P[\lambda+1\,..\,\lambda+h]$ is spelled successfully, this implies an interval of leaves of $\mathcal{T}_{G_\mu}$ corresponding to strings from $\widetilde{T}[i]$ that share $P[\lambda+1\,..\,\lambda+h]$ as a prefix. We spell $P^R[1\,..\,k]$ in $\widetilde{T}^R_{G_\mu}$. If the whole $P^R[1\,..\,k]$ is spelled successfully, this implies an interval of leaves of $\mathcal{T}^R_{G_\mu}$ corresponding to strings from $\widetilde{T}[i]$ that have the same fragment $(P^R[1\,..\,k])^R$. These two intervals form a rectangle in the grid implied by the leaves of $\mathcal{T}_{G_\mu}$ and $\widetilde{T}^R_{G_\mu}$. We need to check if these intervals both contain a leaf corresponding to the same prefix of length μ of a string in $\widetilde{T}[i]$. If they do, then we have obtained an occurrence with 1 mismatch in $\widetilde{T}[i]$.

To do this we construct, for every $(\mathcal{T}_{G_\mu}, \widetilde{T}^R_{G_\mu})$, a 2d range data structure for the set of points (x,y) such that x is the rank of a leaf of $\mathcal{T}_{G_\mu}$, y is the rank of a leaf of $\widetilde{T}^R_{G_\mu}$, and the two leaves correspond to *the same prefix* of length μ of a string in $\widetilde{T}[i]$. For every $(\mathcal{T}_{G_\mu}, \widetilde{T}^R_{G_\mu})$, this takes $\mathcal{O}(|G_\mu|\sqrt{\log|G_\mu|})$ time by Lemma 6. For all G_μ groups, the whole preprocessing takes $\mathcal{O}(N_i\sqrt{\log N_i})$ time.

We then ask 2D RANGE EMPTINESS queries each taking $\mathcal{O}(\log\log|G_\mu|)$ time by Lemma 6. Note that all rectangles for λ can be collected in $\mathcal{O}(m)$ time by spelling $P[\lambda+1\,..\,\lambda+\mu-1]$ through $\mathcal{T}_{G_\mu}$ and $P^R[1\,..\,\mu-1]$ through $\widetilde{T}^R_{G_\mu}$, one letter at a time. This gives a total of $\mathcal{O}(m^2\log\log N_i + N_i\sqrt{\log N_i})$ time for processing all G_μ groups of $\widetilde{T}[i]$, because $\sum_\mu |G_\mu| \leq N_i$.

To solve the Suffix Case (compute active prefixes with 1 error starting in $\widetilde{T}[i]$) we employ the mirror version of the algorithm, but iterating λ over the whole $[0..m]$ instead of AS_{i+1} (like in the reporting version of the Anchor Case).

2.2.7 SHAVING LOGS USING SPECIAL CASES OF GEOMETRIC PROBLEMS

ANCHOR CASE: SIMPLE 2D RECTANGLE STABBING

Lemma 46. *We can solve the Anchor Case (i.e., extend AP_{i-1} into 1-AP_i) in $\mathcal{O}(m^3+N_i)$ time.*

Proof. By Lemma 5, 2D RECTANGLE STABBING queries can be answered in $\mathcal{O}(\log n)$ time using $\mathcal{O}(n\log n)$ space after $\mathcal{O}(n\log n)$-time preprocessing.

Notice that in the case of the 2D RECTANGLE STABBING used in Subsection 2.2.5 the rectangles and points are all in a predefined $[1..m]\times[1..m]$ grid. In such a case we can also use an easy folklore data structure of size $\mathcal{O}(m^2)$, which after an $\mathcal{O}(m^2+|\text{rectangles}|)$-time preprocessing answers such queries in $\mathcal{O}(1)$ time.

Namely, the data structure consists of a $[1..m+1]^2$ grid Γ (a 2d-array of integers) in which for every rectangle $[u..v]\times[w..x]$ we add 1 to $\Gamma[u][w]$ and $\Gamma[v+1][x+1]$ and -1 to $\Gamma[u][x+1]$ and $\Gamma[v+1][w]$. Then we modify Γ to contain the 2d prefix sums of its original values (we first compute prefix sums of each row, and then prefix sums of each column of the result). After these modifications, $\Gamma[x][y]$ stores the number of rectangles containing point (x,y), and hence after $\mathcal{O}(m^2+|\text{rectangles}|)$-time preprocessing we can answer 2D RECTANGLE STABBING queries in $\mathcal{O}(1)$ time.

In our case we have a total of $\mathcal{O}(m)$ such grid structures, each of $\mathcal{O}(m^2)$ size, and ask $\mathcal{O}(m^2)$ queries, and hence obtain an $\mathcal{O}(m^3+N_i)$-time and $\mathcal{O}(m^2)$-space solution for computing 1-AP_i from AP_{i-1}. □

PREFIX CASE: A SPECIAL CASE OF 2D RECTANGLE STABBING

Inspect the example of Figure 2.8 for the Anchor Case. Note that the groups of rectangles for each string have the special property of being composed of *nested intervals*: for each dimension, the interval corresponding to a given node is included in the one corresponding to any of its ancestors. Thus for the Prefix Case, where we only spell fragments of the same string P in both compacted tries, we consider the following special case of off-line 2D RECTANGLE STABBING.

Lemma 47. *Let $p_1,\ldots,p_h$ and $q_1,\ldots,q_h$ be two permutations of $[1..h]$. We denote by Π the set of h points $(p_1,q_1),(p_2,q_2),\ldots,(p_h,q_h)$ on $[1..h]^2$.*

Further let R be a collection of r axis-aligned rectangles

$$([u_1,v_1],[w_1,x_1]),\ldots,([u_r,v_r],[w_r,x_r]),$$

such that $[u_r,v_r]\subseteq[u_{r-1},v_{r-1}]\subseteq\cdots\subseteq[u_1,v_1]$ and $[w_1,x_1]\subseteq[w_2,x_2]\subseteq\cdots\subseteq[w_r,x_r]$. Then we can find out, for every point from Π, if it stabs any rectangle from R in $\mathcal{O}(h+r)$ total time.

Proof. Let H be a bit vector consisting of h bits, initially all set to zero. We process one rectangle at a time. We start with $([u_1,v_1],[w_1,x_1])$. We set $H[p]=1$ if and only if $(p,q)\in\Pi$ for $p\in[u_1,v_1]$ and any q. We collect all p such that $(p,q)\in\Pi$ and $q\in[w_1,x_1]$,

and then search for these p in H: if for any p, $H[p] = 1$, then the answer is positive for p. Otherwise, we remove from H every p such that $p \in [u_1, v_1]$ and $p \notin [u_2, v_2]$ by setting $H[p] = 0$. We proceed by collecting all p such that $(p,q) \in \Pi$, $q \in [w_2, x_2]$ and $q \notin [w_1, x_1]$, and then search for them in H: if for any p, $H[p] = 1$, then the answer is positive for p. We repeat this until H is empty or until there are no other rectangles to process.

The whole procedure takes $\mathcal{O}(h+r)$ time, because we set at most h bits on in H, we set at most h bits back off in H, we search for at most h points in H, and then we process r rectangles. Note that we can efficiently scan $(p_1,q_1),(p_2,q_2),\ldots,(p_h,q_h)$ to *update* and *query* the bit vector H, because we can maintain set Π sorted in two copies: one by the first dimension and another one by the second dimension. □

Lemma 48. *We can solve the Prefix (resp. Suffix) Case, that is, report* 1*-error occurrences ending in* $\widetilde{T}[i]$ *(resp. compute active prefixes with* 1 *error starting in* $\widetilde{T}[i]$*) in* $\mathcal{O}(m^2+N_i)$ *time.*

Proof. We employ Lemma 47 to get rid of the 2d range data structure. The key is that for every length-μ suffix $P[\lambda+1\,..\,m]$ of the pattern we can afford to pay $\mathcal{O}(\mu+|G_\mu|)$ time plus the time to construct $\mathcal{T}_{G_\mu}$ and $\widetilde{T}^R_{G_\mu}$ for set G_μ. Because the grid is $[1\,..\,|G_\mu|]^2$, we exploit the fact that the intervals found by spelling $P[\lambda+1\,..\,\lambda+\mu-1]$ through $\mathcal{T}_{G_\mu}$ and $P^R[1\,..\,\mu-1]$ through $\widetilde{T}^R_{G_\mu}$, one letter at a time, are subset of each other, and querying μ such rectangles is done in $\mathcal{O}(\mu+|G_\mu|)$ time by employing Lemma 47. Since we process at most m distinct length-μ suffixes of P, the total time is $\mathcal{O}(m^2+N_i)$, because $\sum_\mu |G_\mu| \leq N_i$. □

2.2.8 WRAPPING-UP

To obtain Theorem 39 for the decision version of the problem we first compute AP_i and AS_i, for all $i \in [1\,..\,n]$, in $\mathcal{O}(nm^{\omega-1}+N)$ total time (Corollary 42). We then compute all the occurrences in the Easy Cases using $\mathcal{O}(N)$ time in total (Subsection 2.2.4); and we finally compute all the occurrences in the Prefix and Suffix Cases in $\sum_i \mathcal{O}(m^2+N_i) = \mathcal{O}(nm^2+N)$ total time (Lemma 48).

Now, to solve the decision version of the problem, we solve the Anchor Cases with the use of the precomputed AP_{i-1} and AS_{i+1} for each $i \in [2\,..\,n-1]$ in $\mathcal{O}(m^2\sqrt{\log m} + N_i \log\log m)$ time (Subsection 2.2.5), which gives $\mathcal{O}(nm^2\sqrt{\log m} + N\log\log m)$ total time for the whole algorithm.

For the reporting version we proceed differently to obtain an online algorithm; note that this is possible because we can proceed without AS_i (see Figure 2.6). We thus consider one segment $\widetilde{T}[i]$ at the time, for each $i \in [1\,..\,n]$, and do the following. We compute 1-AP_i, as the union of three sets obtained from:

- The Suffix Case for $\widetilde{T}[i]$, computed in $\mathcal{O}(m^2+N_i)$ time (Lemma 48).
- Standard APE with 1-AP_{i-1} as the input bit vector, computed in $\mathcal{O}(m^2+N_i)$ time (Lemma 9).
- Anchor Case computed from AP_{i-1} in $\mathcal{O}((m^2+N_i)\log N_i)$ (Subsection 2.2.5) or $\mathcal{O}(m^3+N_i)$ time (Lemma 46).

If $N_i \geq m^3$, the algorithm of Lemma 46 works in the optimal $\mathcal{O}(m^3+N_i)=\mathcal{O}(N_i)$ time, hence we can assume that the $\mathcal{O}((m^2+N_i)\log N_i)$-time algorithm is only used when $N_i \leq m^3$, and thus it runs in $\mathcal{O}((m^2+N_i)\log m)$ time. Therefore over all i the computations require $\mathcal{O}((nm^2+N)\log m)$ or $\mathcal{O}(nm^3+N)$ total time. For every segment i we can also check whether an active prefix from 1-AP_{i-1} or from AP_{i-1} can be completed to a full match in $\widetilde{T}[i]$ using the algorithms of Grossi et al. from [109] and Prefix Case, respectively, in $\mathcal{O}(m^2+N_i)$ extra time.

By summing up all these we obtain Theorem 39.

2.2.9 1-MISMATCH EDSM

In this section, we give an alternative to the construction presented in Subsection 2.2.5, in the case of 1-MISMATCH EDSM. We do so by finding matches in a tree containing both suffixes of P and elements from the segment $\widetilde{T}[i]$, as well as modified versions of those strings. The number of additional strings is bounded by using the *heavy-light decomposition* of Sleator and Tarjan [196]. The construction is directly inspired by the one presented by Thankachan et al. in [203], which is itself inspired by the *k-errata tree* construction introduced by Cole et al. in [64] for indexing with errors. We give an algorithm to find all occurrences of P in $\widetilde{T}$ with 1 mismatch by computing sets 1-AP_i under Hamming distance, which, combined with the previously developed techniques, results in solving the 1-MISMATCH EDSM problem in $\mathcal{O}(nm^2+N\log m)$ time.

Let us start with the following basic definition.

Definition 49 ([196])**.** Let $\widetilde{\mathcal{T}}$ be a rooted tree. The *heavy path* of $\widetilde{\mathcal{T}}$ is the path that starts at the root and at each node descends to the child (called *heavy node*) with the largest number of leaf nodes in its subtree (ties are broken arbitrarily). The *heavy-light decomposition* of $\widetilde{\mathcal{T}}$ is defined recursively as a union of the heavy path of $\widetilde{\mathcal{T}}$ and the heavy path decompositions of the off-path subtrees of the heavy path. The nodes that are not heavy nodes are called *light nodes* (the root of $\widetilde{\mathcal{T}}$ is always a light node). An edge on a heavy path is called *heavy*; and the other edges are called *light*.

A crucial well-known property following from Definition 49 is that any root-to-leaf path crosses $\mathcal{O}(\log|\widetilde{\mathcal{T}}|)$ heavy paths. Each light edge on a path from the root decreases the size of the descending subtree by at least half. Thus the number of light edges on a path from any node to the root is $\mathcal{O}(\log|\widetilde{\mathcal{T}}|)$.

We use the above properties to efficiently construct a tree $\widetilde{\mathcal{T}}_1(P,\widetilde{T}[i])$ (for a given ED text $\widetilde{T}[1..n]$ of size N, a pattern $P[1..m]$ and an index $1\leq i\leq n$ with $||\widetilde{T}[i]||=N_i$) in the following three steps (inspect also Figure 2.9):

Step 1 We construct the compacted trie containing the strings in $\widetilde{T}[i]$ and suffixes $P[j+1..m]$ of P for each $j\in AP_{i-1}$. We call this set of suffixes of P $act_{i-1}(P)$. We also add labels $(\iota(X),\#)$ to each node in the tree corresponding to a string X in $act_{i-1}(P)\cup\widetilde{T}[i]$, where $\iota(X)$ is a pointer to X and # is a special label. This takes $\mathcal{O}(m+N_i)$ time and space [91] (we add the suffixes of P in $\mathcal{O}(m)$ total time by constructing the suffix tree of P and truncating the superfluous suffixes). We

call $\widetilde{\mathcal{T}}_0(P,\widetilde{T}[i])$ the tree we obtain from this step. In the next steps, $\widetilde{\mathcal{T}}_0(P,\widetilde{T}[i])$ will be extended with new nodes and labels to obtain $\widetilde{\mathcal{T}}_1(P,\widetilde{T}[i])$.

Step 2 We compute a heavy-light decomposition [196] of $\widetilde{\mathcal{T}}_0(P,\widetilde{T}[i])$, which takes time linear in its size, namely $\mathcal{O}(m+N_i)$.

Step 3 For each light node u of $\widetilde{\mathcal{T}}_0(P,\widetilde{T}[i])$ let u' be the leaf on the heavy path starting at u. Leaf u' corresponds to a string X, and for each labeled descendant v of u outside of the heavy path $u \dots u'$, if Y is the string corresponding to v, we compute $p = 1 + \mathrm{LCP}(X,Y)$ (in $\mathcal{O}(1)$ time after linear-time preprocessing of the tree, see Theorem 1) and add to $\widetilde{\mathcal{T}}_1(P,\widetilde{T}[i])$ the string obtained from Y by replacing $Y[p]$ with $X[p]$, with a label $(\iota(Y),p)$ (a given node can store multiple labels). Intuitively, p is the position of a *mismatch* between (a prefix of) X and (a prefix of) Y. Since the tree $\widetilde{\mathcal{T}}_0(P,\widetilde{T}[i])$ has $\mathcal{O}(m+N_i)$ nodes and each of them has $\mathcal{O}(\log(m+N_i))$ light node ancestors, there are no more than $\mathcal{O}((m+N_i)\log(m+N_i))$ additional nodes and labels. Also the construction of new nodes can be done each time in $\mathcal{O}(1)$ time, because we in fact just copy a subtree of a light node and merge it with the subtree of its heavy sibling.

We have thus arrived at the following lemma.

Lemma 50. *The construction of $\widetilde{\mathcal{T}}_1(P,\widetilde{T}[i])$ takes $\mathcal{O}((m+N_i)\log(m+N_i))$ time and space.*

We now prove that the tree $\widetilde{\mathcal{T}}_1(P,\widetilde{T}[i])$ satisfies the following property.

Lemma 51. *Let $X \in act_{i-1}(P)$. A string $Y \in \widetilde{T}[i]$ is at Hamming distance at most 1 from a prefix of X having length $|Y|$ if and only if $\widetilde{\mathcal{T}}_1(P,\widetilde{T}[i])$ contains two nodes u, v respectively labeled by $(\iota(X),p)$ and $(\iota(Y),p')$, for some $p,p' \in \mathbb{N}\cup\{\#\}$, such that u is a descendant of v, and one of the following is satisfied:*

- $p = p' \in \mathbb{N}$
- $p = \#$ *or* $p' = \#$.

Proof. For the forward implication, if Y is a prefix of X then the claim is trivial since $\widetilde{\mathcal{T}}_0(P,\widetilde{T}[i])$ contains nodes with labels $(\iota(X),\#)$ and $(\iota(Y),\#)$, and thus the first node is a descendant of the second one. Now, we assume that Y has one mismatch with $X' = X[1..|Y|]$ at a position p. Let u, v be nodes in $\widetilde{\mathcal{T}}_0(P,\widetilde{T}[i])$ respectively corresponding to X and Y, and let w be their lowest common light ancestor. Let Z be the string corresponding to the leaf on the heavy path starting at w. Since X and Y have a mismatch at position p, at least one of them has a mismatch with Z at position p and there are no mismatches to the left of p. Indeed, suppose towards a contradiction that there exists some $p' < p$ such that $Z[p'] \neq X[p'] (= Y[p'])$: then the node corresponding to $X[1..p'] (= Y[1..p'])$ would not be on the heavy path corresponding to Z, but would be a common ancestor of u and v, and thus w would not be the lowest common light ancestor of u and v, a contradiction.

Assume first that $X[p] \neq Z[p]$. Then, there is a node with a label $(\iota(X),p)$ in the tree, which is a descendant of either v, having label $(\iota(Y),\#)$ (if $Y[p] = Z[p]$), or a node

having label $(\iota(Y), p)$ (if $Y[p] \neq Z[p]$), because we assumed that X and Y do not have any other mismatch. Finally, if $X[p] = Z[p]$, then $Y[p] \neq Z[p]$ and the node with label $(\iota(Y), p)$ is an ancestor of u, having label $(\iota(X), \#)$.

To prove the reverse implication, let us assume that the consequences are satisfied. Let u be the node whose label contains $(\iota(X), p)$ and v the node whose label contains $(\iota(Y), p')$. We first assume $p = p' \in \mathbb{N}$. Note that, by the construction of $\widetilde{\mathcal{T}}_1(P, \widetilde{T}[i])$, the node u (resp. v) corresponds to a string obtained by one letter modification on X (resp. on Y) at the same position p. We denote the resulting string $\widehat{X}$ (resp $\widehat{Y}$). Since v is an ancestor of u in $\widetilde{\mathcal{T}}_1(P, \widetilde{T}[i])$, $\widehat{Y}$ is a prefix of $\widehat{X}$. But this exactly means that Y is at Hamming distance 1 to the length-$|Y|$ prefix of X (or Hamming distance 0 if both replacements replaced the same letter). If the second condition is satisfied, namely if $p = \#$ or $p' = \#$, then it means that one replacement in Y gives $\widehat{Y}$ which is a prefix of X, or that Y is a prefix of $\widehat{X}$, which is one replacement away from X, therefore we have the claimed result. □

We next formalize (Algorithm 1) how to find nodes satisfying one of the conditions from Lemma 51 and deduce the approximate active prefixes corresponding to the Anchor Case for segment $\widetilde{T}[i]$. Let v_1 `OR` v_2 denote a bit-wise OR of two vectors, and $v_1 \oplus x$ denote vector v_1 shifted by x positions to the right (the first x positions are set to 0).

Algorithm 1 Search($\widetilde{\mathcal{T}}$)

1: **Global variables**: the set $act_{i-1}(P)$, a segment $\widetilde{T}[i]$, and bit vectors $V_{\#}$, V_1, …, V_m, V_{ANY}, V_{res} all of size $|P|+1$, and initially set to all 0's
2: **Input**: $\widetilde{\mathcal{T}}$ - a subtree of $\widetilde{\mathcal{T}}_1(P, \widetilde{T}[i])$ with root r
3: **Output**: represented by global bit vector V_{res}
4: **for** each label $(\iota(X), p)$ with $X \in \widetilde{T}[i]$ on r **do**
5: set $V_p[|X|]$ and $V_{ANY}[|X|]$ to 1
6: **end for**
7: **for** each label $(\iota(X), p)$ with $X \in act_{i-1}(P)$ on r **do**
8: **if** $p = \#$ **then**
9: update V_{res} to V_{res} `OR` $(V_{ANY} \oplus (m - |X|))$.
10: **else** update V_{res} to V_{res} `OR` $((V_p$ `OR` $V_{\#}) \oplus (m - |X|))$
11: **end if**
12: **end for**
13: **for** each $\widetilde{\mathcal{T}}'$ a subtree of $\widetilde{\mathcal{T}}$ rooted at a child of r **do**
14: run Search($\widetilde{\mathcal{T}}'$)
15: **end for**
16: **for** each label $(\iota(X), p)$ with $X \in \widetilde{T}[i]$ on r **do**
17: set $V_p[|X|]$ and $V_{ANY}[|X|]$ to 0
18: **end for**

Proposition 52. *Algorithm 1 with input $\widetilde{\mathcal{T}}_1(P,\widetilde{T}[i])$ returns V_{res} such that $V_{res}[p]=1$ if and only if p is an element of 1-AP_i corresponding to the Anchor Case for segment $\widetilde{T}[i]$. Algorithm 1 runs in $\mathcal{O}((m+N_i)\log(m+N_i)+m^2)$ time.*

Proof. We first need the following remark: if $P[1\,..\,k]$ extends into $P[1\,..\,k']$ in $\widetilde{T}[i]$, that means that some $Y\in\widetilde{T}[i]$ is at Hamming distance 1 from the prefix $P[k+1\,..\,k']$ of $P[k+1\,..\,|P|]$. Therefore, we are looking for the pairs described in Lemma 51. We show that $V_{res}[k']=1$ after the end of the procedure if and only if there is a pair of nodes u,v in $\widetilde{\mathcal{T}}_1(P,\widetilde{T}[i])$ satisfying the conditions of Lemma 51 for $X=P[k+1\,..\,|P|]$, $Y\in\widetilde{T}[i]$, and $|Y|=k'-k$.

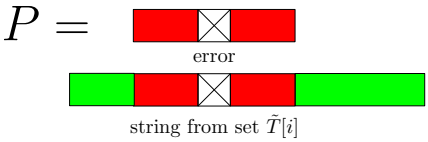
Let us assume the existence of such a pair (u,v). Since the tree is traversed in a DFS [71], the node v (with a label $(\iota(Y),p')$, $p'\in\mathbb{N}\cup\{\#\}$) is traversed before u, which is its descendant; and at this moment, $V_{p'}[|Y|]$ is set to 1, as well as $V_{ANY}[|Y|]$. Since u is a descendant of v, the vectors are not modified at position $|Y|$ until u is visited: that would mean that v has a strict descendant representing a string of the same length as the string represented by v. When the label $(\iota(X),p)$ for $X\in act_{i-1}(P)$ is visited on u, we set the position $(m-|X|)+|Y|=k'$ of V_{res} to 1 if $V_p[|Y|]=1$ or $V_\#[|Y|]=1$ (which happens if $p=p'\in\mathbb{N}\cup\{\#\}$ or if $p'=\#$) and when $p=\#$ if $V_{ANY}[|Y|]=1$.

Vice versa, if after the processing one has $V_{res}[k']=1$, this means that at some point in the DFS a node u having a label $(\iota(X),p)$ with $X\in act_{i-1}(P)$ and $p\in\mathbb{N}\cup\{\#\}$ was visited, and that at this point, for $k=m-|X|$, one had $V_{ANY}[k'-k]=1$ or $V_{p'}[k'-k]=1$ for (p,p') satisfying the conditions from Lemma 51. This one had to be set previously in the DFS at a node v having label $(\iota(Y),p')$ for $Y\in\widetilde{T}[i]$ with $|Y|=k'-k$. Finally, only an ancestor of u can be chosen as such v, because otherwise, from the DFS traversal order, the corresponding component of the vectors would have been set to 0. Now, the pair of nodes (u,v) satisfy the conditions of Lemma 51, and from our observations that means that there is an active prefix with 1 error of P having length k, extending up to $\widetilde{T}[i]$.

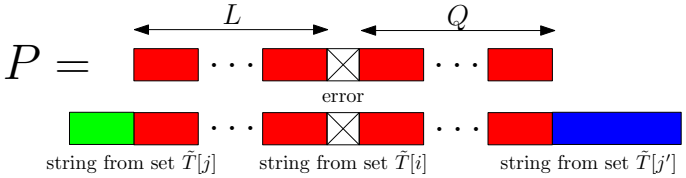
The running time follows from the fact that the algorithm visits only $\mathcal{O}((m+N_i)\log(m+N_i))$ labels by Lemma 50, and from the fact that the tree is traversed in a DFS. The analysis of each label consists in reading it and doing a constant number of bit modifications in the stored vectors, and, for $\mathcal{O}(m\log(m+N_i))$ of them (the one corresponding to a suffix of P), doing an `OR` operation which takes $\mathcal{O}(\frac{m}{\log(N+m)})$ time in the word RAM model. This gives us the required running time. □

Corollary 53. 1-*Mismatch EDSM can be solved in $\mathcal{O}(nm^2+N\log m)$ time.*

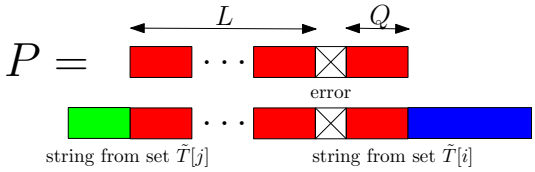
Proof. We proceed in the same way as in the reporting version of Subsection 2.2.8; the only difference is that, when $N_i\leq m^3$, to extend AP_{i-1} into 1-AP_i, instead of using the $\mathcal{O}((m^2+N_i)\log m)$-time algorithm, we use the one from Proposition 52. Due to this change, the algorithm runs in the desired time. Indeed, notice that when $N_i\leq m^3$, $\mathcal{O}((N_i+m)\log(m+N_i)+m^2)=\mathcal{O}(N_i\log m+m^2)$, and when $N_i\geq m^3$, $\mathcal{O}(m^3+N_i)=\mathcal{O}(N_i)$. The total time is thus $\mathcal{O}(nm^2+N\log m)$. □



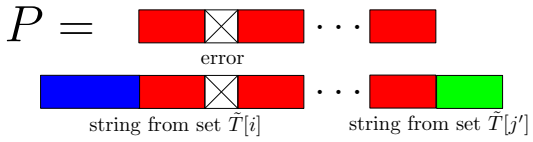
(a) Easy Case: $j = i = j'$.



(b) Anchor Case: $j \neq i, i \neq j'$.



(c) Prefix Case: $j \neq i, i = j'$.



(d) Suffix Case: $i = j, i \neq j'$.

Figure 2.7: Possible alignments of 1-error occurrences of P in $\widetilde{T}$. Each occurrence starts at segment $\widetilde{T}[j]$, ends at $\widetilde{T}[j']$, and the error occurs at $\widetilde{T}[i]$.

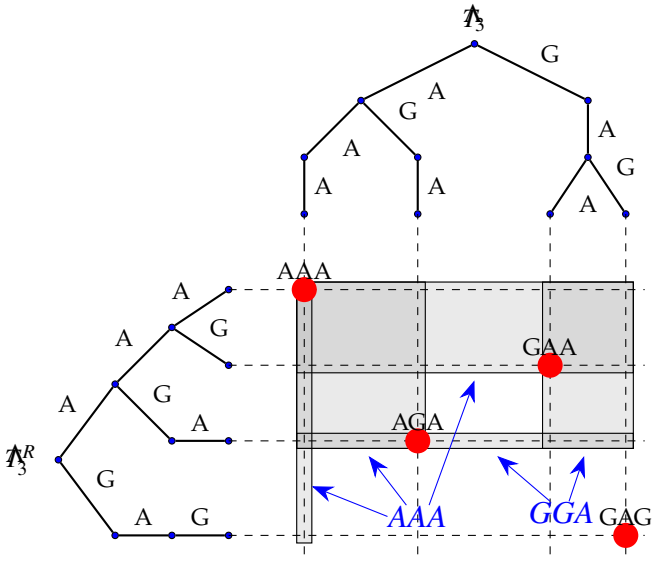


Figure 2.8: An example of points and rectangles (solid shapes) for the decision version of the Anchor Case with 1 mismatch. Here $P = GGAAAAGAGAGG$, $AP_{i-1} = \{1,2,4,7,8,9\}$, $AS_{i+1} = \{5,6,9,11,12\}$, $\mu = 3$, and $\widetilde{T}[i] = \{AAA, GGA\}$. $\mathcal{T}_3$ and $\widetilde{\mathcal{T}}_3^R$ are built for 4 strings: $P[2..4] = GAA$, $P[3..5] = AAA$, $P[8..10] = AGA$, $P[9..11] = GAG$; the 5 rectangles correspond to pairs $(\varepsilon, AA), (A, A), (AA, \varepsilon), (\varepsilon, AG), (G, A)$, namely, the pairs of prefixes and reversed suffixes of AAA and GGA (rectangle (GG, ε) does not exist as $\mathcal{T}_3$ contains no node GG).

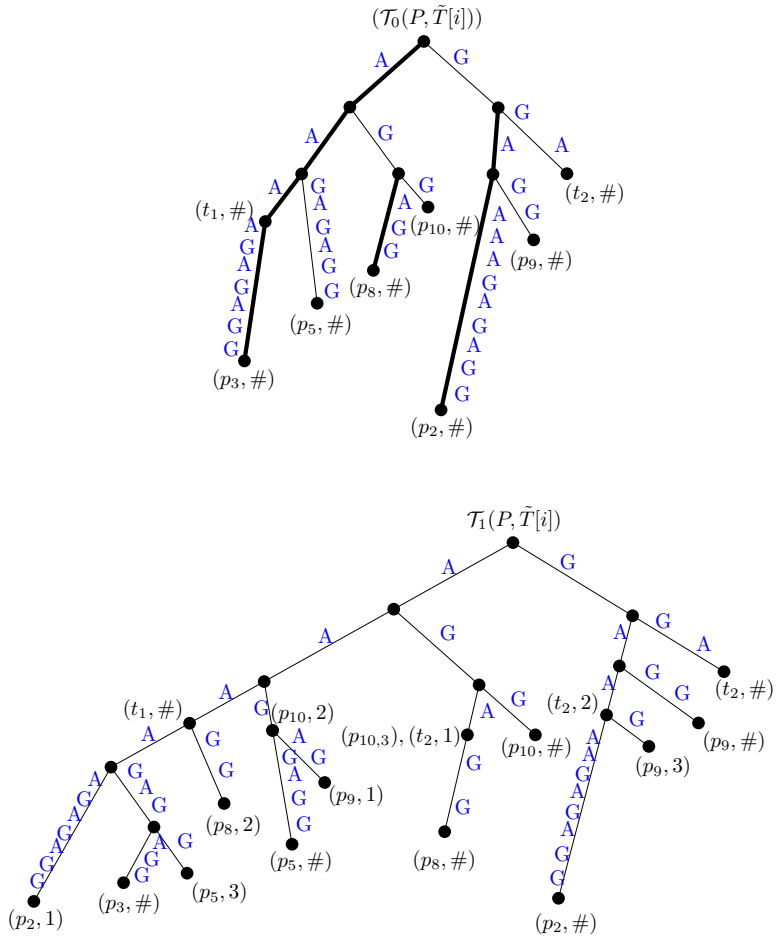


Figure 2.9: $\widetilde{\mathcal{T}}_0(P,\tilde{T}[i])$ (top) and $\widetilde{\mathcal{T}}_1(P,\tilde{T}[i])$ (bottom) for the example from Figure 2.8 ($P = GGAAAAGAGAGG$, $AP_{i-1} = \{1,2,4,7,8,9\}$, $\tilde{T}[i] = \{AAA, GGA\}$) with labels $(p_j = \iota(P[j..m]), t_j = \iota(\tilde{T}[i][j]))$ and heavy paths.

2.3 CONCLUSION AND FUTURE WORK

In this Chapter, we discussed *exact* and *approximate* pattern matching on degenerate texts.

In Section 2.1, for almost all combinations of variable strings X and Y, we showed that problem PVART(X,Y) admits either a truly sub-quadratic upper bound or a quadratic lower bound conditioned on SETH. Notably, two cases are left open: when $X =$ SOLID and $Y = GD$, and when $X =$ SOLID and $Y = F$. In these cases, our algorithms, although sub-quadratic in some cases, are quadratic in the worst case and no lower bound is known. These problems correspond to the yellow cells in the first row of Table 2.1. Notice that the other yellow cell, that is $X =$ 1-D and $Y =$ 1-D, is a completely solved case for which all bounds are known.

Other open problems are the following:

- The bounds of Theorem 19 and 20 are only sub-quadratic when $N/n = \omega(1)$; whether there exist sub-quadratic algorithms for the case $N/n = \mathcal{O}(1)$ is an open problem, as no conditional lower bound for PVART(SOLID, GD) or PVART(SOLID, F) is known yet.
- In both Theorems 21 and 22 we assume that $k = \omega(\log n)$; it remains open whether there exist sub-quadratic algorithms when $k = O(\log n)$.

In Section 2.2, we showed that 1-EDSM can be solved in $\mathcal{O}((nm^2+N)\log m)$ or $\mathcal{O}(nm^3+N)$ time under edit distance. For the decision version of the problem, we presented a faster $\mathcal{O}(nm^2\sqrt{\log m}+N\log\log m)$-time algorithm. We also showed that 1-EDSM can be solved in $\mathcal{O}(nm^2+N\log m)$ time under Hamming distance.

While our techniques (Sections 2.2.3 and 2.2.9) seem to generalize relatively easily to k errors, they would incur some exponential factor with respect to k.

We leave the following basic questions open:

1. Can we design a combinatorial $\mathcal{O}(nm^2+N)$-time algorithm for 1-EDSM under edit or Hamming distance?
2. Can our techniques be efficiently generalized for $k > 1$ errors or mismatches?
3. Can our Hamming distance improvement for 1 mismatch (Subsection 2.2.9) be extended to edit distance for 1 error?
4. Can one more generally adapt the results from Section 2.1 to the approximate case, and draw a taxonomy of *approximate* pattern matching on degenerate texts?

3

COMPARISON

3.1 INTRODUCTION

This chapter is based on [95, 96, 98].

Our main goal here is to give the first algorithms and lower bounds for comparing two pangenomes represented by two *ED* strings. We consider the most basic notion of matching, namely, to decide whether two *ED* strings, each encoding a language, have a nonempty intersection. Like with standard strings, algorithms for pairwise *ED* string comparison can serve as computational primitives for many analysis tasks (e.g., phylogeny reconstruction); lower bounds for pairwise *ED* string comparison can serve as meaningful lower bounds for the comparison of more powerful pangenome representations such as, for instance, variation graphs [51].

We next define the main problem in scope; inspect also Figure 3.1 for an example.

Problem 9. ED STRING INTERSECTION (EDSI)
Input: Two *ED* strings, $\widetilde{T}_1$ of length n_1, cardinality B_1 and size N_1, and $\widetilde{T}_2$ of length n_2, cardinality B_2 and size N_2.
Output: YES if $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$ have a nonempty intersection, NO otherwise.

ED strings

$$T_1 = \left\{\texttt{A}\right\}\cdot\left\{\begin{matrix}\texttt{G}\\ \varepsilon\end{matrix}\right\}\cdot\left\{\begin{matrix}\texttt{C}\\ \texttt{GCT}\\ \varepsilon\end{matrix}\right\}\cdot\left\{\begin{matrix}\texttt{TC}\\ \texttt{CTC}\end{matrix}\right\}$$

$$T_2 = \left\{\texttt{A}\right\}\cdot\left\{\begin{matrix}\texttt{T}\\ \texttt{GCT}\\ \texttt{CC}\end{matrix}\right\}\cdot\left\{\begin{matrix}\texttt{C}\\ \texttt{TC}\end{matrix}\right\}$$

Parameters

$n_1 = 4$
$B_1 = 8$
$N_1 = 13$
$n_2 = 3$
$B_2 = 6$
$N_2 = 10$

$\mathcal{L}(\mathbf{T_1}) \cap \mathcal{L}(\mathbf{T_2})$

AGCTC
ACCTC
AGCTTC
ATC

Figure 3.1: An example of two *ED* strings $\widetilde{T}_1$ and $\widetilde{T}_2$ with their parameters and the intersection of their languages. In this instance, we see that $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$ have a nonempty intersection.

In this chapter, we will use classical techniques from formal languages and automata theory:

Definition 54 (NFA)**.** *A nondeterministic finite automaton* (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states; Σ is an alphabet; $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is a transition function, where $\mathcal{P}(Q)$ is the power set of Q; $q_0 \in Q$ is the starting state; and $F \subseteq Q$ is the set of accepting states.

Using the folklore product automaton construction, one can check whether two NFA have a nonempty intersection in $\mathcal{O}(N_1 \cdot N_2)$ time, where N_1 and N_2 are the sizes of the two NFA [153].

We use a different, compacted representation of automata, which in some special cases allows a more efficient algorithm for computing and representing the intersection. In general, we call *intersection graph* (identifying states to nodes and transitions to directed labeled edges) the underlying graph of any automaton representing $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$.

Our Results We assume that the *ED* strings are over an integer alphabet $[1\,..\,(N_1+N_2)^{\mathcal{O}(1)}]$. We make the following specific contributions:

1. In Subsection 3.2.1, we give several conditional lower bounds. In particular, we show that there is no $\mathcal{O}((N_1N_2)^{1-\varepsilon})$-time algorithm, thus no $\mathcal{O}((N_1B_2+N_2B_1)^{1-\varepsilon})$-time algorithm and no $\mathcal{O}((N_1n_2+N_2n_1)^{1-\varepsilon})$-time algorithm, for any constant $\varepsilon > 0$, for EDSI even when $\widetilde{T}_1$ and $\widetilde{T}_2$ are over a binary alphabet, unless the Strong Exponential-Time Hypothesis (SETH) [50] is false.

2. In Subsection 3.2.2, we present other conditional lower bounds. In particular, we show that there is no combinatorial $\mathcal{O}((N_1+N_2)^{1.2-\varepsilon} f(n_1,n_2))$-time algorithm, for any constant $\varepsilon > 0$ and any function f, for EDSI even when $\widetilde{T}_1$ and $\widetilde{T}_2$ are over a binary alphabet, unless the BMM conjecture (Conjecture 13 [7]) is false.

3. In Section 3.3, we show an $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$-time algorithm for outputting a compact representation of the intersection language of two unary *ED* strings. In the case when $\widetilde{T}_1$ and $\widetilde{T}_2$ are given in a compact representation, we show that the problem is NP-complete.

4. In Subsection 3.4.1, we show an $\mathcal{O}(N_1B_2 + N_2B_1)$-time algorithm for EDSI, by introducing their *intersection graph*.

5. In Subsection 3.4.2, we show an $\widetilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$-time algorithm for EDSI, where ω is the matrix multiplication exponent.

6. We show, in Section 3.5, Section 3.7, Section 3.6, and Section 3.8 that the techniques we develop here have many applications, both theoretical and practical.

7. In Section 3.9 we experimentally evaluate the efficiency of our algorithms.

Related Work Apart from its applications to pangenome comparison, EDSI is interesting theoretically on its own as a special case of regular expression (regex) matching. Regex is a basic notion in formal languages and automata theory. Regular expressions are commonly used in practical applications to define search patterns. Regex matching and membership testing are widely used as computational primitives in programming languages and text processing utilities (e.g., the widely-used agrep). The classic algorithm for solving these problems constructs and simulates an NFA corresponding to the regex, which gives an $\mathcal{O}(MN)$ running time, where M is the length of the pattern and N is the length of the text. Unfortunately, significantly faster solutions are unknown and unlikely [32]. However, much faster algorithms exist for many special cases of the problem: dictionary matching; wildcard matching; subset matching; and the word break problem; see [32] and references therein.

Special cases of EDSI have also been studied. First, let us consider the case when both $\widetilde{T}_1$ and $\widetilde{T}_2$ are 1-*D* strings. In this case, the problem is trivial: EDSI has a positive answer if and only if for every i, $\widetilde{T}_1[i] \cap \widetilde{T}_2[i]$ is nonempty. Alzamel et al. [15, 16] studied the case when $\widetilde{T}_1$ and $\widetilde{T}_2$ are *GD* strings: in that case, they showed that deciding if $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$ have a nonempty intersection can be done in $\mathcal{O}(N_1+N_2)$ time. If $\widetilde{T}_2$ is a standard string, i.e., an *ED* string with $B_2 = n_2 = 1$, then we can resort to the results of Bernardini et al. [43] for *ED* string matching. In particular: there is no combinatorial algorithm for EDSI working in $\mathcal{O}(n_1 N_2^{1.5-\varepsilon} + N_1)$ time unless the BMM conjecture is false; and we can solve EDSI in $\widetilde{\mathcal{O}}(n_1 N_2^{\omega-1} + N_1)$ time. Moreover, Gawrychowski et al. [102] provided a systematic study of the complexity of degenerate string comparison under different notions of matching: Cartesian tree matching; order-preserving matching; and parameterized matching.

3.2 CONDITIONAL LOWER BOUNDS

In this section, we show several conditional lower bounds for the EDSI problem. Bounds in the first group (see Section 3.2.1) are based on OVH (Conjecture 11) [50], ; the second group of bounds (see Section 3.2.2) is based, instead, on the Boolean Matrix Multiplication (BMM) conjecture (Conjecture 13) [7].

3.2.1 LOWER BOUNDS BASED ON SETH

We are going to reduce the ORTHOGONAL VECTORS problem (Problem 10) to the EDSI problem.

We show the following reduction.

Theorem 55. *Given two sets $X = \{x_1, \ldots, x_k\}$, $Y = \{y_1, \ldots, y_k\}$ of k binary vectors of length d, we can construct in linear time two ED strings $\widetilde{T}_1$ and $\widetilde{T}_2$ over a binary alphabet such that:*

- *$\widetilde{T}_1$ has length, cardinality, and size $\Theta(kd)$;*
- *$\widetilde{T}_2$ has length $\Theta(\log k)$, cardinality $\Theta(k)$ and size $\Theta(kd)$; and*
- *There is a pair $(x_a, y_b) \in X \times Y$ of orthogonal vectors if and only if $\widetilde{T}_1$ and $\widetilde{T}_2$ have a nonempty intersection.*

Proof. Let $u_i = 1^d - x_i$ for all $i \in [1..k]$, where 1^d is the d-dimensional all-1 vector. We construct $\widetilde{T}_1$ and $\widetilde{T}_2$ as follows (see Example 56):

$$\widetilde{T}_1 = \prod_{i=1}^{k} \prod_{j=1}^{d} \{0, u_i[j]\}, \qquad \widetilde{T}_2 = \prod_{i=0}^{\lfloor \log_2 k \rfloor} \left\{0^{d \cdot 2^i}, \varepsilon\right\} \cdot Y \cdot \prod_{i=0}^{\lfloor \log_2 k \rfloor} \left\{0^{d \cdot 2^i}, \varepsilon\right\}.$$

We now show that $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$ have a nonempty intersection if and only if there exists a pair of orthogonal vectors $(x_a, y_b) \in X \times Y$.

- Suppose that x_a and y_b are orthogonal. Then for all $j \in [1..d]$, $y_b[j] \in \{0, u_a[j]\}$ and hence $y_b \in \prod_j \{0, u_a[j]\}$. It follows that

$$0^{(a-1)d} y_b 0^{(k-a)d} \in 0^{(a-1)d} \prod_j \{0, u_a[j]\}\, 0^{(k-a)d} \subseteq \mathcal{L}(\widetilde{T}_1).$$

  By decomposing $a - 1 = \sum_{i \in S_{a-1}} 2^i$ and $k - a = \sum_{i \in S_{k-a}} 2^i$, where for any integer p, the set S_p contains the positions with a 1 in the binary representation of p, we find that

$$0^{(a-1)d} y_b 0^{(k-a)d} \in \prod_{i \in S_{a-1}} 0^{d \cdot 2^i} \cdot Y \cdot \prod_{i \in S_{k-a}} 0^{d \cdot 2^i} \subseteq \mathcal{L}(\widetilde{T}_2).$$

  We conclude that $0^{(a-1)d} y_b 0^{(k-a)d} \in \mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$.

- Conversely, suppose that $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$ have a nonempty intersection and consider a string $S \in \mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$. Let y_b be the vector from Y which is chosen in $\widetilde{T}_2$ when constructing S. The strings in the sets of $\widetilde{T}_2$ all have length divisible by d. Thus y_b starts at an index $(a-1)d+1$ of string S for some integer a. Since $S \in \mathcal{L}(\widetilde{T}_1)$, we have $y_b \in \prod_{j=1}^{d} \{0, u_a[j]\}$. This implies that x_a and y_b are orthogonal.

Therefore, solving the orthogonal vectors problem for X, Y is equivalent to checking whether $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$ have a nonempty intersection. □

Example 56. Let $k = 3$, $d = 3$, $X = \{x_1 = 010,\ x_2 = 100,\ x_3 = 011\}$ and $Y = \{y_1 = 001,\ y_2 = 010\ , y_3 = 110\ \}$.

We have that

$$\widetilde{T}_1 = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \{0\} \cdot \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \{0\} \cdot \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \{0\} \cdot \{0\}$$

and

$$\widetilde{T}_2 = \begin{Bmatrix} 000 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 000000 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 001 \\ 010 \\ 110 \end{Bmatrix} \cdot \begin{Bmatrix} 000 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 000000 \\ \varepsilon \end{Bmatrix}$$

.

One can observe that each string from $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ corresponds to a pair of orthogonal vectors from X and Y. For example, the string 000010000 is in $\mathcal{L}(\widetilde{T}_2)$ because $y_2 = 010 \in Y$. Since the vector $x_2 = 100 \in X$ is orthogonal to y_2, one also has $000010000 \in \mathcal{L}(\widetilde{T}_1)$. This is because the 3 middle segments of $\widetilde{T}_1$ are constructed to encode any vector which is orthogonal to x_2.

Note that when $d = \Theta(\log k)$, the length n_1, the cardinality B_1 and the size N_1 of $\widetilde{T}_1$ are $\mathcal{O}(k \log k)$, whereas $\widetilde{T}_2$ has length $n_2 = \mathcal{O}(\log k)$, cardinality $B_2 = \mathcal{O}(k)$ and size $N_2 = \mathcal{O}(k \log k)$. Moreover, both ED strings are over a binary alphabet $\Sigma = \{0,1\}$. This implies various hardness results for EDSI. For example, we can see that, for any constant $\varepsilon > 0$, and an alphabet Σ of size at least 2 the problem cannot be solved in

$$\mathcal{O}\left((N_1 + N_2 + n_1 + n_2)^{2-\varepsilon} \cdot \mathrm{poly}(n_2)\right)$$

time, conditional on the OV conjecture. By using the fact that $n_1 \leq B_1 \leq N_1$ and $n_2 \leq B_2 \leq N_2$, we obtain the following bounds.

Corollary 57. For any constant $\varepsilon > 0$, there exists no

- $\mathcal{O}((N_1 N_2)^{1-\varepsilon})$-time
- $\mathcal{O}((N_1 B_2 + N_2 B_1)^{1-\varepsilon})$-time
- $\mathcal{O}((N_1 n_2 + N_2 n_1)^{1-\varepsilon})$-time

algorithm for the EDSI problem, unless the OV conjecture is false.

3.2.2 COMBINATORIAL LOWER BOUNDS BASED ON THE BMM CONJECTURE

In this subsection, we give a reduction from TD (Problem 2) to apply the BMM conjecture (Conjecture 13) and prove combinatorial lower bounds on EDSI. Our reduction from TD to EDSI is based on the construction of Bernardini et al. from [43] for ED string matching.

Corollary 58. If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1 + N_2)^{1.2-\varepsilon} f(n_1, n_2))$ time, for any constant $\varepsilon > 0$ and any function f, then there exists a truly subcubic combinatorial algorithm for TD.

Proof. Let D be a positive integer and let A, B, and C be three $D \times D$ Boolean matrices. Further let $s \leq D$ be a positive integer to be set later. In the rest of the proof, we can assume that s divides D, up to adding α rows and columns containing only 0's to all three matrices, where α is the smallest non-negative representative of the equivalence class $-D \bmod s$.

Let us first construct an ED string $\widetilde{T}_1 = X_1 X_2 X_3$ over a large alphabet with $n_1 = 3$, where each X_p, $p \in [1..n_1]$, contains a string for each occurrence of value 1 in A, B and C, respectively. Below i iterates over $[1..D]$, j and k over $[0..\frac{D}{s} - 1]$, and x and y iterate over $[1..s]$. Moreover, $x, y \in [1..s]$, v_i for $i \in [1..D]$, and a, \$ are all distinct letters.

- If $A[i..x\cdot\frac{D}{s}+j]=1$, then X_1 contains the string $v_i x a^j$;
- If $B[x\cdot\frac{D}{s}+j, y\cdot\frac{D}{s}+k]=1$, then X_2 contains the string $a^{\frac{D}{s}-j}x\$\$ya^{\frac{D}{s}-k}$;
- If $C[y\cdot\frac{D}{s}+k,i]=1$, then X_3 contains the string $a^k y v_i$;

The length of each string in each X_p is $\mathcal{O}(D/s)$ and the total number B_1 of strings is up to $3D^2$. Overall, $N_1=\mathcal{O}(D^3/s)$.

We construct an *ED* string $\widetilde{T}_2$ with $n_2=1$ containing the following strings:

$$P(i,x,y)=v_i x a^{\frac{D}{s}} x\$\$y a^{\frac{D}{s}} y v_i \quad \text{for every } x,y\in[1..s] \text{ and } i\in[1..D].$$

Each string has length $\mathcal{O}(D/s)$ and there are $B_2=Ds^2$ strings, so $N_2=\mathcal{O}(D^2 s)$.

We use the following fact.

Fact 59 ([43]). *$P(i,x,y)\in\mathcal{L}(\widetilde{T}_1)$ if and only if the following holds for some $j,k\in[1..D/s]$:*

$$A[i..x\cdot\frac{D}{s}+j]=B[x\cdot\frac{D}{s}+j,y\cdot\frac{D}{s}+k]=C[y\cdot\frac{D}{s}+k,i]=1.$$

We choose $s=\lfloor\sqrt{D}\rfloor$; then $N_1,N_2=\mathcal{O}(D^{2.5})$ and $n_1,n_2=\mathcal{O}(1)$. Then indeed an $\mathcal{O}((N_1+N_2)^{1.2-\varepsilon}f(n_1,n_2))$-time algorithm for EDSI would yield an $\mathcal{O}(D^{3-2.5\varepsilon})$-time algorithm for the TD problem.

Note also that even though the size of the alphabet used above is $\Theta(s+D)=\Theta(D)$, we can encode all letters by equal-length binary strings blowing N_1 and N_2 up only by a factor of $\Theta(\log D)$ and, hence, obtain the same lower bound for a binary alphabet. □

Both B_1 and B_2 in the reduction are $\mathcal{O}(D^2)$, thus an $\mathcal{O}((B_1+B_2)^{1.5-\varepsilon}f(n_1,n_2))$-time algorithm would yield an $\mathcal{O}(D^{3-2\varepsilon})$-time algorithm for TD; hence we obtain the following.

Corollary 60. *If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1^{1.2}+N_2^{1.2}+B_1^{1.5}+B_2^{1.5})^{1-\varepsilon}f(n_1,n_2))$ time, for any constant $\varepsilon>0$ and any function f, then there exists a truly subcubic combinatorial algorithm for TD.*

3.3 EDSI: THE UNARY CASE

An *ED* string is called *unary* if it is over an alphabet of size 1. In this special case, if both $\widetilde{T}_1$ and $\widetilde{T}_2$ are over the same alphabet $\Sigma=\{\mathrm{A}\}$, EDSI boils down to checking whether there exists any $b\geq 0$ such that A^b belongs to both $\mathcal{L}(\widetilde{T}_1)$ and $\mathcal{L}(\widetilde{T}_2)$.

Let $\widetilde{T}$ be a unary *ED* string of length n over alphabet $\Sigma=\{\mathrm{A}\}$. We define the *compact representation* $R(\widetilde{T})$ of $\widetilde{T}$ as the following sequence of sets of integers:

$$\forall i\in[1..n],\ R(\widetilde{T})[i]=\{b_{i,1},b_{i,2},\ldots,b_{i,B_i}\}\iff\widetilde{T}[i]=\{\mathrm{A}^{b_{i,1}},\mathrm{A}^{b_{i,2}},\ldots,\mathrm{A}^{b_{i,B_i}}\},$$

where $b_{i,j}\geq 0$ for all $i\in[1..n]$ and $j\in[1..B_i]$, the cardinality of $\widetilde{T}$ is $B=\sum_{i=1}^n B_i$, and its size is $N=N_\varepsilon+\sum_{i=1}^n\sum_{j=1}^{B_i}b_{i,j}$, where N_ε is the total number of empty strings in $\widetilde{T}$.

Theorem 61. If $\widetilde{T}_1$ and $\widetilde{T}_2$ are unary *ED* strings and each is given in a compact representation, the problem of deciding whether $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ is nonempty is NP-complete.

Proof. The problem is clearly in NP, as it is enough to guess a single element for each set in both $\widetilde{T}_1$ and $\widetilde{T}_2$, and then simply check if the sums match in linear time. We show the NP-hardness through a reduction from the SUBSET SUM problem, which takes n integers $b_1, b_2, \ldots, b_n$ and an integer c, and asks whether there exist $x_i \in \{0,1\}$, for all $i \in [1..n]$, such that $\sum_{i=1}^{n} x_i b_i = c$. SUBSET SUM is NP-complete [140] also for non-negative integers. For any instance of SUBSET SUM, we set $R(\widetilde{T}_1)[i] = \{b_i, 0\}$ for all $i \in [1..n]$, $n_2 = 1$ and $R(\widetilde{T}_2)[1] = \{c\}$. Then the answer to the SUBSET SUM instance is YES if and only if $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ is nonempty. □

In what follows, we provide an algorithm which runs in polynomial time in the size of the two unary *ED* strings when the latter are given uncompacted.

The set $\mathcal{L}(\widetilde{T})$ can be represented as a set $L(\widetilde{T}) \subset \mathbb{N}$ such that $\mathcal{L}(\widetilde{T}) = \{A^\ell : \ell \in L(\widetilde{T})\}$. The set $L(\widetilde{T})$ will be stored as a list (without repetitions). We will show how to efficiently compute $L(\widetilde{T}_1)$ and $L(\widetilde{T}_2)$. Then one can compute $L(\widetilde{T}_1) \cap L(\widetilde{T}_2)$ in $\mathcal{O}(N_1 + N_2)$ time, which allows, in particular, to check if $L(\widetilde{T}_1) \cap L(\widetilde{T}_2) = \emptyset$ (which is equivalent to $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2) = \emptyset$).

We show the computation for $L(\widetilde{T}_1)$. The workhorse is an algorithm from the following Lemma 62 that allows to compute the set $L(X_1X_2)$ of concatenation of two *ED* strings based on their sets $L(X_1), L(X_2)$.

Lemma 62. *Let X_1 and X_2 be ED strings. Given $L(X_1)$ and $L(X_2)$ such that $t_1 = \max L(X_1)$ and $t_2 = \max L(X_2)$, we can compute $L(X_1X_2)$ in $\mathcal{O}((t_1+t_2)\log(t_1+t_2))$ time.*

Proof. For two sets $A, B \subset \mathbb{N}$, by $A + B$ we denote the set $\{a + b : a \in A, b \in B\}$. We then have $L(X_1X_2) = L(X_1) + L(X_2)$. Fast Fourier Transform (FFT) [71] can be used directly to compute $L(X_1) + L(X_2)$ in $\mathcal{O}((t_1+t_2)\log(t_1+t_2))$ time. □

Lemma 63. *$L(\widetilde{T}_1)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1)$ time.*

Proof. We apply the recursive algorithm described in Algorithm 2 to $\widetilde{T}_1$.

Algorithm 2 Compute-$L(\widetilde{T}[1..k])$

```
if k = 1 then
    Compute L(T̃[1]) naïvely
else
    i ← ⌊k/2⌋
    L1 ←Compute-L(T̃[1..i])
    L2 ←Compute-L(T̃[i+1..k]
    return L1 + L2
end if
```

Let $N_{1,i} = \sum_{x \in L(\widetilde{T}_1)[i]} x$ and $t_{1,i} = \max L(\widetilde{T}_1[i])$ for $i \in [1..n_1]$. Obviously, $t_{1,i} \leq N_{1,i}$.

We analyze the complexity of the recursion by levels. For the bottom level, $L(\widetilde{T}_1[i])$ can be computed in $\mathcal{O}(N_{1,i})$ time for each $i \in [1..n_1]$, which sums up to $\mathcal{O}(N_1)$. For the remaining levels, we notice that $\max L(\widetilde{T}_1[i\ ddj]) = t_{1,i} + \cdots + t_{1,j}$. On each level, the fragments of $\widetilde{T}_1$ that are considered are disjoint. Thus, the complexity on each level via Lemma 62 is $\mathcal{O}((\sum_{i=1}^{n_1} t_{1,i}) \log(\sum_{i=1}^{n_1} t_{1,i})) = \mathcal{O}(N_1 \log N_1)$. The number of levels of recursion is $\mathcal{O}(\log n_1)$; the complexity follows. □

3

Theorem 64. If $\widetilde{T}_1$ and $\widetilde{T}_2$ are unary *ED* strings, then $L(\widetilde{T}_1) \cap L(\widetilde{T}_2)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ time.

Proof. We use Lemma 63 to compute $L(\widetilde{T}_1)$ and $L(\widetilde{T}_2)$ in the required complexity. Then $L(\widetilde{T}_1) \cap L(\widetilde{T}_2)$ can be computed via bucket sort [71]. □

3.4 EDSI: GENERAL CASE

In this section, we give an algorithm to solve EDSI for two general *ED* strings. We start with an introductory remark:

Assuming that the two *ED* strings, $\widetilde{T}_1$ and $\widetilde{T}_2$, of total size $N_1 + N_2$ are over an integer alphabet $[1..(N_1+N_2)^{\mathcal{O}(1)}]$, we can sort the suffixes of all strings in $\widetilde{T}_1[i]$, for all $i \in [1..n_1]$, and the suffixes of all strings in $\widetilde{T}_2[j]$, for all $j \in [1..n_2]$, in $\mathcal{O}(N_1 + N_2)$ time [91].

3.4.1 COMPACTED NFA INTERSECTION

In this section we show an algorithm for computing a representation of the intersection of the languages of two *ED* strings using techniques from formal languages and automata theory.

Definition 65 (Compacted NFA)**.** An *extended transition* is a transition function of the form $\delta^{ext} : Q \times \Sigma^* \to \mathcal{P}(Q)$, where Q is a finite set of states, Σ^* is the set of strings over alphabet Σ, and $\mathcal{P}(Q)$ is the power set of Q. A *compacted NFA* is an NFA in which we allow extended transitions. Such an NFA can also be represented by a standard (uncompacted) NFA, where each extended transition is subdivided into standard one-letter transitions (and ε-transitions), $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$. The states of the compacted NFA are called *explicit*, while the states obtained due to these subdivisions are called *implicit*.

Given a compacted NFA A with V explicit states and E extended transitions, we denote by V^u and E^u the number of states and transitions, respectively, of its uncompacted version A^u. Henceforth we assume that in the given NFA every state is reachable, and hence we have $V^u = \mathcal{O}(E^u)$ and $V = \mathcal{O}(E)$.

Lemma 66. *Given two compacted NFA A_1 and A_2, with V_1 and V_2 explicit states and E_1 and E_2 extended transitions, respectively, a compacted NFA representing the intersection of A_1 and A_2 with $\mathcal{O}(V_1^u V_2 + V_1 V_2^u)$ explicit states and $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$ extended transitions can be computed in $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$ time.*

Proof. We start by constructing an LCP data structure over the concatenation of all the labels of extended transitions of both NFA of total size $\mathcal{O}(E_1^u + E_2^u)$. It requires $\mathcal{O}(E_1^u + E_2^u)$-time preprocessing and allows answering LCP queries on any two substrings of such labels in $\mathcal{O}(1)$ time (Theorem 1).

We construct A, a compacted NFA representing the intersection of A_1 and A_2.

Every state of A is composed of a pair: an explicit state of one automaton and any explicit or implicit state of the other automaton (or equivalently a state of the uncompacted version of the automaton). Thus the total number of explicit states of A is $\mathcal{O}(V_1^u V_2 + V_1 V_2^u)$.

We need to compute the extended transitions of A. For a state (u, v) we check every string pair (T_1, T_2), where T_1 iterates over all extended transitions going out of u and T_2 iterates over all extended transitions going out of v (a transition going out of an implicit state is represented by a suffix of the transition it belongs to). For every pair (T_1, T_2) we ask an $\mathrm{LCP}(T_1, T_2)$ query. If $\mathrm{LCP}(T_1, T_2)$ is equal to one of $|T_1|$, $|T_2|$ (possibly both), we create an extended transition between (u, v) and the pair of states reachable through those transitions (if one of the transitions is strictly longer, we prune it to the right length, ending it at an implicit state of its input NFA). Otherwise such a transition does not lead to any explicit state of A and thus cannot be used to reach the accepting state; hence we ignore it.

Finally, the starting (resp. accepting) state of A corresponds to a pair of starting (resp. accepting) states of A_1 and A_2.

Since any pair representing an explicit state of A contains an explicit state of A_1 or A_2, the number of such transition pair checks (and hence also the number of the extended transitions of A) is $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$. Since each such check takes $\mathcal{O}(1)$ time, the construction complexity follows. Note that NFA A may contain unreachable states; such states can be removed afterwards in linear time. The algorithms' correctness follows from the observation that A^u is in fact the standard intersection automaton of A_1^u and A_2^u with some states, that do not belong to any path between the starting and the accepting states, removed. □

We next define the path-automaton of an *ED* string (inspect Figure 3.2 for an example).

Definition 67 (Path-automaton)**.** Let $\widetilde{T}$ be an *ED* string of length n, cardinality B, and size N. The *path-automaton* of $\widetilde{T}$ is the compacted NFA consisting of:

- $V = n + 1$ explicit states, numbered from 1 through $n + 1$. State 1 is the starting state and state $n + 1$ is the accepting state. State $i \in [2..n]$ is the state *in-between* $\widetilde{T}[i-1]$ and $\widetilde{T}[i]$.
- B_i extended transitions from state i to state $i + 1$ labeled with the strings in $\widetilde{T}[i]$, for all $i \in [1..n]$, where $E = B = \sum_i B_i$.

The *path-automaton* of $\widetilde{T}$ accepts exactly $\mathcal{L}(\widetilde{T})$. The uncompacted version of this path-automaton has $V^u = \mathcal{O}(N)$ states and $E^u = N$ transitions.

An example is constructed in Figure 3.2.

Lemma 66 thus implies the following result.

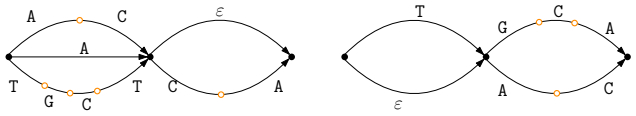


Figure 3.2: The path automatons A_1 and A_2 for *ED* strings $\widetilde{T}_1 = \left\{ \begin{matrix} \mathrm{A} \\ \mathrm{AC} \\ \mathrm{TGCT} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \varepsilon \\ \mathrm{CA} \end{matrix} \right\}$ and $\widetilde{T}_2 = \left\{ \begin{matrix} \varepsilon \\ \mathrm{T} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \mathrm{GCA} \\ \mathrm{AC} \end{matrix} \right\}$. The filled black nodes are explicit states, while the orange empty nodes are implicit states.

3

Corollary 68. *The compacted NFA representing the intersection of two path-automata with $\mathcal{O}(N_1n_2+N_2n_1)$ explicit states and $\mathcal{O}(N_1B_2+N_2B_1)$ extended transitions can be constructed in $\mathcal{O}(N_1B_2+N_2B_1)$ time.*

Theorem 69. EDSI can be solved in $\mathcal{O}(N_1B_2+N_2B_1)$ time. If the answer is YES, a witness can be reported within the same time complexity.

Proof. The path-automaton of an *ED* string of size N can be constructed in $\mathcal{O}(N)$ time. Given two *ED* strings, we can construct their path-automata in linear time and apply Corollary 68. By finding any path from the starting to the accepting state in linear time (if it exists), we obtain the result. The construction is detailed on an example in Figure 3.3 □

Notice that the path-automata representing *ED* strings, as well as their intersection, are always acyclic, but may contain ε-transitions. In the following we are only interested in the graph underlying the path-automaton, that is the directed acyclic graph (DAG), where every *node* represents an explicit state and every labeled directed *edge* represents an extended transition of the path-automaton (inspect also Figure 3.2).

3.4.2 An $\widetilde{\mathcal{O}}(N_1^{\omega-1}n_2+N_2^{\omega-1}n_1)$-time algorithm for EDSI

In this section, we start by showing a construction of the *intersection graph* computed by means of Lemma 66 in the case when the input is a pair of path-automata that allows an easier and more efficient implementation. The construction is then adapted to obtain an $\widetilde{\mathcal{O}}(N_1^{\omega-1}n_2+N_2^{\omega-1}n_1)$-time algorithm for solving the EDSI problem.

For $x \in \{1,2\}$ by A_x we denote the compacted NFA (henceforth, graph A_x) representing the *ED* string $\widetilde{T}_x$. By $I_x[i]$ we denote the set of implicit states (henceforth, implicit nodes) appearing on the extended transitions (henceforth, edges) between explicit states (henceforth, explicit nodes) i and $i+1$. For convenience, the implicit nodes in the sets $I_x[1],\ldots,I_x[n_x]$ can be numbered consecutively starting from n_x+2.

Let $U_{i,j} = \{(i,k) : k \in \{j\} \cup I_2[j]\}$ and $U'_{i,j} = \{(k,j) : k \in \{i\} \cup I_1[i]\}$, for all $i \in [1\,..\,n_1+1]$ and $j \in [1\,..\,n_2+1]$. As in the construction of Lemma 66, the union of all $U_{i,j}$ and $U'_{i,j}$ is the set of explicit nodes of the intersection graph that we construct; this can be represented graphically by a grid, where the horizontal and vertical lines correspond to $U_{i,j}$ and $U'_{i,j}$, respectively (inspect Figure 3.4a). In particular, we would

like to compute the edges between these explicit nodes (inspect Figure 3.4b) in $\mathcal{O}(N_1B_2 + N_2B_1)$ time.

Consider an explicit node of the intersection graph; this node is represented by a pair of nodes: one from A_1 and one from A_2. We need to consider two cases: explicit *vs* explicit node; or explicit *vs* implicit node. Without loss of generality, we consider the first node to be explicit. Let us denote this pair by $(i,k) \in U_{i,j}$, where i is an explicit node of A_1 and k is a node of A_2. Let us further denote by ℓ_1 the label of one of the edges going from node i to node $i+1$. For k, we have two cases. If k is explicit (i.e., $k = j$) then we denote by ℓ_2 the label of one of the edges going from k to $k+1$. Otherwise (k is implicit), we denote by ℓ_2 the path label (concatenation of labels) from node k to node $j+1$.

As noted in the proof of Lemma 66, an edge is constructed only if $\mathrm{LCP}(\ell_1,\ell_2) = \min(|\ell_1|,|\ell_2|)$. If $\mathrm{LCP}(\ell_1,\ell_2) = |\ell_2| < |\ell_1|$ (a prefix of a string in $\widetilde{T}_1[i]$ is equal to the suffix of a string in $\widetilde{T}_2[j]$ starting at the position corresponding to node $k \in \{j\} \cup I_2[j]$), the edge ends in a node from $U'_{i,j+1}$ (Figure 3.4c). If $\mathrm{LCP}(\ell_1,\ell_2) = |\ell_1| < |\ell_2|$ (a whole string from $\widetilde{T}_1[i]$ occurs in a string from $\widetilde{T}_2[j]$ starting at the position corresponding to node $k \in \{j\} \cup I_2[j]$), the edge ends in a node from $U_{i+1,j}$ (Figure 3.4d). Otherwise ($\mathrm{LCP}(\ell_1,\ell_2) = |\ell_1| = |\ell_2|$; the two strings are equal) the edge ends in $(i+1, j+1)$. Symmetrically (i.e., the second node is explicit), the edge going out of a node from $U'_{i,j}$ ends at a node from the same set $U'_{i,j+1} \cup U_{i+1,j} \cup \{(i+1, j+1)\}$ (inspect Figure 3.4b).

We next show how to construct the intersection graph by computing all such edges going out of $U_{i,j}$ or $U'_{i,j}$ in a *single batch* using suffix trees (inspect Figure 3.5 for an example). This construction allows an easier and more efficient implementation in comparison to the LCP data structure used in the general NFA intersection construction. Let us recall that $||T||$ denotes the size of the *ED* string $\widetilde{T}$.

Lemma 70. *For any $i \in [1\,..\,n_1+1]$ and $j \in [1\,..\,n_2+1]$, we can construct all $K_{i,j}$ edges going out of nodes in $U_{i,j}$ in $\mathcal{O}(N_{1,i} + N_{2,j} + K_{i,j})$ time, where $N_{1,i} = ||\widetilde{T}_1[i]||$ and $N_{2,j} = ||\widetilde{T}_2[j]||$, using the generalized suffix tree of the strings in $\widetilde{T}_2[j]$.*

Proof. We first construct the generalized suffix tree of the strings in $\widetilde{T}_2[j]$ in $\mathcal{O}(N_{2,j})$ time [91]. We also mark each node corresponding to a suffix of a string in $\widetilde{T}_2[j]$ with a $\widetilde{T}_2$-label. Each such node is also decorated with one or multiple starting positions, respectively, from one or multiple elements of $\widetilde{T}_2[j]$ sharing the same suffix. For each branching node of the suffix tree, we construct a hash table, to ensure that any outgoing edge can be retrieved in constant time based on the first letter (the key) of its label. This can be done in $\mathcal{O}(N_{2,j})$ time with perfect hashing [93]. We next spell each string from $\widetilde{T}_1[i]$ from the root of the suffix tree making implicit nodes explicit or adding new ones if necessary to create the compacted trie of all those strings; and, finally, we mark the reached nodes of the suffix tree with a $\widetilde{T}_1$-label. Spelling all strings from $\widetilde{T}_1[i]$ takes $\mathcal{O}(N_{1,i})$ time.

Every pair of different labels marking two nodes in an ancestor-descendant relationship corresponds to exactly one outgoing edge of the nodes in $U_{i,j}$: (i) if a node marked with a $\widetilde{T}_2$-label is an ancestor of a node marked with a $\widetilde{T}_1$-label, then the suffix of a string from $\widetilde{T}_2[j]$ matches a prefix of a string from $\widetilde{T}_1[i]$ forming an edge

ending in $U'_{i,j+1}$; (ii) if a node marked with a $\widetilde{T}_1$-label is an ancestor of a node marked with a $\widetilde{T}_2$-label, then a string from $\widetilde{T}_1[i]$ occurs in a string from $\widetilde{T}_2[j]$ extending its prefix and forming an edge ending in $U_{i+1,j}$; (iii) if a node is marked with a $\widetilde{T}_1$-label and with a $\widetilde{T}_2$-label, then the suffix of a string from $\widetilde{T}_2[j]$ matches a string from $\widetilde{T}_1[i]$ forming an edge ending in $(i+1, j+1)$. After constructing the generalized suffix tree of $\widetilde{T}_2[j]$ and spelling the strings from $\widetilde{T}_1[i]$, it suffices to make a DFS traversal on the annotated tree to output all $K_{i,j}$ such pairs of nodes. □

3

Theorem 71. *We can construct the intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$ in $\mathcal{O}(N_1B_2 + N_2B_1)$ time using the suffix tree data structure and tree search traversals.*[1]

Proof. We apply Lemma 70 for $U_{i,j}$ and $U'_{i,j}$, for all $i \in [1\,..\,,n_1+1]$ and $j \in [1\,..\,,n_2+1]$. We have that the total number of nodes is $\sum_{i,j} \mathcal{O}(N_{1,i} + N_{2,j}) = \mathcal{O}(N_1n_2 + N_2n_1)$, and then the sum of all output edges is bounded by $\mathcal{O}(N_1B_2 + N_2B_1)$ by Corollary 68. □

Note that if we are interested only in checking whether the intersection is nonempty, and not in the computation of its graph representation, it suffices to check which of the nodes are *reachable* from the starting node, which may be more efficient as there are $\mathcal{O}(N_1n_2 + N_2n_1)$ explicit nodes in this graph.

Let X be the set of nodes of $U_{i,j}$ that are reachable from the starting node. From this set of nodes we need to compute two types of edges (inspect Figure 3.4b). The first type of edges, namely, the ones from X to $U'_{i,j+1} \cup \{(i+1, j+1)\}$ (green edges in Figure 3.4b) are computed by means of Lemma 72, which is similar to Lemma 70. For the second type of edges, namely, the ones from X to $U_{i+1,j} \cup \{(i+1, j+1)\}$ (blue edges in Figure 3.4b), we use a reduction to APE (Problem 1 [43]) (Lemma 73).

Lemma 72. *For any given $X \subseteq U_{i,j}$, we can compute the subset of $U'_{i,j+1} \cup \{(i+1, j+1)\}$ containing all and only the nodes that are reachable from the nodes of X in $\mathcal{O}(N_{1,i} + N_{2,j})$ time.*

Proof. In Lemma 70, the edges from nodes of $U_{i,j}$ to nodes of $U'_{i,j+1}$ come from a pair of nodes in the generalized suffix tree of $\widetilde{T}_2[j]$: one marked with a $\widetilde{T}_1$-label and its ancestor marked with a $\widetilde{T}_2$-label. Notice that the $\widetilde{T}_2$-labels are in a correspondence with the elements of $U_{i,j}$ (the labels on a proper suffix of a string in $\widetilde{T}_2$ are in a one-to-one correspondence with $U_{i,j} \setminus \{(i, j)\}$, and (i, j) corresponds to whole strings in $\widetilde{T}_2[j]$), and hence we can trivially remove the $\widetilde{T}_2$-labels that do not correspond to the elements of X. Furthermore, we are not interested in the set of starting positions decorating a node with a $\widetilde{T}_2$-label; we are interested only in whether a node is $\widetilde{T}_2$-labeled or not (i.e., we do not care from which node of X the edge originates). Since the nodes marked with a $\widetilde{T}_1$-label have in total $N_{1,i}$ ancestors (including duplicates), we can compute the result of this case in $\mathcal{O}(N_{1,i} + N_{2,j})$ time in total. Finally, the node $(i+1, j+1)$ is reachable when a single node is marked with both a $\widetilde{T}_1$-label and a $\widetilde{T}_2$-label. This can be checked within the same time complexity. □

[1] Our implementation of this algorithm can be found at https://github.com/urbanslug/junctions.

The remaining edges (blue edges in Figure 3.4b) are dealt with via a reduction to APE.

Lemma 73. *For any given $X \subseteq U_{i,j}$, we can compute the subset of $U_{i+1,j} \cup \{(i+1, j+1)\}$ containing all and only the nodes that are reachable from the nodes of X in $\widetilde{\mathcal{O}}(N_{1,i} + N_{2,j}^{\omega-1})$ time.*

Proof. The problems of computing the subset of $U_{i+1,j}$ reachable from X and the APE problem can be reduced to one another in linear time.

For the forward reduction, let us set $\mathcal{S} = \widetilde{T}_1[i]$ and $P = \prod_{S \in \widetilde{T}_2[j]} \$S$, where \$ is a letter outside of the alphabet of $\widetilde{T}_1$. This means that we order the strings in $\widetilde{T}_2[j]$, in an arbitrary but fixed way. For a single string $\$S$ (where $S \in \widetilde{T}_2[j]$), the positions from $S[1..|S|-1]$ correspond to the implicit nodes (along the path spelling S) of $I_2[j]$, while the position with \$ corresponds to the explicit node j of A_2 and the one with $S[|S|]$ to the explicit node $j+1$ of A_2. Through this correspondence, we can construct two bit vectors W and V, each of them of size $|P|$, and whose positions are in correspondence with $\{j\} \cup I_2[j] \cup \{j+1\}$ (note that this correspondence is not a bijection, as the explicit nodes j and $j+1$ have several preimages when $|\widetilde{T}_2[j]| \geq 2$). As $U_{i,j} \cup \{(i, j+1)\}$ and $U_{i+1,j} \cup \{(i+1, j+1)\}$ are copies of $\{j\} \cup I_2[j] \cup \{j+1\}$, we use the same correspondence to match positions between W and $U_{i,j} \cup \{(i, j+1)\}$ and between V and $U_{i+1,j} \cup \{(i+1, j+1)\}$. Finally, we set $W[k+1] = 1$ if and only if the corresponding node of $U_{i,j}$ belongs to X (for k corresponding to $(i, j+1)$, we set $W[k] = 0$ as such a node cannot belong to X). After solving APE with W as input vector, we have $V[k] = 1$ for some[2] k corresponding to a node of $U_{i+1,j} \cup \{(i+1, j+1)\}$ if and only if this node is reachable from X.

In more detail, observe that since \$ does not belong to the alphabet of $\widetilde{T}_1$, a string S from $\widetilde{T}_1[i]$ has to match a fragment of a string from $\widetilde{T}_2[j]$ to set $V[k]$ to 1. This happens only if additionally $W[k-|S|] = 1$; both things happen at the same time exactly when: (i) there exists a node $(i, \ell) \in X$; (ii) there exists an edge from (i, ℓ) to $(i+1, \ell')$; and (iii) the positions $k-|S|$ and k in P correspond to ℓ, ℓ', respectively.

In the above reduction we have $|P| = \sum_{S \in \widetilde{T}_2[j]} |S| + 1 = \mathcal{O}(N_{2,j})$, and $||\mathcal{S}|| = N_{1,j}$, hence the lemma statement follows by Lemma 9.

For the reverse reduction, given an instance P, W, $\mathcal{S}$ of APE, we encode it by setting $\widetilde{T}_1[i] = \mathcal{S}, \widetilde{T}_2[j] = \{P\}$ ($N_{1,i} = ||\mathcal{S}||$, $N_{2,j} = |P|$) and X containing the nodes corresponding to positions k where $W[k] = 1$ (the last element of such X is potentially $(i+1, j)$, but we do not care about this corner case of extending the prefix which is already the full string P).

This reduction shows that a more efficient solution to the problem of finding the endpoints of edges originating in X would result in a more efficient solution to APE. □

[2]Here, note that if the node is $(i+1, j)$ or $(i+1, j+1)$, then a corresponding k is not unique, but *at least* one of them must satisfy $V[k] = 1$.

Theorem 74. We can solve EDSI in $\widetilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ time, where ω is the matrix multiplication exponent. If the answer is YES, we can output a witness within the same time complexity.

Proof. It suffices to set the starting node $(1,1)$ as reachable, apply Lemmas 72 and 73, and their symmetric versions for $U'_{i,j}$, for each value of $(i,j) \in [1\,..\,n_1+1] \times [1\,..\,n_2+1]$ in lexicographical order, with X equal to the set of reachable nodes of $U_{i,j}$ (respectively of $U'_{i,j}$); and, finally, check whether node (n_1+1, n_2+1) is set as reachable. We bound the total time complexity of the algorithm by:

$$\sum_{i,j} \widetilde{\mathcal{O}}(N_{1,i}^{\omega-1} + N_{2,j}^{\omega-1}) = \widetilde{\mathcal{O}}(n_2 \sum_i N_{1,i}^{\omega-1} + n_1 \sum_j N_{2,j}^{\omega-1}) \leq \widetilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1).$$

If $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ is nonempty, that is, if the node (n_1+1, n_2+1) is set as reachable from node $(1,1)$, then we can additionally output a witness of the intersection – a single string from $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ – within the same time complexity. To do that we mimic the algorithm on the graph with reversed edges. This time, however, we do not mark all of the reachable nodes; we rather choose a single one that was also reachable from $(1,1)$ in the forward direction. This way, the marked nodes form a single path from $(1,1)$ to (n_1+1, n_2+1). The witness is obtained by reading the labels on the edges of this path. □

Observe that if $n_2 = 1$, that is simply a set of standard strings no node in $U'_{i,1}$ other than $(i,1) \in U_{i,1}$ is reachable – nodes $(i,1)$ can be reached from $(1,1)$ through ε-transitions from $\widetilde{T}_1$, but reaching other nodes would require reading a letter, that is also a change in the state of $\widetilde{T}_2$, and the only explicit state other than 1 in $\widetilde{T}_2$ is 2 (and $(i,2) \in U_{i,2}$). Due to this Lemma 73 never needs to be used to compute transitions between $U'_{i,1}$ and $U'_{i+1,2}$, this allows for a more efficient solution in this case:

Corollary 75. *If $n_2 = 1$ then the running time of Theorem 69 is $\mathcal{O}(N_1 + N_2 B_1)$ and the running time of Theorem 74 is $\widetilde{\mathcal{O}}(N_1 + N_2^{\omega-1}n_1)$.*

This observation is the most interesting in case of the generalized versions of EDSI showed in Section 3.7, where we can compare the running time of our algorithms with the running time of the existing solutions solving special cases of those EDSI generalizations.

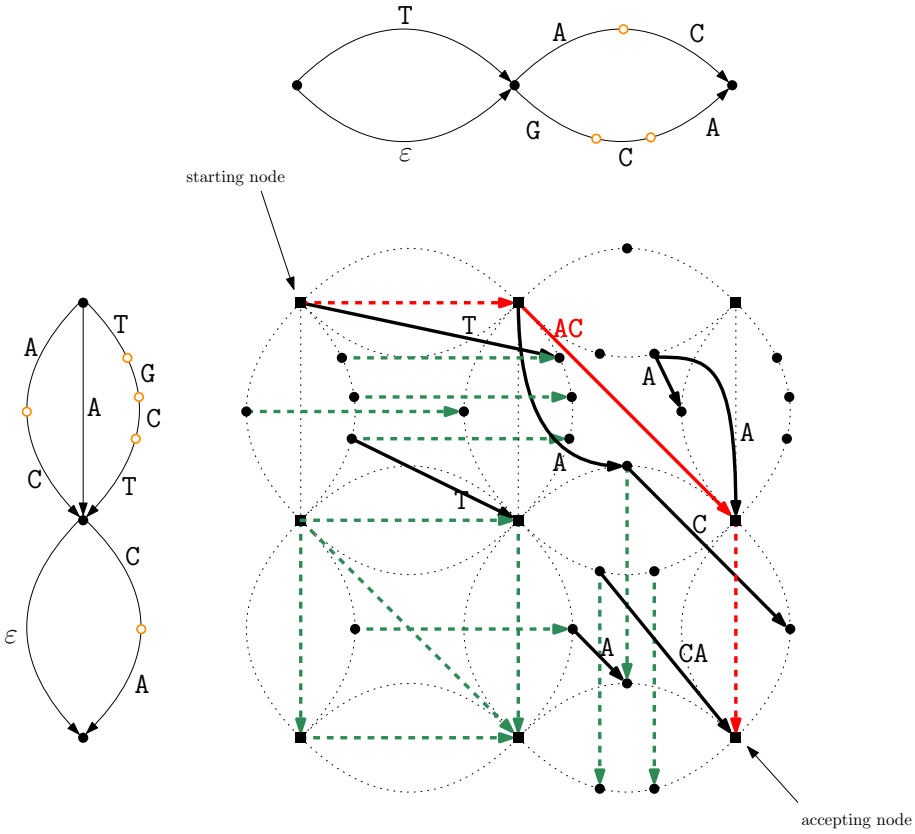
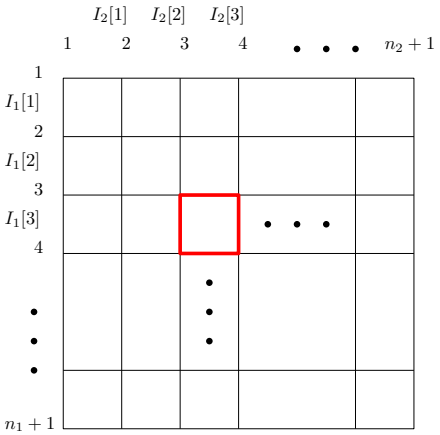
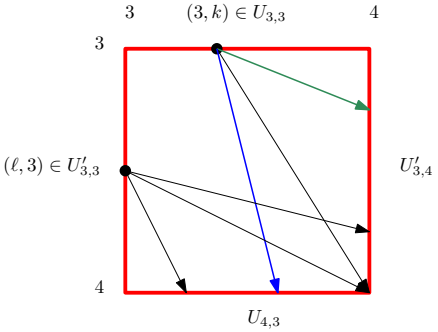


Figure 3.3: Intersection graph G for $\widetilde{T}_1$ and $\widetilde{T}_2$ as in Figure 3.2 where the string AC in $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ that determines a positive answer to the EDSI can be spelled in the path from the starting state to the accepting state. The path automatons A_1 and A_2 are shown on the left and on the top, respectively, and nodes of the intersection graph are arranged along dotted lines that correspond to copies of the layout of A_1 and A_2, to simplify the understanding of G. The dashed edges of the intersection automata correspond to ε-transitions (namely, transitions such that no letter is read when traversed), while the solid edges correspond to the other extended transitions. A string in $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ corresponds to a path from the starting node of G to the accepting node. Here the intersection is nonempty and contains a single string AC, which can be read on the red path.

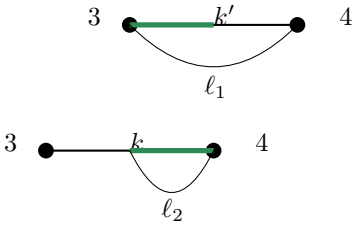


(a) The grid of $U_{i,j}$ and $U'_{i,j}$.



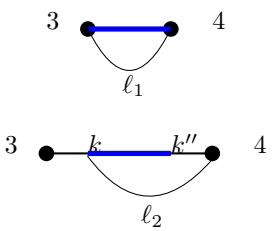
(b) The edges.

$U_{3,3} \to U'_{3,4}$
$(3,k) \to (k',4)$
$\mathsf{LCP}(\ell_1,\ell_2) = |\ell_2|$



(c) The green edge denotes a prefix-suffix match.

$U_{3,3} \to U_{4,3}$
$(3,k) \to (4,k'')$
$\mathsf{LCP}(\ell_1,\ell_2) = |\ell_1|$



(d) The blue edge denotes a full match.

Figure 3.4: An overview of the edges computed by the algorithm.

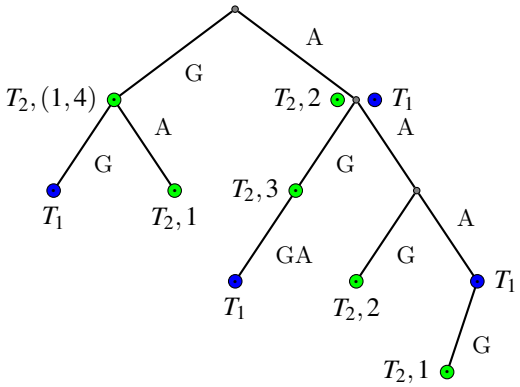


Figure 3.5: The annotated compacted trie constructed for $\widetilde{T}_1[i] = \left\{ \begin{matrix} \text{AGGA} \\ \text{AAA} \\ \text{GG} \\ \text{A} \end{matrix} \right\}$ and $\widetilde{T}_2[j] = \left\{ \begin{matrix} \text{GA} \\ \text{AAAG} \\ \text{G} \end{matrix} \right\}$ in Lemma 70. The node corresponding to G has two $\widetilde{T}_2$ labels and is an ancestor of the node corresponding to GG with a $\widetilde{T}_1$ label; hence two corresponding edges to $U'_{i,j+1}$ are constructed. The node corresponding to AAA has a $\widetilde{T}_1$ label and is an ancestor of the node corresponding to AAAG with a $\widetilde{T}_2$ label; hence a corresponding edge to $U_{i+1,j}$ is constructed. The node corresponding to a has both a $\widetilde{T}_1$ and a $\widetilde{T}_2$ label; hence a corresponding edge to $(i+1, j+1)$ is constructed.

3.5 ACRONYM GENERATION

In this section, we study a problem on standard strings. Given a sequence $P = P_1, \ldots, P_n$ of n strings we define an *acronym* of P as a string $A = A_1 \cdots A_n$, where A_i is a (possibly empty) prefix of P_i, $i \in [1..n]$. We next formalize the ACRONYM GENERATION problem.

Problem 10. ACRONYM GENERATION (AG)
Input: A set D of k strings of total length K and a sequence $P = P_1, \ldots, P_n$ of n strings of total length N.
Output: YES if some acronym of P is an element of D, NO otherwise.

The AG problem is underlying real-world information systems (e.g., see https://acronymify.com/ or https://acronym-generator.com/) and existing approaches rely on brute-force algorithms or heuristics to address different variants of the problem [128, 139, 146, 147, 156, 190, 199, 209]. These algorithms usually accept a sequence P of $n \leq n_{\max}$ strings, for some small integer $n_{\max}$, which highlights the lack of efficient exact algorithms for generating acronyms. Here we show an exact polynomial-time algorithm to solve AG for any n.

We can encode AG by means of EDSI and modify the developed methods. Let $\widetilde{T}_1[i]$, $i \in [1..n]$, be the set of all prefixes of P_i and further let $\widetilde{T}_2[1] = D$. By using Theorem 69 or Corollary 68 we obtain an $\mathcal{O}(\sum_i |P_i|^2 k + KN) = \mathcal{O}(N^2 k + KN)$-time algorithm, while using Theorem 74 we obtain an $\widetilde{\mathcal{O}}(N^{2\omega-2} + K^{\omega-1} n)$-time algorithm, for solving the AG problem ($\mathcal{O}(N^2 + KN)$ and $\widetilde{\mathcal{O}}(N^2 + K^{\omega-1} n)$ respectively by Corollary Corollary 75 since $n_2 = 1$).

Since, however, all elements of set $\widetilde{T}_1[i]$ are prefixes of a single string (P_i), we can obtain a more efficient graph representation of $\widetilde{T}_1$ by joining nodes i and $i+1$ with a single path labeled with P_i, with an additional ε edge between every (implicit) node of the path and node $i+1$. As the size of the graph for $\widetilde{T}_1$ is smaller ($\mathcal{O}(N)$ nodes and edges), by using Lemma 66 we obtain an $\mathcal{O}(Nk + KN) = \mathcal{O}(NK)$-time algorithm for solving the AG problem.

The considered *ED* strings have additional strong properties however. The sets $\widetilde{T}_1[i]$'s are not just sets of prefixes of single strings, but sets of all their prefixes, while the length n_2 of $\widetilde{T}_2$ is equal to 1. By employing these two properties we obtain the following improved result.

Theorem 76. *AG can be solved in $\mathcal{O}(nK + N)$ time.*

Proof. The algorithm of Theorem 74 is based on finding out which elements of sets $U_{i,j}, U'_{i,j}$ are reachable; however, since $n_2 = 1$, the sets $U'_{i,j}$ are trivialized: $\widetilde{T}_2$ has only two explicit nodes - each can only be matched with explicit nodes of $\widetilde{T}_1$ on any path that starts at node $(1,1)$ and ends at node $(n_1 + 1, 2)$ (a node $(k,1)$ for implicit node k has no ingoing edge, while a node $(k,2)$ for implicit node k has no outgoing edge). Thus it is enough to focus on the transitions between $U_{i,j}$ and $U_{i+1,j} \cup (i+1, 2)$.

In Lemma 73, to compute the reachable nodes of $U_{i+1,j}$ knowing the reachable nodes of $U_{i,j}$, fast matrix multiplication is employed (Lemma 9), but in this special

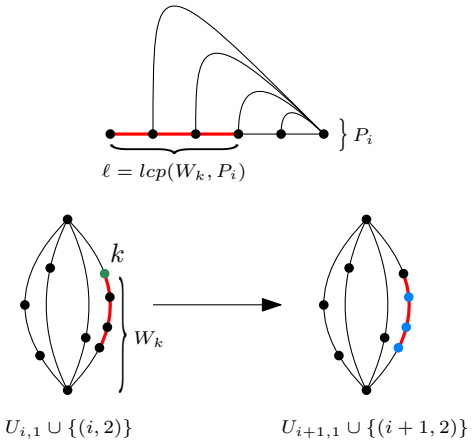


Figure 3.6: For the AG problem, the automaton for the prefixes of P_i can be represented as the one on top of the figure. The sets $U_{i,1} \cup \{(i,2)\}$ and $U_{i+1,1} \cup \{(i+1,2)\}$ can be represented as copies of the path automaton of D, as in the general case. A green node k in $U_{i,1} \cup \{(i,2)\}$ is specified. The red segments are path reading the LCP string between W_k and P_i. The nodes that are reachable from k in $U_{i+1,1} \cup \{(i+1,2)\}$ are highlighted in blue.

case a simpler method will be more effective. Let W_k be the string read between nodes k and 2 in the path-graph of $\widetilde{T}_2$. The crucial observation is: the edges going out of node $(i,k) \in U_{i,1} \cup \{(i,2)\}$ for $k \neq 1$ end in nodes $(i+1,k')$ for $k' \in [k..k+l]$, where $l = \mathrm{LCP}(P_i, W_k)$ as the strings from $\widetilde{T}_1[i]$ matching the prefix of W_k are exactly all the prefixes of P_i of length at most l (see Figure 3.6).

Hence to compute the reachable subset of $U_{i+1,1} \cup \{(i+1,2)\}$, we can handle the edges going out of $(i,1)$ separately in $\mathcal{O}(K+|P_i|)$ time by letter comparisons, then compute the $\mathrm{LCP}(P_i, W_k)$ for all the reachable nodes (i,k) either using the LCP data structure, or with the use of the generalized suffix tree of $\widetilde{T}_2[1] = D$ in $\mathcal{O}(K+|P_i|)$ total time, and finally, using a sweep line approach, compute the union of the obtained intervals in $\mathcal{O}(K)$ time. We answer YES if and only if node $(n+1,2)$ is reachable.

Over all i values this gives an algorithm running in $\sum_i \mathcal{O}(K+|P_i|) = \mathcal{O}(nK+N)$ total time. Furthermore one is allowed to choose, for each $i \in [1..n]$, the minimal length x_i of the prefix of P_i (including length $x_i = 0$ if one wants to allow empty prefixes) used in the acronym (some strings should not be completely excluded from the acronym). The only modification to the algorithm in such a generalized case is replacing intervals $[k..k+l]$ by $[k+x_i, k+l]$, which does not influence the claimed complexity. □

Corollary 77. *If the answer to the instance of the AG problem is YES, we can output all strings in D which are acronyms of P within $\mathcal{O}(nK+N)$ time.*

Proof. We want to find out which paths of $\widetilde{T}_2$ represent acronyms of P. In the algorithm employed by Theorem 76 the reachable nodes of $U_{i,1} \cup \{(i+1,2)\}$ are found. If there exists an edge from a reachable node (i,k) for $k \notin \{1,2\}$ to $(i+1,2)$ for some

$i \in [1..,n_1]$, then the path of the path-graph of $\widetilde{T}_2$ containing node k represents an acronym of P. If node $(i+1,2)$ is reached directly from reachable node $(i,1)$, then the whole prefix of P_i used to do that is in D, and hence is a standalone acronym of P. If for a path neither of the two cases qualifies, then it cannot be used to reach node $(n+1,2)$, and hence is not an acronym of P.

If the generalization with minimal lengths of prefixes is applied, then the values of i used here are restricted to $[i',n]$, where i' is the largest value of i with a restriction $x_i > 0$: node $(i'-1,2)$ does not have an edge to node $(i',2)$, and hence does not belong to any path from $(1,1)$ to $(n+1,2)$. □

3

Let us remark that although the main focus of real-world acronym generation systems is on the natural language parsing and interpretation of acronyms, our new algorithmic solution may inspire practical improvements in such systems or further algorithmic work.

3.6 ED STRING COMPARISON TASKS

In this section, we show some more applications of our techniques from Section 3.4 to solve different *ED* string comparison tasks.

We consider two *ED* strings, $\widetilde{T}_1$ of length n_1, cardinality B_1 and size N_1, and $\widetilde{T}_2$ of length n_2, cardinality B_2 and size N_2. We call *intersection graph* the underlying graph of an automaton representing $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$. By Corollary 68 such an automaton (and therefore the corresponding intersection graph) can be constructed in $\mathcal{O}(N_1B_2 + N_2B_1)$ time. In Section 3.4, such a graph was used to check whether $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ is nonempty.

In the following, we present other applications of intersection graphs (computed by means of Corollary 68 or Lemma 70) to tackle several natural *ED* string comparison tasks with no additional time complexity. Let us start with the most basic such task.

Problem 11. EDSI SHORTEST/LONGEST WITNESS
Input: Two *ED* strings, $\widetilde{T}_1$ of length n_1, cardinality B_1 and size N_1, and $\widetilde{T}_2$ of length n_2, cardinality B_2 and size N_2.
Output: A shortest (resp. longest) element of $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ if it is a nonempty set, FAIL otherwise.

Lemma 78. *The EDSI SHORTEST/LONGEST WITNESS problem can be solved in $\mathcal{O}(N_1B_2 + N_2B_1)$ time by using an intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$.*

Proof. We compute the intersection graph G as the underlying graph of the automaton computed in Corollary 68. Given an edge (k,k') in G, we assign a weight $w(k,k')$ equal to the length of its string label (the string ε has length 0). Note that, by construction, G is a directed acyclic graph, hence we can sort its nodes topologically. In this topological order we define $L_{\min}((1,1)) = 0$ (resp. $L_{\max}((1,1)) = 0$), where $(1,1)$ is the starting node, and we compute inductively, via the topological order, the values $L_{\min}(k) = \min_{\{k' \text{ predecessor of } k\}} L(k') + w(k',k)$ (resp. $L_{\max}(k) = \max_{\{k' \text{ predecessor of } k\}} L(k') + w(k',k)$) for each node k in G. Then, we backtrack from

the accepting node to find the path of length $\mathcal{O}(n_1+n_2)$ from $(1,1)$ to (n_1+1,n_2+1) of minimal (resp. maximal) total weight. By reading the labels on the path in $\mathcal{O}(N_1+N_2)$ time, we can output the shortest (resp. longest) element of $\mathcal{L}(\widetilde{T}_1)\cap\mathcal{L}(\widetilde{T}_2)$.[3] □

In the next task, we would like to compute the size of $\mathcal{L}(\widetilde{T}_1)\cap\mathcal{L}(\widetilde{T}_2)$ and, in particular, by considering multiplicities in $\mathcal{L}(\widetilde{T}_1)\cap\mathcal{L}(\widetilde{T}_2)$. Let us start with the following definition.

Definition 79. We equip $\mathcal{L}(\widetilde{T}_1)\cap\mathcal{L}(\widetilde{T}_2)$ with a multiset structure, inherited from the multiset structure of $\mathcal{L}(\widetilde{T}_1)$ (resp. $\mathcal{L}(\widetilde{T}_2)$), in which the multiplicity of a string T is the number of sequences $T_1\in\widetilde{T}_1[1],\ldots,T_{n_1}\in\widetilde{T}_1[n_1]$ such that $T_1\cdots T_{n_1}=T$ (resp. the number of sequences $T_1\in\widetilde{T}_2[1],\ldots,T_{n_2}\in\widetilde{T}_2[n_2]$ such that $T_1\cdots T_{n_2}=T$). If a string T has multiplicity μ_1 in $\mathcal{L}(\widetilde{T}_1)$ and multiplicity μ_2 in $\mathcal{L}(\widetilde{T}_2)$, then it has multiplicity $\mu_1\cdot\mu_2$ in $\mathcal{L}(\widetilde{T}_1)\cap\mathcal{L}(\widetilde{T}_2)$.

Problem 12. EDSI MULTISET SIZE
Input: Two *ED* strings, $\widetilde{T}_1$ of length n_1, cardinality B_1 and size N_1, and $\widetilde{T}_2$ of length n_2, cardinality B_2 and size N_2.
Output: The cardinality of the multiset $\mathcal{L}(\widetilde{T}_1)\cap\mathcal{L}(\widetilde{T}_2)$.

Each element of the multiset $\mathcal{L}(\widetilde{T}_1)\cap\mathcal{L}(\widetilde{T}_2)$ can be represented by a pair of *alignments*: the sequence of n_1+n_2 choices of a single $S_i\in\widetilde{T}_1[i]$ and $S'_j\in\widetilde{T}_2[j]$ for each $i\in[1\,..\,n_1]$ and $j\in[1\,..\,n_2]$, such that the resulting standard string is the same. Such a pair of alignments can in turn be represented by a path in the intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$ as the edges correspond to (parts of) a string in some $\widetilde{T}_1[i]$ and in some $\widetilde{T}_2[j]$.

The representation is almost unique; the only reason this does not need to be the case is when a node $(i+1,j+1)$ is reached from a node (i,j) for $i\in[1\,..\,n_1]$, $j\in[1\,..\,n_2]$ using only ε-edges. In this case, the three subpaths $(i,j)\to(i+1,j+1),(i,j)\to(i,j+1)\to(i+1,j+1)$ and $(i,j)\to(i+1,j)\to(i+1,j+1)$, all correspond to the choice of $\varepsilon\in\widetilde{T}_1[i]$ and $\varepsilon\in\widetilde{T}_2[j]$, even though this choice should be counted as 1.

To fix the notation, we call an ε-edge: *vertical*, when it leads from a node (i,k) to $(i+1,k)$ for some explicit node i of A_1; *horizontal*, when it leads from a node (k,j) to $(k,j+1)$ for some explicit node j of A_2; and *diagonal* if it leads from a node (i,j) to $(i+1,j+1)$, where both i and j are explicit nodes (see Figure 3.4b, where such ε-edges correspond to the vertical, horizontal, and diagonal – from the top-left corner to the bottom-right one – arrows).

The problem may gets more difficult when a few such ε are used in a row in both *ED* strings (a node $(i+x,j+y)$ is reached from (i,j) using only ε-edges), as a single alignment would correspond to all the "down, right or diagonal" paths in the $x\times y$ grid (see Figure 3.4a). The number of such paths can be large, and even if we remove all such diagonal ε-edges (those can be always simulated with a single horizontal

[3] In the case of a shortest witness, we can obtain an equally efficient algorithm by employing Dijkstra's algorithm using bucket queue since the bound $\mathcal{O}(N_1+N_2)$ on the weight of the path is known beforehand.

and a single vertical one), we cannot remove the horizontal or vertical ones, as those can be traversed by other paths independently. We are still left with $\binom{x+y}{x}$ equivalent subpaths from (i, j) to $(i+x, j+y)$. In order to mitigate this problem we will restrict the usage of such subpaths of many ε-edges to a single, regular one.

We call a path *ε-regular*, if it does not use diagonal ε-edges, and if two ε-edges are used sequentially, then they either have the same direction, or the latter one is horizontal.

Lemma 80. *There is a one-to-one correspondence between pairs of alignments that produce the same string and ε-regular paths from the starting to the accepting node in the intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$.*

Proof. Consider a pair of alignments – it can be represented by the produced string together with some positions in-between letters marked with (n_1+1) $\widetilde{T}_1$-labels and (n_2+1) $\widetilde{T}_2$-labels in total specifying the beginning and ending of the productions used in $\widetilde{T}_1[i]$ and $\widetilde{T}_2[j]$, for $i \in [1\,..\,n_1], j \in [1\,..\,n_2]$. A single position can be marked with both types of label and even many labels of the same type (when ε-productions are used). Each position corresponds to a pair of states in the path-automata: those pairs of states with at least one label read from left to right form a path in the intersection graph, as the $\widetilde{T}_i$-label shows that the state is explicit in $\widetilde{T}_i$. If two labels of a different type appear in the same place, this corresponds to a node composed of two explicit states. When a single position contains multiple labels from both types, the exact ordering between those labels represents the actual path in the graph. At the same time from the point of view of $\widetilde{T}_1$ and $\widetilde{T}_2$ separately all those orderings correspond to exactly the same pair of alignments - this is, where ε-regularity plays its role and only one subpath is created (all $\widetilde{T}_1$-labels are placed before the $\widetilde{T}_2$-labels). This way we have defined an injection from the pairs of alignments to the paths of the intersection graph (each pair of alignments corresponds to only one ε-regular path).

On the other hand the labels of the edges in the path represent (the parts of) the transitions used in each automaton, and hence also a pair of alignments, thus the function is also surjective.

We have thus shown that the function relating a pair of alignments to a path is both injective and surjective, and hence a bijection (the one-to-one correspondence from the statement). □

The main result then follows.

Lemma 81. *The EDSI MULTISET SIZE problem can be solved in $\mathcal{O}(N_1B_2 + N_2B_1)$ time by using an intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$.*

Proof. We compute the intersection graph as the underlying graph of the automaton computed in Corollary 68 or Lemma 70. We are counting the number of distinct ε-regular paths from the starting node to the accepting one, that is by Lemma 80, the number of elements of the multiset. As previously, we can sort the nodes topologically. If we wanted to compute all the paths it would be enough to compute

for each node k the value $S(k) = \sum_{\{(k',k)\}\in E(G)} S(k')$, over all edges (k',k) (including parallel edges with different labels) and setting $S((1,1)) = 1$ where $(1,1)$ is the starting node. This time however, we apply a little more complicated formulas, which count the paths ending with the ε-edges separately from the other ones, and among those separately depending on the direction of the ε-edge.

- $S(k) = \sum_{\{(k',k) \text{ positive length edges}\}} S(k') + S^h(k') + S^v(k')$
- $S^h(k) = \sum_{\{(k',k) \text{ horizontal } \varepsilon\text{-edges}\}} S(k') + S^h(k') + S^v(k')$
- $S^v(k) = \sum_{\{(k',k) \text{ vertical } \varepsilon\text{-edges}\}} S(k') + S^v(k')$

By induction, for the accepting node (n_1+1, n_2+1), the number of distinct ε-regular paths from the starting to the accepting node is equal to the sum of the values computed for node (n_1+1, n_2+1). □

Asking whether $\mathcal{L}(\widetilde{T}_1) \cap \mathcal{L}(\widetilde{T}_2)$ is not nonempty, as a way to tell if two *ED* strings share something, can be too restrictive in practical applications. We will thus consider two more elaborate *ED* string comparison tasks that consider local matches rather than a match that necessarily involves the entire *ED* strings from beginning to end. Both notions are heavily employed on standard strings for practical applications – especially in bioinformatics: MATCHING STATISTICS and LONGEST COMMON SUBSTRING.

We start by extending the classic MATCHING STATISTICS problem [112] from the standard string setting to the *ED* string setting.

Further discussion and implementations[4] on this solution can be found in [96].

Problem 13. *ED* MATCHING STATISTICS
Input: Two *ED* strings, $\widetilde{T}_1$ of length n_1, cardinality B_1 and size N_1, and $\widetilde{T}_2$ of length n_2, cardinality B_2 and size N_2.
Output: For each $i \in [1\,..\,,n_1]$, the length $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[i]$ of the longest prefix of a string in $\mathcal{L}(\widetilde{T}_1[i\,..\,n_1])$, which is a substring of a string in $\mathcal{L}(\widetilde{T}_2)$.

Example 82. Let us consider again $\widetilde{T}_1 = \left\{ \begin{array}{c} \mathrm{A} \\ \mathrm{AC} \\ \mathrm{TGCT} \end{array} \right\} \cdot \left\{ \begin{array}{c} \varepsilon \\ \mathrm{CA} \end{array} \right\}$ and $\widetilde{T}_2 = \left\{ \begin{array}{c} \varepsilon \\ \mathrm{T} \end{array} \right\} \cdot \left\{ \begin{array}{c} \mathrm{GCA} \\ \mathrm{AC} \end{array} \right\}$ of our running example. We have that $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[1] = 3$ and $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[2] = 2$. Indeed, the longest prefix of a string in $\mathcal{L}(\widetilde{T}_1)$ that occurs in $\mathcal{L}(\widetilde{T}_2)$ is TGC, having length 3, and the longest prefix of a string in $\mathcal{L}(\widetilde{T}_1[2])$ that occurs in $\mathcal{L}(\widetilde{T}_2)$ is CA, having length 2.

Lemma 83. *The ED MATCHING STATISTICS problem can be solved in $\mathcal{O}(N_1B_2 + N_2B_1)$ time by using an intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$.*

[4] Our implementation of this algorithm can be found at https://github.com/urbanslug/junctions.

Proof. This time, we will use a slightly augmented version of the intersection graph coming from the unpruned intersection automaton. Namely, we construct the automaton as in Corollary 68, but do not remove the unreachable parts or the "partial transitions". That is, when we process an explicit state corresponding to a pair (u,v) of states in the path automata A_1 and A_2, and a pair (s,t) of transitions going out of u and v, we construct the corresponding transition to the pair of states that can be reached through s and t, even if this transition finishes in a pair of implicit states, namely when $0 < \mathrm{LCP}(\ell_1,\ell_2) < \min(|\ell_1|,|\ell_2|)$, where ℓ_1 and ℓ_2 are the respective labels of s and t. Even in that case, the number of transition pair checks remains the same, and therefore the total size of the constructed underlying graph G stays $\mathcal{O}(N_1B_2+N_2B_1)$.

Once again we assign to each edge the weight w storing the length of their string label and process the nodes in the reversed topological order to compute for each node k the value $M(k) = \max_{k'} M(k') + w(k,k')$ where k' iterates over all successors of k. One has $M(k) = 0$ for the nodes that do not have successors (for example the accepting node or nodes corresponding to a pair of implicit states).

By construction, we have $M((i,v)) = \ell$, for an explicit state i of A_1 and a state v of A_2, if and only if ℓ is equal to the maximal LCP between a pair (T_1,T_2), where T_1 a string in $\mathcal{L}(\widetilde{T}_1[i\,..\,n])$ and T_2 is a string read starting at (explicit or implicit) state v in A_2.

For every explicit state i of A_1 we can compute $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[i] = \max_v M((i,v))$ over all (explicit or implicit) states v of A_2 to obtain the output.

□

An example of this construction is shown in Figure 3.7.

We now move to extending the classic LONGEST COMMON SUBSTRING problem [112] from the standard string setting to the *ED* string setting.

Problem 14. ED LONGEST COMMON SUBSTRING
Input: Two *ED* strings, $\widetilde{T}_1$ of length n_1, cardinality B_1 and size N_1, and $\widetilde{T}_2$ of length n_2, cardinality B_2 and size N_2.
Output: A longest string that occurs in a string of $\mathcal{L}(\widetilde{T}_1)$ and a string of $\mathcal{L}(\widetilde{T}_2)$

In the *ED* MATCHING STATISTICS problem only strings starting in explicit nodes of A_1 were considered; here we want to lift this restriction. Computing the value of M (as in the proof of Lemma 83) for all pairs of nodes of A_1 and A_2 would take $\Theta(N_1N_2)$ time; we are only interested in the globally maximal value of M however, and hence we can focus only on the computation of those values for a certain subset of those pairs.

Lemma 84. *The ED LONGEST COMMON SUBSTRING problem can be solved in $\mathcal{O}(N_1B_2 + N_2B_1)$ time by using an intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$.*

Proof. We start from the augmented graph G, as defined in the proof of Lemma 83, and computed in $\mathcal{O}(N_1B_2+N_2B_1)$ time. In addition to the standard edges between the explicit nodes, the graph contains edges from a pair containing at least one explicit state to a pair of implicit states, that are inclusion-wise maximal, that is cannot be

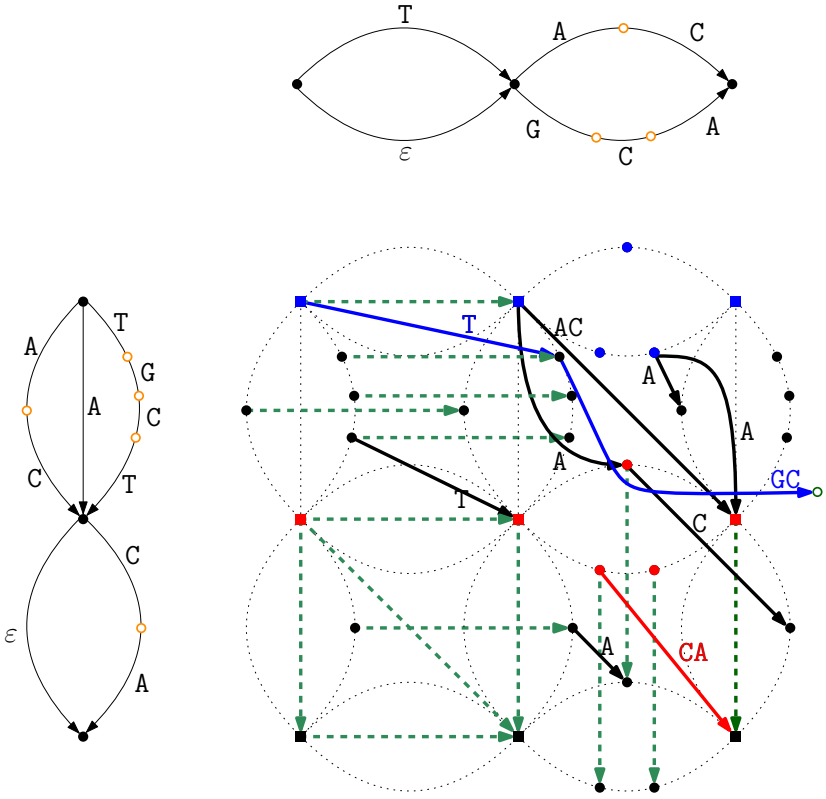


Figure 3.7: Matching Statistics of $\widetilde{T}_1$ and $\widetilde{T}_2$ of our running example on their intersection graph G, where, again - to simplify the understanding - we also draw A_1 and A_2 at the left and on the top, respectively. Note that this time, the pairs of implicit nodes that are reachable in a single extended transition from a pair that was previously computed are added. In the figure, there is only one such extra node, which is represented by a green open circle at the right of the graph. Here we highlight the paths that are relevant for computing the Matching Statistics array $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}$. To compute $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[1]$, we look at the paths starting at nodes (i,j) where i is the explicit state 1 in the path-automaton of $\widetilde{T}_1$, and return the length of the longest label of such a path. These are the paths starting in one of the blue nodes (these are the nodes that correspond to the uppermost explicit node of A_1 paired with any node of A_2, that is, they correspond to the uppermost dotted copy of A_2). The longest one of such paths (also drawn in blue) corresponds to the string TGC having length 3; therefore, $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[1] = 3$. For $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[2]$ we do the same but using as starting nodes those in red that correspond to the internal explicit node of A_1 paired with any node of A_2 (i.e. the nodes of the middle dotted copy of A_2). Here the longest path is drawn in red and it spells the string CA, and therefore we set $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[2] = 2$. Computing $\mathrm{MS}_{\widetilde{T}_2,\widetilde{T}_1}$ can be performed in a dual manner on the same graph, but using as starting nodes those of the left dotted copy of A_1 for $\mathrm{MS}_{\widetilde{T}_2,\widetilde{T}_1}[1]$, and those of the middle dotted copy of A_1 for $\mathrm{MS}_{\widetilde{T}_2,\widetilde{T}_1}[2]$.

extended to obtain a longer edge. For the ED Longest Common Substring problem, we additionally need edges symmetric to those – the (inclusion-wise maximal) edges starting in pairs of implicit states that end in pairs containing at least one explicit state. By a symmetric argument (argument for the reversed automata), there are $\mathcal{O}(N_1B_2 + N_2B_1)$ such edges and all can be computed together with the rest of the graph within the same time complexity. We compute the values $M(k)$ for each node k as previously, and find their global maximum. Notice that we do not need to compute the value M for the pairs of implicit nodes lying in the middle of an edge as its predecessor on the edge always has a bigger value of M.

We are left with the nodes that are not (weakly) connected to any explicit node – the isolated edges containing such nodes were not computed at all. Those edges correspond to strings that, in both *ED* strings, are fully contained in a single set, hence it is enough to compute the longest common substring between sets $\bigcup_i \widetilde{T}_1[i]$ and $\bigcup_j \widetilde{T}_2[j]$. This can be done in $\mathcal{O}(N_1 + N_2)$ time after constructing the generalized suffix tree of all these strings, and finding the deepest nodes with descendants of both types: a descendant originating from a suffix of a string in $\bigcup_i \widetilde{T}_1[i]$ and a descendant originating from a suffix of a string in $\bigcup_j \widetilde{T}_2[j]$.

The method of obtaining the witness is the same as in the previous problems (after computing both endpoints of the optimal path using the algorithm described above). □

Finally, we show how to extend the classic Longest Common Subsequence (LCS) problem [112] from the standard string setting to the *ED* string setting. We remark that, unlike the previous problems, in the standard string setting, LCS is not solvable in linear time: there exists a conditional lower bound saying that no algorithm with running time $\mathcal{O}(n^{2-\varepsilon})$, for any $\varepsilon > 0$, can solve LCS unless SETH fails [48].

Problem 15. ED Longest Common Subsequence
Input: Two *ED* strings, $\widetilde{T}_1$ of size N_1, and $\widetilde{T}_2$ of size N_2.
Output: A longest string that occurs in a string of $\mathcal{L}(\widetilde{T}_1)$ and a string of $\mathcal{L}(\widetilde{T}_2)$ as a subsequence.

Lemma 85. *The ED Longest Common Subsequence problem can be solved in $\mathcal{O}(N_1 \cdot N_2)$ time by using an uncompacted intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$.*

Proof. Consider the uncompacted path automata for $\widetilde{T}_1$ and $\widetilde{T}_2$, having respective sizes $\mathcal{O}(N_1)$, $\mathcal{O}(N_2)$. For every single transition in those graphs (that is a single letter transition since they are uncompacted), we add a parallel ε-transition (that is we allow to skip the letter). The sizes of the automata remain $\mathcal{O}(N_1)$ and $\mathcal{O}(N_2)$, and the intersection has size $\mathcal{O}(N_1 \cdot N_2)$. We can then find the longest witness (Lemma 78) of this intersection in time linear in the size of the automaton, that is in $\mathcal{O}(N_1 \cdot N_2)$. Notice that when $\widetilde{T}_1$ and $\widetilde{T}_2$ are standard strings, then $n_1 = B_1 = n_2 = B_2 = 1$, hence $\mathcal{O}(N_1 \cdot N_2)$ running time matches the lower bound for standard strings, and no N_i can be replaced by B_i or n_i - in particular an $\mathcal{O}(N_1B_2 + N_2B_1)$-time algorithm would contradict the lower bound [48].

□

Remark 86. In all the solutions presented in this section (as well as in the previous ones) the algorithm can be slightly modified to achieve space complexity $\mathcal{O}(N_1+N_2)$ without any increase in the running time.

Proof. Notice that all the problems are solved through computing a certain value (reachability, $L(k), S(k), M((i,v))$) for all the explicit nodes of the intersection graph in a topological order. Each time this order is compatible with the order of sets of $\widetilde{T}_1$ and $\widetilde{T}_2$, and the recurrent formulas refer only to nodes from a single set $U_{i,j}$ or $U'_{i,j}$ that is the basic computation is enclosed in a single cell of the grid (inspect Figure 3.4a). For a single cell of the grid the space used for computation is $\Theta(N_{1,i}+N_{2,j})$, while globally the information passed between the cells is bounded by $\mathcal{O}(N_1+N_2)$ by going through the cells (i,j) in a lexicographical (or reversed lexicographical) order (it is enough to store information for nodes of $U_{i,j}, U'_{i,j}$ only for the next two values of i). Finally the size of the input as well as of all the generalized suffix trees is also bounded by $\mathcal{O}(N_1+N_2)$. □

3.7 Approximate EDSI

Another popular extension of string equality is *sequence alignment*, where one wants to find the distance between two strings: the minimal number of operations that modify one string into the other one, or decide whether this number is at most k, for a given integer $k > 0$.

We denote by $d_H(T_1,T_2)$ (resp. $d_E(T_1,T_2)$) the Hamming distance (resp. edit distance) of two standard strings T_1, T_2. The problem of finding the Hamming distance of two standard strings is easily solvable in $\mathcal{O}(N_1+N_2)$ time, while for edit distance the time increases to $\mathcal{O}(N_1 \cdot N_2)$ time in general case [210] or $\mathcal{O}(N_1+N_2+k^2)$ [149] for a given upper bound k.

Approximate EDSI gives another measure of similarity when the normal intersection turns out to be empty.

Theorem 87. Given a pair $\widetilde{T}_1, \widetilde{T}_2$ of *ED* strings, we can find the pair $T_1 \in \mathcal{L}(\widetilde{T}_1), T_2 \in \mathcal{L}(\widetilde{T}_2)$ minimizing the distance $d_H(T_1,T_2)$ or $d_E(T_1,T_2)$ in $\mathcal{O}(N_1N_2)$ time.

Proof. We adapt the classic Wagner–Fischer algorithm [210] to our case – we construct the standard (uncompacted) NFA intersection of the two path-automata for $\widetilde{T}_1$ and $\widetilde{T}_2$, set the weight of all its edges to 0, and then add extra edges with weight 1 that represent edit operations.

In case of Hamming distance, when a pair of transitions $u \xrightarrow{e_1} u'$, $v \xrightarrow{e_2} v'$ does not match ($e_1 \neq e_2$) we still add the weight-1 edge between (u,v) and (u',v') (e_1 is substituted with e_2).

In case of edit distance we additionally construct weight-1 edges from (u,v) to (u',v) for every transition $u \xrightarrow{e_1} u'$ (corresponding to deletion of the letter e_1), and to (u,v') for every transition $v \xrightarrow{e_2} v'$ (corresponding to insertion of letter e_2).

Now all we need to do is to find the minimum cost path from the starting node to the accepting one (in case of Hamming distance only if such a path exists), which can be done in linear time using Dial's implementation of Dijkstra's algorithm [78].

By simply spelling the labels of the found path we obtain strings T_1 and T_2 (the difference between them is encoded through the labels of the weight-1 edges). □

Without having a threshold k, even when $\widetilde{T}_1, \widetilde{T}_2$ are standard strings ($n_1 = n_2 = B_1 = B_2 = 1$), we cannot obtain an algorithm running in $\mathcal{O}((N_1N_2)^{1-\varepsilon})$ time, for a constant $\varepsilon > 0$, (unless SETH fails [31]) or a faster algorithm through parameterization with n_1, n_2, B_1, B_2.

On the other hand, if we are given a threshold k, we can adapt another classic algorithm, given by Landau and Vishkin [152], to obtain the following theorem.

Theorem 88. Given a pair $\widetilde{T}_1, \widetilde{T}_2$ of *ED* strings and an integer $k > 0$, we can check whether a pair $T_1 \in \mathcal{L}(\widetilde{T}_1), T_2 \in \mathcal{L}(\widetilde{T}_2)$ with $d_H(T_1, T_2) \leq k$ (resp. $d_E(T_1, T_2) \leq k$) exists in $\mathcal{O}(k(N_1B_2 + N_2B_1))$ time (resp. in $\mathcal{O}(k^2(N_1B_2 + N_2B_1))$ time) and, if that is the case, return the pair with the smallest distance.

Proof. This time we make use of the compacted intersection automaton from Lemma 66. In addition to standard weight-0 edges, which are constructed when for $\mathrm{LCP}(l_1, l_2) = \min(|l_1|, |l_2|)$ for l_1, l_2 the labels of extended transitions in the two path-automata, we also add new edges when one of the strings is at distance at most k from a prefix of the other one. More formally, for a pair of extended transitions $u \xrightarrow{l_1} u'$, $v \xrightarrow{l_2} v'$, if l_1 is at distance k' from the length x prefix of l_2, we produce a weight-k' edge from (u, v) to (u', v_x), where v_x is the implicit state between v and v' representing the length x prefix of l_2 (symmetrically for l_2 at distance at most k from a prefix of l_1). All the smallest values of $k' \leq k$ can be found with the use of standard LCP queries data structure and kangaroo jumps [100] in $\mathcal{O}(k)$ time for Hamming distance and in $\mathcal{O}(k^2)$ for edit distance [152].

In case of Hamming distance, the total number of extended transitions is still $\mathcal{O}(N_1B_2 + N_2B_1)$ (the number of transition checks does not change). In case of edit distance, it can increase by a factor of k – for a single pair of transitions l_1, l_2, we can produce up to $2k + 1$ weighted transitions.

After that, once again, we can use Dial's implementation of Dijkstra's algorithm [78] to find the smallest weight path in this graph obtaining the claimed result. □

One may notice that the modifications to the standard intersection algorithms from this section and the previous one are independent – in the previous section those consisted of marking additional nodes as starting/accepting, while in this section those consisted of adding edges.

3.8 PRACTICAL PANGENOME COMPARISON

3.8.1 BREAKPOINT MATCHING STATISTICS

The *Breakpoint Matching Statistics* between two *ED* strings $\widetilde{T}_1$ and $\widetilde{T}_2$ is a restriction of the Matching Statistics that refers to the notion of *breakpoint* in the genome rearrangement literature; see [36]. An *ED* string $\widetilde{T}$ representing a pangenome results

from a multiple sequence alignment of several genomes with the length n of $\widetilde{T}$ corresponding to the loci from where either multiple variants or a conserved fragment begin: these are the *breakpoints* that the alignment has detected. The assumption underlying Breakpoint Matching Statistics is that biologically relevant fragments in pangenomes are those that begin at a breakpoint. The *Breakpoint Matching Statistics* between $\widetilde{T}_1$ of length n_1 and $\widetilde{T}_2$ of length n_2, is an array $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}$ of length n_1 storing, for each $i \in [1..,n_1]$, the length of the longest local match between $\widetilde{T}_1$ and $\widetilde{T}_2$ that is a prefix of a string in $\mathcal{L}(\widetilde{T}_1[i..n_1])$ and hence starts at a breakpoint in $\widetilde{T}_1$, with the additional constraint that this must be part of a match that starts at a breakpoint in both $\widetilde{T}_1$ and $\widetilde{T}_2$ and ends at a breakpoint in at least one of the two *ED* strings.

Formally, for any two *ED* strings, $\widetilde{T}_1$ and $\widetilde{T}_2$, a *breakpoint match* (b-match) of $\widetilde{T}_1$ and $\widetilde{T}_2$, for some $1 \le i_1 \le i_2 \le n_1$ and $1 \le j_1 \le j_2 \le n_2$, is a string S such that $S \in \mathcal{L}(\widetilde{T}_1[i_1..i_2])$ and $S \in \mathcal{L}(\widetilde{T}_2[j_1..j_2])$. Intuitively, S starts *and* ends at a breakpoint in both $\widetilde{T}_1$ and $\widetilde{T}_2$.

A string S is a *left-breakpoint match* (lb-match) if **(i)** $S \in \mathcal{L}(\widetilde{T}_1[i_1..i_2])$ and S is a prefix of a string in $\mathcal{L}(\widetilde{T}_2[j_1..n_2])$ *or* **(ii)** S is a prefix of a string in $\mathcal{L}(\widetilde{T}_1[i_1..n_1])$ and $S \in \mathcal{L}(\widetilde{T}_2[j_1..j_2])$. Intuitively, S begins at a breakpoint in both *ED* strings and ends at a breakpoint in at least one of the two *ED* strings. We denote this specific alignment of S by $S^{i_2,j_2}_{i_1,j_1}$. Note that any b-match is also an lb-match.

Let us now formalize the problem of computing the Breakpoint Matching Statistics that we solve in this section.

Problem 16. BREAKPOINT MATCHING STATISTICS
Input: Two *ED* strings, $\widetilde{T}_1$ of length n_1, cardinality B_1 and size N_1, and $\widetilde{T}_2$ of length n_2, cardinality B_2 and size N_2.
Output: For each $i \in [1..,n_1]$, the length $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[i]$ of a longest prefix P of a string in $\mathcal{L}(\widetilde{T}_1[i..n_1])$ that can be left-extended with a string from $\mathcal{L}(\widetilde{T}_1[i_1..i-1])$ into an lb-match $S^{i_2,j_2}_{i_1,j_1}$, for some i_1, i_2, j_1, j_2.

The motivation for allowing a left-extension to an lb-match and not forcing P to be an lb-match is to maintain the property of the Matching Statistics on standard strings of not decreasing rapidly from $\mathrm{MS}_{T_1,T_2}[i]$ to $\mathrm{MS}_{T_1,T_2}[i+1]$.

Example 89. Let $\widetilde{T}_1 = \left\{ \begin{array}{c} \mathrm{A} \\ \mathrm{AC} \\ \mathrm{TGCT} \end{array} \right\} \cdot \left\{ \begin{array}{c} \varepsilon \\ \mathrm{CA} \end{array} \right\}$ and $\widetilde{T}_2 = \left\{ \begin{array}{c} \varepsilon \\ \mathrm{T} \end{array} \right\} \cdot \left\{ \begin{array}{c} \mathrm{GCA} \\ \mathrm{AC} \end{array} \right\}$ as in Example 82. We have that $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[1] = 2$, because $P = \mathrm{AC}$ starting at position $i = 1$ is equal to $S = \mathrm{AC}$, which is a b-match (for $i_1 = i_2 = 1$ and $j_1 = j_2 = 2$) and hence an lb-match. We also have $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[2] = 1$, because $P = \mathrm{C}$ starting at position $i = 2$ is left-extended to $S = \mathrm{AC}$, which is an lb-match (for $i_1 = 1$, $i_2 = 2$ and $j_1 = j_2 = 2$). For both cases ($i = 1$ and $i = 2$), we have that P is the longest possible such prefix.

We now show that the intersection graph G of $\widetilde{T}_1$ and $\widetilde{T}_2$ can also be used to efficiently compute their Breakpoint Matching Statistics. Indeed, the local matches that the Breakpoint Matching Statistics account for can be characterized in the intersection graph G of $\widetilde{T}_1$ and $\widetilde{T}_2$ as the common substrings that start at a node

(i, j) where i is an explicit state of $\widetilde{T}_1$, and j can be either explicit or implicit in $\widetilde{T}_2$. Additionally, (i, j) must be *reachable* in G from a node (i_1, j_1) where i_1 and j_1 are both explicit: they correspond to a common breakpoint of the two pangenomes. Moreover, if such a substring ends at node (i_2, j_2), then at least one state among i_2 and j_2 must be explicit. Notice that, should j be an explicit node, then the reachability condition above can be fulfilled by $j = j_1$ itself; in that case we also have $i = i_1$. On the other hand, if j is implicit, then it must be that $i_1 \neq i$ and $j_1 \neq j$.

Figure 3.8 shows the computation of the Breakpoint Matching Statistics in the intersection graph G of our running example. For example, for $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[1]$, we use the occurrence of AC (blue edge) starting at a node that corresponds to a pair of explicit states and ending at a node that also corresponds to a pair of explicit states (a breakpoint for both $\widetilde{T}_1$ and $\widetilde{T}_2$). No longer match satisfies these conditions; hence, we set $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[1] = 2$.

Notice that the Breakpoint Matching Statistics require more restricted matches than *ED* Matching Statistics. Indeed we have that for any two *ED* strings $\widetilde{T}_1$ and $\widetilde{T}_2$, it holds that $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[i] \leq \mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[i]$ for all $1 \leq i \leq n_1$. It should be clear that BREAKPOINT MATCHING STATISTICS can be computed within the same complexities as the ones described in Lemma 83. We thus obtain the following:

Theorem 90. *The BREAKPOINT MATCHING STATISTICS problem can be solved in $\mathcal{O}(N_1B_2 + N_2B_1)$ time by using an intersection graph of $\widetilde{T}_1$ and $\widetilde{T}_2$, which can be constructed within the same complexity.*

3.8.2 OUR MEASURES FOR COMPARING PANGENOMES

In this section we describe a similarity and a distance measure for pangenome comparison. These measures can be derived from either the MS or the BMS array. We assume that these have been pre-computed.

We consider both arrays $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}$ and $\mathrm{MS}_{\widetilde{T}_2,\widetilde{T}_1}$ (resp. $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}$ and $\mathrm{BMS}_{\widetilde{T}_2,\widetilde{T}_1}$) as the Matching Statistics is not *per se* a symmetric notion: the two arrays do not even need to have the same size (when $n_1 \neq n_2$). A simple solution for a similarity measure is to consider the sum of the two averages: each array is turned into a number being the average of its values, and the sum makes everything symmetric. Thus, we define the *similarity measure* $\mathcal{MS}(\widetilde{T}_1, \widetilde{T}_2)$ (resp. $\mathcal{BMS}(\widetilde{T}_1, \widetilde{T}_2)$) between $\widetilde{T}_1$ and $\widetilde{T}_2$ as follows:

$$\mathcal{MS}(\widetilde{T}_1, \widetilde{T}_2) = \mathcal{MS}(\widetilde{T}_2, \widetilde{T}_1) = \frac{\sum_{i\in[1..,n_1]} \mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}[i]}{n_1} + \frac{\sum_{j\in[1..,n_2]} \mathrm{MS}_{\widetilde{T}_2,\widetilde{T}_1}[j]}{n_2}$$

and

$$\mathcal{BMS}(\widetilde{T}_1, \widetilde{T}_2) = \mathcal{BMS}(\widetilde{T}_2, \widetilde{T}_1) = \frac{\sum_{i\in[1..,n_1]} \mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[i]}{n_1} + \frac{\sum_{j\in[1..,n_2]} \mathrm{BMS}_{\widetilde{T}_2,\widetilde{T}_1}[j]}{n_2}$$

We now move further in order to design a notion of *distance* between pangenomes based on $\mathcal{MS}(\widetilde{T}_1, \widetilde{T}_2)$ (resp. $\mathcal{BMS}(\widetilde{T}_1, \widetilde{T}_2)$). Unlike the notion of similarity, the distance has to decrease when the two pangenomes get more similar; hence, following

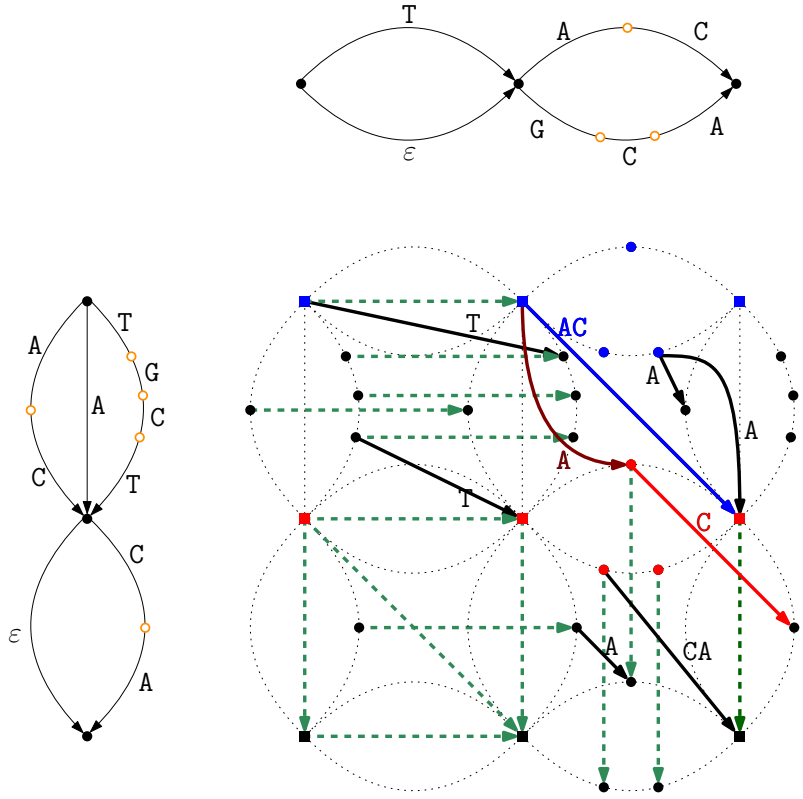


Figure 3.8: Breakpoint Matching Statistics computation in the intersection graph G of $\widetilde{T}_1$ and $\widetilde{T}_2$. To compute $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[1]$, the candidate starting nodes of the match in G are those in blue: nodes (i, j) where i is an explicit state of $\widetilde{T}_1$ in the uppermost dotted copy of A_2, and j is either an explicit state of $\widetilde{T}_2$ (squared blue nodes) or an implicit one (circled blue nodes). Note that TGC is the longest match that starts at the first set of $\widetilde{T}_1$ but it does not fulfill the conditions for the Breakpoint Matching Statistics because it does not end at any breakpoint; for the same reason, TG is also not a good candidate match. The occurrence of AC corresponding to the blue edge starts at a blue square node; hence it is reachable from the node itself that corresponds to a pair of explicit states, and it ends at a node that is again a pair of explicit states, and hence a breakpoint for both $\widetilde{T}_1$ and $\widetilde{T}_2$. There is no longer match satisfying these conditions; therefore we set $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[1] = 2$. For $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[2]$ we do the same but use as starting nodes those in red that correspond to the internal explicit node of A_1 paired with any node of A_2 (i.e. the nodes of the middle dotted copy of A_2). The red path spelling C: (i) is a prefix in $\widetilde{T}_1[2]$ starting at an explicit node of $\widetilde{T}_1$; (ii) is reachable from a square node in G by spelling A in both strings (curved brown red edge labeled with A); and (iii) ends where $\widetilde{T}_2[2]$ does, that is, at a breakpoint. Since this is the longest such path in G, we set $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[2] = 1$. Note, for example, that the match CA that occurs in $\widetilde{T}_1[2]$ and inside $\widetilde{T}_2[2]$ cannot be used for $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}[2]$ because it starts at a node that is not reachable from a pair of explicit nodes, meaning that it is not upperbounded by a breakpoint in $\widetilde{T}_2$. Computing $\mathrm{BMS}_{\widetilde{T}_2,\widetilde{T}_1}$, which is of size $n_2 = 2$, can be done in a dual manner on the very same graph, using as starting nodes those of the leftmost dotted copy of A_1 for $\mathrm{BMS}_{\widetilde{T}_2,\widetilde{T}_1}[1] = 2$ (obtained by traversing an ε-transition and then AC), and those of the middle dotted copy of A_1 for $\mathrm{BMS}_{\widetilde{T}_2,\widetilde{T}_1}[2] = 2$ (AC again).

a standard procedure, we invert the similarity measure while normalizing over the logarithm of the size of the pangenome. The reason for the normalization is that the values inside arrays $\mathrm{MS}_{\widetilde{T}_1,\widetilde{T}_2}$ and $\mathrm{BMS}_{\widetilde{T}_1,\widetilde{T}_2}$ are affected by the sizes of both strings – a very short *ED* string cannot contain a long match. Therefore, for a given $\widetilde{T}_1$, to account for the size N_2 of $\widetilde{T}_2$, we normalize $\mathcal{MS}(\widetilde{T}_1,\widetilde{T}_2)$ (resp. $\mathcal{BMS}(\widetilde{T}_1,\widetilde{T}_2)$) by $\log N_2$ and then invert[5], thus obtaining $\log N_2/\mathcal{MS}(\widetilde{T}_1,\widetilde{T}_2)$ (resp. $\log N_2/\mathcal{BMS}(\widetilde{T}_1,\widetilde{T}_2)$). Again, this gives rise to a non-symmetric notion, while symmetry is a desired property for a distance. Another desired property is reflexivity, requiring any pangenome to have distance zero from itself. The latter can be ensured by subtracting[6] a "correction term" as follows:

$$\overline{d}(\widetilde{T}_1,\widetilde{T}_2)=\frac{\log N_2}{\mathcal{MS}(\widetilde{T}_1,\widetilde{T}_2)}-\frac{\log N_1}{\mathcal{MS}(\widetilde{T}_1,\widetilde{T}_1)}.$$

and

$$\overline{bd}(\widetilde{T}_1,\widetilde{T}_2)=\frac{\log N_2}{\mathcal{BMS}(\widetilde{T}_1,\widetilde{T}_2)}-\frac{\log N_1}{\mathcal{BMS}(\widetilde{T}_1,\widetilde{T}_1)}.$$

thus guaranteeing that $\overline{d}(\widetilde{T}_1,\widetilde{T}_1)=\overline{bd}(\widetilde{T}_1,\widetilde{T}_1)=0$ for any nonempty $\widetilde{T}_1$. However, both $\overline{d}$ and $\overline{bd}$ are not symmetric yet, and hence, we finally ensure that our distance is *symmetric* resorting again to the sum. Therefore, we set:

$$d(\widetilde{T}_1,\widetilde{T}_2)=d(\widetilde{T}_2,\widetilde{T}_1)=\overline{d}(\widetilde{T}_1,\widetilde{T}_2)+\overline{d}(\widetilde{T}_2,\widetilde{T}_1).$$

and

$$bd(\widetilde{T}_1,\widetilde{T}_2)=bd(\widetilde{T}_2,\widetilde{T}_1)=\overline{bd}(\widetilde{T}_1,\widetilde{T}_2)+\overline{bd}(\widetilde{T}_2,\widetilde{T}_1).$$

Our *d* and *bd* distances resemble an analogous widely used distance measure for standard sequences often referred to as *kmacs* ([154, 204, 206]) that is based on standard string Matching Statistics.

The practical applications of the notions from this sections are tested on real data and discussed in [96].

3.9 EXPERIMENTS

We implemented the $\mathcal{O}(N_1B_2+N_2B_1)$-time algorithm for solving EDSI as well as an $\mathcal{O}(N_1N_2)$-time algorithm for EDSI based on the classic product automaton construction. We also implemented the $\mathcal{O}(N_1B_2+N_2B_1)$-time algorithm computing both Matching Statistics notions as well as the downstream similarities and distance measures for any two *ED* strings. All implementations were written in C++ and the source code is freely available at https://github.com/urbanslug/junctions. We compiled all implementations with gcc version 12.2.0 at optimization level (-O3).

[5] We assume that $\mathcal{MS}(\widetilde{T}_1,\widetilde{T}_2),\mathcal{BMS}(\widetilde{T}_1,\widetilde{T}_2)>0$.

[6] For a nonempty $\widetilde{T}_1$, the quantities $\mathcal{MS}(\widetilde{T}_1,\widetilde{T}_1)$ and $\mathcal{BMS}(\widetilde{T}_1,\widetilde{T}_1)$ are always greater than zero.

3.9.1 EFFICIENCY ON SIMULATED DATA

In this section, we compare the running time of our EDSI with that of the naïve method and with the parameters that define the characteristics of the input ED strings, with the purpose of validating the time efficiency of our algorithm and show how it actually improves in practice with respect to the baseline quadratic running time. In order to do that, we use synthetic data generated on the alphabet $\{\mathrm{A},\mathrm{C},\mathrm{G},\mathrm{T}\}$.

The synthetic ED strings were generated using another tool of ours named SIMED (https://github.com/urbanslug/simed). The tool starts by generating a random standard string of length W over the DNA alphabet, assuming a uniform distribution of letters. This is considered to be an initial sequence. We can view this as an ED string with $N = W$ and $n = B = 1$. The SIMED tool assumes a very simple evolutionary model (where each position has an equal chance of mutating, and each letter has the same probability of occurring at any position) and generates an ED string from the initial sequence based on the following input parameters:

- W as the length of the initial random (not ED yet) string;
- d as the number of positions where a set of multiple strings occurs, given as a percentage of W (that is, d is the fraction of degenerate positions);
- S as the maximum number of strings in any set of the resulting ED string;
- L as the maximum length of the strings in any set of the resulting ED string.

As aforementioned, the tool first generates a standard string uniformly at random, which we denote by X ($|X| = W$). It then samples $\delta = \lceil dW \rceil$ non-overlapping length-L substrings of X uniformly at random. We denote these by $X[i_1..i_1+L-1],\ldots,X[i_\delta..i_\delta+L-1]$, where $i_j + L \leq i_{j+1}$ for $j \in [1..,\delta-1]$. For every $j \in [1..,\delta]$, it picks a uniformly random integer s from $[1..,S]$ and produces $s-1$ strings of uniformly random lengths from $[0..L]$, each string generated uniformly at random; these $s-1$ strings and the original fragment $X[i_j..i_j+L-1]$ form a set D_{i_j} of strings. Finally it sets $\widetilde{T}$ as $X[1..i_1-1]\cdot D_{i_1}\cdot X[i_1+L..i_2-1]\cdot D_{i_2}\cdots D_{i_\delta}\cdot X[i_\delta+L..W]$. Note that $\widetilde{T}$ is indeed an ED string; we denote its length, cardinality, and size by n,B,N, respectively. If we choose d,S,W such that $(\delta+\delta\cdot S) \ll W$ then we have that $B \leq (\delta+\delta\cdot S) \ll W \leq N \implies B \ll N$. It is worth noting that if the same initial string X is used to generate two distinct ED strings, then X will appear in their (nonempty) intersection.

Starting from the same base sequence X of length W, in each experiment described in this section, we used the same parameters d,L and S to generate both $\widetilde{T}_1$ and $\widetilde{T}_2$. Thus, X guarantees a nonempty intersection between $\widetilde{T}_1$ and $\widetilde{T}_2$, and both implementations always answered YES (as expected) without a premature ending (hence, detecting their worst-case running time). As expected, the improved $\mathcal{O}(N_1B_2+N_2B_1)$-time implementation of EDSI was faster than the naïve $\mathcal{O}(N_1N_2)$-time implementation in all datasets, especially with longer variants and/or with short widely interspersed variants, that is for ED strings where $B \ll N$. Results are shown below.

We report a table for each set of parameters, and in each table, we show the data for different starting synthetic string lengths W, up to $|W|=100$k bases. The data reported in the columns of Tables 3.1-3.2 are: the length W of the initial string, the size N_1 and cardinality B_1 of the first synthetic ED string, the size N_2 and cardinality B_2 of the second synthetic ED string, and the time taken by the Naïve method and by EDSI, both measured in seconds. The parameters d (frequency of positions with multiple variants), S (maximum number of variants in such positions), and L (maximum length of such variants) determine the degree of *degeneracy* of the ED strings. As shown below, we have $G \ll N$ because (i) wherever the sequence is not degenerate, N grows linearly with W while G is constant, and (ii) wherever there is a degenerate position, $N \in \mathcal{O}(S \times L)$ while $G \in \mathcal{O}(S)$. This explains why our $\mathcal{O}(N_1B_2 + N_2B_1)$-time algorithm is much faster than the $\mathcal{O}(N_1N_2)$-time one.

Table 3.1: Results with simulation parameters: $d = 0.1\%$ with $S = 3$, $L = 3$ and with $S = 5$, $L = 5$.

W	N_1	B_1	N_2	B_2	Naïve (s)	EDSI (s)
		$S = 3$	$L = 3$			
10k	10019	36	10023	38	0.69	0.04
30k	30062	107	30071	107	6.20	0.14
50k	50106	173	50110	172	17.57	0.29
100k	100225	354	100203	344	72.81	0.47
		$S = 5$	and $L = 5$			
10k	10084	49	10066	50	0.68	0.06
30k	30198	144	30212	148	6.21	0.16
50k	50381	244	50358	250	18.00	0.29
100k	100837	515	100776	500	74.04	0.65

The tables show that the Naïve scales quadratically in the size of the ED strings while EDSI is much faster as $B \ll N$. A comparison of the second and third experiments reported in Tables 3.2 highlights how, when only L grows (it doubles from 5 to 10 while d and S remain 1% and 5, respectively), our tool has basically the same performance whereas the Naïve becomes slower. The explanation is that when L grows, only N grows while B does not (as we can see), and hence B and N diverge even more. Finally, we remark that the parameter that most affects B/N (i.e. the ratio of our asymptotic gain with respect to the Naïve) is d, as the comparison of Tables 3.1 and 3.2 shows for the corresponding values of S and L.

These experiments were conducted on a laptop with a 64 bit 11th Gen Intel(R) Core(TM) i7-11800H 8 core processor and 16 GB of RAM. Timings were recorded using std::chrono from the C++ standard library.

3.9.2 EFFICIENCY ON HUMAN GENOME DATA

In this section, we present some experiments for the running time of EDSI on real human genome data with variants. The goal is to show that our tool is fast even on real data, as the ratio between B and N is not too large for real pangenomes built out of real human genome fragments and their Variant Call Format (VCF) data. We built

Table 3.2: Simulation parameters: $d = 1\%$ with $S = 3$, $L = 3$, with $S = 5$, $L = 5$, and with $S = 5$, $L = 10$.

W	N_1	B_1	N_2	B_2	Naïve (s)	EDSI (s)
		$S = 3$	$L = 3$			
10k	10218	346	10232	348	0.70	0.05
30k	30688	1064	30659	1040	6.46	0.21
50k	51155	1758	51104	1752	18.84	0.48
100k	102227	3469	102258	3497	77.86	1.71
		$S = 5$	$L = 5$			
10k	10838	504	10796	494	0.80	0.06
30k	32362	1479	32415	1505	7.36	0.25
50k	54098	2508	54146	2525	20.84	0.56
100k	108071	4987	107947	4986	84.62	1.89
		$S = 5$	$L = 10$			
10k	11696	498	11803	500	0.96	0.06
30k	35405	1531	35140	1495	8.83	0.25
50k	58745	2503	58659	2484	25.22	0.59
100k	117444	4985	117417	4989	101.10	1.97

ED strings for human genome data from the GRCh38.p13 dataset, specifically from HLA-B in chromosome VI as well as the actin beta (ACTB) gene in chromosome VII. We used human genomic sequence data in the FASTA format and variation data in the Variant Call Format (.vcf file) from the following three databases: 1000G [119], TOPmed [118], and gnomAD [117].

The human leukocyte antigen (HLA) gene is contained in the major histocompatibility complex on the p arm (chromosomal region 6p21.33) of Chromosome VI which is known to be one of the most polymorphic regions of the human genome. The HLA gene is involved in immune system regulation ([74, 185]) and is found in the region between positions 31,353,872 and 31,367,067 (hence it is 13kb long). ACTB is a highly conserved gene in humans that codes for several proteins involved in cell structure and integrity. For each genome fragment (HLA and ACTB data), and for each database (out of the three 1000G, TOPmed, and gnomAD), we selected from the .vcf file only the variants that are recorded in that specific database, and we updated the regions containing variation, as denoted in the .vcf file into sets containing both the original in the reference and the variants contained in .vcf, thus creating an *ED* string. We performed this in two ways: one for all variants of that database for that genome fragment, and one by selecting the SNPs variants only. We then used AEDSO (https://github.com/urbanslug/aedso) to build the *ED* strings. Data download links and commands used are available at https://github.com/urbanslug/junctions/blob/master/test_data/human.org.

In summary, we have two *ED* strings (one with all variants and one with SNPs only) per each database, and each genome fragment. We remark that all of these *ED* strings include the original non-mutated string in the language; hence for each pair *the intersection is nonempty*. This ensures detecting a running time of EDSI

without a premature ending due to empty intersection: we ran EDSI for all pairs. For the HLA data, table 3.3 shows the sizes (and types) of the ED strings and the running times of a few of these experiments. Table 3.4 shows results for the ACTB data. We observe that EDSI improves over Naïve by one order of magnitude whenever $B \ll N$ (1000G and gNomad variants datasets), and still improves over the Naïve even when N/B is a small constant, like with TOPMed data.

Table 3.3: ED strings with databases and VCF and sizes, and time (in seconds) required by Naïve and by EDSI for HLA data.

DB_1	VCF_1	N_1	B_1	DB_2	VCF_2	N_2	B_2	Naïve (s)	EDSI (s)
1000G	all	13332	224	1000G	SNP	13247	161	1.25	0.05
TOPMed	all	15090	3452	gNomad	SNP	13941	1785	2.11	1.06
gNomad	all	14436	2044	TOPMed	SNP	14355	3170	2.10	1.13

Table 3.4: ED strings with databases and VCF and sizes, and time (in seconds) required by Naïve and by EDSI for ACTB data.

DB_1	VCF_1	N_1	B_1	DB_2	VCF_2	N_2	B_2	Naïve (s)	EDSI (s)
1000G	all	37019	644	gNomad	SNP	37876	3146	9.82	0.53

Finally, to conduct an experiment on these data with larger inputs, we picked a larger fragment of reference from Chromosome VI (spanning over the HLA region) of length 100Kb, and we repeated the same procedure as above. Table 3.5 presents the results of the experiment. We observe that even for these longer ED strings, EDSI is generally significantly faster than the Naïve method, especially on data such as that of the 1000G variants dataset – therein the ratio between N and B is larger than in the other data.

Table 3.5: ED strings with databases and VCF and sizes, and time (in seconds) required by Naïve and by EDSI for a fragment of data.

DB_1	VCF_1	N_1	B_1	DB_2	VCF_2	N_2	B_2	Naïve (s)	EDSI (s)
1000G	all	100951	3730	1000G	SNP	101252	2753	73	1.12
TOPMed	all	113111	21253	TOPMed	SNP	108669	18931	99.04	21.44
gNomad	all	130918	42572	gNomad	SNP	117793	38877	150.72	109.83

These experiments were conducted on a laptop with a 64 bit 11th Gen Intel(R) Core(TM) i7-11800H 8 core processor and 16 GB of RAM. Timings were recorded using $\mathrm{std::chrono}$ from the $\mathrm{C++}$ standard library.

3.10 Conclusion and future work

In this chapter, we focused on ED string comparison. We gave conditional lower bounds and introduced *intersection-graph* as a tool to compute the intersection of the languages from two ED strings. Then, we showed that those graphs can be used for several more practical comparison metrics. The main open questions are:

- We showed an $\widetilde{\mathcal{O}}(n_2N_1^{\omega-1}+n_1N_2^{\omega-1})$-time algorithm for EDSI. Can one design an $\mathcal{O}(n_2N_1^{\omega-1-\varepsilon}+n_1N_2^{\omega-1-\varepsilon})$-time (perhaps not combinatorial) algorithm for EDSI, for some $\varepsilon > 0$?
- We showed that there is no combinatorial $\mathcal{O}((N_1+N_2)^{1.2-\varepsilon}f(n_1,n_2))$-time algorithm for EDSI. Can one show a stronger conditional lower bound for combinatorial algorithms?
- We showed an $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$-time algorithm for outputting a representation of the intersection language of two unary *ED* strings. Can one design an $o(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$-time algorithm?

4

Indexing

4.1 Introduction

This chapter is based on [94].

In the real world, strings are often encoded with some level of uncertainty; for example, due to: (i) imprecise, incomplete or unreliable data measurements, such as sensor measurements, RFID measurements or trajectory measurements [12]; (ii) deliberate flexible sequence modeling, such as the representation of a *pangenome*, that is, a collection of closely-related genomes to be analyzed together [70]; or (iii) the existence of confidential information in a dataset that has been distorted deliberately for privacy protection [11].

While there are many practical solutions for text indexing [28, 92, 99, 108, 163] and also for answering different types of queries on various types of uncertain data (see Section 4.6), *practical indexing schemes* for uncertain strings are rather undeveloped. In response, our work makes an important step towards developing such practical space-efficient indexes.

4.1.1 Our data model and motivation

We use the standard *character-level uncertainty model* [130]. An *uncertain string* (or *weighted string*) X of length n on an alphabet Σ is a sequence of n sets. Every $X[i]$, for all $i \in [1\,..\,n]$, is a set of $|\Sigma|$ ordered pairs $(\alpha, p_i(\alpha))$, where $\alpha \in \Sigma$ and $p_i(\alpha)$ is the probability that letter α occurs at position i. In bioinformatics, weighted strings are known as *position weight matrices* [136].

Example 91. The table below shows a weighted string X, with $n = 6$ and $\Sigma = \{\mathrm{A}, \mathrm{G}\}$, represented as a $|\Sigma| \times n$ matrix.

	1	2	3	4	5	6
A	1	1/2	**3/4**	4/5	**1/2**	1/4
G	0	1/2	1/4	**1/5**	1/2	3/4

The data model of [130] has been employed by many works [18, 104, 142, 144, 155, 176]. As in these works, we define the *occurrence probability* of pattern $P = \mathrm{AGG}$ at position 3 in X of Example 91 as $3/4 \times 1/5 \times 1/2 = 3/40$ (shown in bold).

Weighted Indexing. The *Weighted Indexing* problem is defined as follows [19]: Given a weighted string X of length n on an alphabet Σ of size σ and a weight threshold $\frac{1}{z} \in (0, 1]$, preprocess X into a *compact* data structure (*the index*) that supports *efficient* pattern matching queries; i.e., report all positions in X where P occurs with probability at least $\frac{1}{z}$.

State of the Art. The indexing problem on weighted strings has attracted a lot of attention by the theory community [19, 34, 35, 47, 53, 123], culminating in the following bounds [34, 35]: there exists an index of $\mathcal{O}(nz)$ size supporting optimal $\mathcal{O}(m + |\mathbf{Occ}|)$-time queries for any pattern of length m; it can be constructed in $\mathcal{O}(nz)$ time using $\mathcal{O}(nz)$ space. We call this index the *weighted suffix tree* (WST).

4

Our Motivation. Although these bounds are very appealing from a theoretical perspective, from a practical perspective, $\mathcal{O}(nz)$ size and construction space are *prohibitive* for large-scale datasets. Say we have an input weighted string of length $n = 10^9$ bytes, that $z = 100$, and that the constant in $\mathcal{O}(nz)$ is something small, like 20 bytes. Then we need around 2TBs of RAM to store the index for an input of 1GB! *We were thus motivated to seek space-query time trade-offs for indexing weighted strings.* In particular, we seek a parameterized version of *Weighted Indexing* for which we can have indexes of size smaller than $\mathcal{O}(nz)$. Ideally, we would also like to construct these smaller indexes using less than $\mathcal{O}(nz)$ space. A good such input parameter is a lower bound ℓ on the length m of any queried pattern. It is arguably a reasonable assumption to know ℓ in advance. For instance, in bioinformatics [129, 157, 213], the length of sequencing reads (patterns) ranges from a few hundreds to 30,000 [157]. Even when at most k errors must be accommodated for matching, at least one out of $k + 1$ fragments must be matched exactly. In natural language processing, the queried patterns can also be long [207]. Examples of such patterns are queries in question answering systems [111], *description queries* in TREC datasets [25, 38], and representative phrases in documents [165]. Similarly, a query pattern can be long when it encodes an entire document (e.g., a webpage in the context of deduplication [115]), or machine-generated messages [131].

4.1.2 OUR TECHNIQUES AND RESULTS

Let us first recall WST, the state-of-the-art index. In [35], Barton et al. showed that, for any weight threshold $\frac{1}{z}$, a weighted string X of length n can be transformed into a family $\mathcal{S}$ of z standard strings (each of length n), so that a pattern P occurs in X at position i with probability p only if P occurs at position i in $\lfloor p \cdot z \rfloor$ strings from $\mathcal{S}$. The authors have also shown how to index $\mathcal{S}$ using (a modified version of) suffix trees [212] resulting in WST: an index of total size $\mathcal{O}(nz)$ supporting pattern matching queries in the optimal $\mathcal{O}(m + |\mathrm{Occ}|)$ time. A more space-efficient, array-based version was also presented in [53]; it is known as the *weighted suffix array* (WSA).

Here we make the following three main contributions (see Figure 4.1 for an informal overview of our techniques):

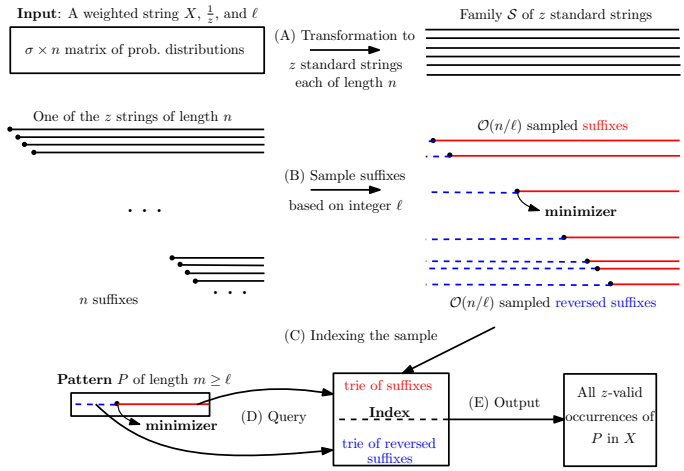


Figure 4.1: An informal overview of our techniques: Given a weighted string X of length n over an alphabet of size σ, a weight threshold $\frac{1}{z}$, and an integer ℓ, we **(A)** use the algorithm from [35] to construct a family $\mathcal{S}$ of z standard strings, each of length n. **(B)** For each such string, we consider all of its n suffixes and sample them for the given integer ℓ using the minimizers mechanism [184, 189]. These suffixes imply a set of $\mathcal{O}(n/\ell)$ suffixes and a set of $\mathcal{O}(n/\ell)$ reversed suffixes in expectation. **(C)** We then index these suffixes in two suffix trees [212], which we link using a 2D grid, so as to answer pattern matching queries for patterns of length at least ℓ. **(D)** When such a pattern P of length m is given, we find its leftmost minimizer, which implies a suffix and a reversed suffix of P, and query those using our index. Our index efficiently merges the partial results (i.e., occurrences of suffixes and reversed suffixes); and **(E)** outputs all z-valid occurrences of P in X. The size of the resulting index is $\mathcal{O}(\frac{nz}{\ell}\log z)$. The extra multiplicative $\log z$ factor comes from our representation of the edge labels in the suffix trees.

1. We show how to slash the size of both WST and WSA roughly by ℓ, while still supporting very fast queries in expectation for any pattern P of length $m \geq \ell$, by combining the *minimizers* sampling mechanism [184, 189], *suffix trees* [212], several combinatorial and probabilistic arguments, and a *geometric data structure* (2D grid) [162]. The size of the resulting index is $\mathcal{O}(\frac{nz}{\ell}\log z)$. The extra multiplicative $\log z$ factor comes from a new way to encode the edge labels in the suffix trees. The main novelty of our approach is the combination of minimizers and the new edge encoding allowing us to delete the z strings after the construction, thus resulting in a significantly smaller index size for large ℓ values.

2. Our technique still requires us to first construct the family $\mathcal{S}$ of the z strings, which in any case gives an index with $\mathcal{O}(nz)$ construction space. The challenge is how to construct the index *without* constructing the z strings. To achieve this, we develop a fast, highly non-trivial, algorithm for constructing the *same* index *without generating $\mathcal{S}$ explicitly*. Our new algorithm samples an implicit representation of $\mathcal{S}$ using minimizers outputting the final index directly. The construction space of the resulting index matches its size: $\mathcal{O}(\frac{nz}{\ell}\log z)$. This is the most technically involved result of our work.

3. Following the different paradigms of suffix trees [212] and suffix arrays [133] in the classic setting of standard strings, we have implemented tree and array-based versions of our index underlying Contributions 1 and 2. The results show that our indexes are up to *two orders of magnitude smaller* than the state of the art in terms of *both* index size and construction space. For instance, for indexing $1,432$ bacterial samples, with $\ell = 1024$ and $z = 128$, which are reasonable in applications, our space-efficient index has size 640MBs and needs only 772MBs of memory to be constructed, while WSA has size 7GBs and needs 32GBs of memory to be constructed and WST has size 126GBs and needs 241GBs of memory to be constructed! Our results also show that our array-based indexes outperform the tree-based ones, offering *very competitive query times and construction times* to those of the state of the art.

4.1.3 CHAPTER ORGANIZATION

In Section 4.2, we provide the necessary background. In Section 4.3, we present our index. In Section 4.4, we present the space-efficient algorithm for constructing our index. In Section 4.5, we present a more practical algorithm for querying our index. In Section 4.6, we discuss related work. In Section 4.7, we provide an extensive experimental evaluation of our algorithms. We conclude in Section 4.8 with a discussion on limitations and future work.

4.2 PRELIMINARIES AND PROBLEM DEFINITION

Sampling. Given a fixed pair of positive integers (ℓ, k) s.t. $\ell \geq k$, we call a function $f : \Sigma^\ell \to [1..\ell - k + 1]$ that selects the starting position of a length-k fragment, for any string of length ℓ, an (ℓ, k)-*local scheme*. We call the set $\mathcal{M}_f(S) = \{i + f(S[i..i + \ell - 1]) - 1 \mid 1 \leq i \leq |S| - \ell + 1\}$, for an (ℓ, k)-*local scheme* f on a string S, *the set of selected indices*. An (ℓ, k)-*minimizer scheme* is an (ℓ, k)-local scheme that selects the position of the leftmost occurrence of the smallest length-k substring, for a fixed k and a fixed order on Σ^k. In that case, we call *minimizers* the selected indices [184, 189]. The minimizer scheme can be based on a lexicographic order.

Example 92. Let $S = \mathrm{AGAAGG}$, and f be the $(4, 2)$-minimizer scheme for the standard lexicographical order. We obtain $\mathcal{M}_f(S) = \{3\}$ because $S[3..4] = \mathrm{AA}$ is the lexicographically smallest length-2 substring in all length-4 fragment of S.

The minimizer scheme can also be specified by a hash function, e.g., Karp-Rabin fingerprints [134].

Definition 93. Let f be an (ℓ, k)-minimizer scheme. The *specific density* of f on S is the value $|\mathcal{M}_f(S)|/|S|$. The *density* of f is the expected specific density on a sufficiently long random string (with letters chosen independently at random).

Lemma 94 ([219])**.** *The density of an (ℓ, k)-minimizer scheme on alphabet Σ with $k \geq \log_{|\Sigma|} \ell + c$ is $\mathcal{O}(\frac{1}{\ell})$, for some $c = \mathcal{O}(1)$.*

Weighted Strings. A *weighted string* X of length n on an alphabet Σ is a sequence of n sets $X[1], \ldots, X[n]$. Every $X[i]$, for all $1 \leq i \leq n$, is a set of $|\Sigma|$ ordered pairs $(\alpha, p_i(\alpha))$,

where $\alpha \in \Sigma$ is a letter and $p_i(\alpha)$ is the probability of having α at position i of X. Formally, $X[i] = \{(\alpha, p_i(\alpha)) \mid \alpha \in \Sigma\}$, where for every $\alpha \in \Sigma$ we have $p_i(\alpha) \in [0,1]$, and $\sum_{\alpha\in\Sigma} p_i(\alpha) = 1$. A letter α *occurs* at position i of a weighted string X if and only if $p_i(\alpha)$, the *occurrence probability* of α at position i, $p_i(\alpha)$, is greater than 0. A string U of length m is a *factor* of a weighted string X if and only if it occurs at some starting position i with *occurrence probability* $\mathbb{P}(X[i..i+m-1] = U) = \Pi_{j=1}^{m} p_{j+i-1}(U[j]) > 0$. Given a *weight threshold* $1/z \in (0,1]$, we say that factor U is *z-solid* (or *z-valid*) or equivalently that factor U has a z-solid occurrence in X at some position i, if $\mathbb{P}(X[i..i+m-1] = U) \geq 1/z$. When the context is clear we may simply say *solid* (or *valid*). We say that factor U is *(right-)maximal* at position i of X if U has a solid occurrence at position i of X and no string $U' = U\alpha$, for any $\alpha \in \Sigma$, has a solid occurrence at position i of X. For a weighted string X, a pattern P, and a weight threshold $1/z \in (0,1]$, $\mathbf{Occ}_{1/z}(P,X)$ is the set of starting positions of valid occurrences of P in X. A *property* Π of a string S is a hereditary collection (namely, a collection that contains all the subintervals of its elements) of integer intervals contained in $[1..n]$. For simplicity, we represent every property Π with an array $\pi[1..|S|]$ such that the longest interval $I \in \Pi$ starting at position i is $[i..\pi[i]]$. Observe that π can be an arbitrary array satisfying $\pi[i] \in [i-1, n]$, and $\pi[1] \leq \pi[2] \leq \cdots \leq \pi[n]$ (where $\pi[i] = i-1$ means that i is not contained in any interval $I \in \Pi$). For a string P, by $\mathbf{Occ}_\pi(P,S)$ we denote the set of occurrences of P in S such that $i + |P| - 1 \leq \pi[i]$.

Example 95. Consider the string-property pair (S_2, π_2) in Figure 4.2. The pattern $P = \mathrm{AAA}$ occurs at position $i = 3$ because $i + |P| - 1 = 3 + 3 - 1 \leq \pi_2[3] = 5$.

Let us consider an indexed family $\mathcal{S} = (S_j, \pi_j)_{j=1}^{k}$ of strings S_j with properties π_j. For a string P and an index i, by $\mathrm{Count}_S(P,i) = |\{j \in [1..k] \mid i \in \mathbf{Occ}_{\pi_j}(P, S_j)\}|$ we denote the total number of occurrences of P at position i in the strings $S_1, \ldots, S_k$ of $\mathcal{S}$ that respect the properties. We say that an indexed family $\mathcal{S} = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ is a *z-estimation* of a weighted string X of length n if and only if, for every string P and position $i \in [1..n]$, $\mathrm{Count}_S(P,i) = \lfloor \mathbb{P}(X[i..i+|P|-1] = P) \cdot z \rfloor$. The following result has been shown by Barton et al.:

Theorem 96 ([34]). *For any weighted string X of length n and any weight threshold $\frac{1}{z}$, X has a z-estimation of size $\mathcal{O}(nz)$ constructible in $\mathcal{O}(nz)$ time using $\mathcal{O}(nz)$ space.*

Example 97. For $\frac{1}{z} = \frac{1}{4}$, the weighted string X in Example 91 admits the 4-estimation $\mathcal{S}$ in Table 4.2, given by Theorem 96.

For pattern $P = \mathrm{AG}$ and S_3, we have that $\mathbf{Occ}_{\pi_3}(P, S_3) = \{1, 4\}$ because P occurs at position 1, with $1 + |P| - 1 \leq \pi_3[1] = 4$, and at position 4, with $4 + |P| - 1 \leq \pi_3[4] = 6$.

For pattern $P = \mathrm{AG}$ and $i = 1$, we have that $\mathbb{P}(X[i..i+|P|-1] = P) = 1 \cdot 1/2 = 1/2$ and so P occurs in $\mathrm{Count}_S(P,i) = \lfloor \mathbb{P}(X[i..i+|P|-1] = P) \cdot z \rfloor = \lfloor (1/2) \cdot 4 \rfloor = 2$ strings of the 4-estimation at position 1 (namely, strings S_3 and S_4).

We construct the set of (lexicographic) minimizers that respect the property, for $\ell = 3$ and $k = 2$, for every S_j, $j \in [1..4]$, from $\mathcal{S}$; namely for $S_j \in \mathcal{S}$ we compute $f(S_j[i..i+2])$ if and only if $i + 2 \leq \pi_j[i]$. We underline the positions of the minimizers. Note that we have selected no minimizer in S_1 or S_4 as they have no solid factor of length 3.

i	1	2	3	4	5	6
S_1	A	A	A	A	A	A
π_1	2	2	3	4	5	6
S_2	A	A	A	A	A	G
π_2	4	4	5	6	6	6
S_3	A	G	A	A	G	G
π_3	4	4	5	6	6	6
S_4	A	G	G	G	G	G
π_4	2	2	3	3	5	6

Figure 4.2: A 4-estimation $\mathcal{S}$ of X from Example 91.

Problem Definition. In our ℓ-WEIGHTED INDEXING problem, we are given a weighted string X of length n over an alphabet Σ, a weight threshold $\frac{1}{z} \in (0,1]$, and an integer $\ell > 0$, and we are asked to preprocess them in a *compact* data structure (the index) to support the following queries *efficiently*: For any string P of length $m \geq \ell$, report all elements of $\mathbf{Occ}_{\frac{1}{z}}(P,X)$. Other than the *index size* and the *query time*, we seek to minimize the *construction time* and the *construction space*.

4.3 THE NEW INDEX: MINIMIZER-BASED WST

In this section, we describe our index for solving ℓ-WEIGHTED INDEXING. We assume random access to X (we can discard X at the end of the construction). To simplify the analysis we assume X is over an alphabet of size $\sigma = \mathcal{O}(1)$.

Main Idea. We start the index construction by building a z-estimation of X, whose total size is $\Theta(nz)$ (Theorem 96). We then use minimizers sampling to select $\mathcal{O}(\frac{nz}{\ell})$ positions (Lemma 94) of the z-estimation, where ℓ is a predetermined lower bound on the length of the supported queries. Next, we construct two suffix trees, called *minimizer solid factor trees*: (1) the compacted trie *of all suffixes* of the solid factors in the z-estimation starting at the minimizer positions, and (2) the compacted trie *of all the reversed prefixes* of the solid factors in the z-estimation ending at the minimizer positions. To reduce the size of both trees, we discard the z-estimation using a combinatorial observation (Lemma 100 and Corollary 102) that allows us to store only $\mathcal{O}(\log z)$ information to label a suffix tree edge. This concludes the construction of the minimizer solid factor trees (Lemma 103). After that, we pair up the leaf nodes corresponding to the same minimizer position, from one of these trees to the other, using a 2D grid for *range reporting* [52] (Lemmas 7 and 104). This results in an index of expected total size $\mathcal{O}(n + \frac{nz}{\ell}\log z)$. Finally, we show how to query the index efficiently by using a probabilistic argument on the number of expected points returned by the 2D grid (Lemma 105). We thus arrive to Contribution 1 (Theorem 106).

Minimizer Solid Factor Trees. Let us fix a weighted string X of length n over an alphabet Σ and a weight threshold $\frac{1}{z}$. We first define a *forward solid factor tree* (resp. *backward solid factor tree*) for X as the suffix tree for the set of maximal solid factors (resp. the set of reversed solid factors) in X. By Theorem 96, we know that each such solid

factor appears in a z-estimation of size $\mathcal{O}(nz)$, and therefore both solid factor trees have size $\mathcal{O}(nz)$ as well. This argument also gives a method to construct the solid factor trees [34].

We adapt the solid factor trees to make them more space-efficient for ℓ-Weighted indexing by employing minimizer schemes. Let us fix ℓ, k and an (ℓ,k)-minimizer scheme f. We can assume throughout, from Lemma 94, that ℓ and k are chosen so that the density of f (see Def. 93) is $\mathcal{O}(\frac{1}{\ell})$. We then construct a z-estimation $\mathcal{S} = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ of X using Theorem 96 and compute the set $\mathcal{M}_X$ of minimizers from $\mathcal{S}$ respecting the property; namely for $S_j \in \mathcal{S}$ we compute $f(S_j[i..i+\ell-1])$ if and only if $i+\ell-1 \leq \pi_j[i]$.

We represent each minimizer in $\mathcal{M}_X$ by a pair (i,j), where i is the minimizer position in the string $S_j \in \mathcal{S}$. In the following, we consider $\mathcal{M}_X$ fixed with $|\mathcal{M}_X| = \mathcal{O}(\frac{nz}{\ell})$, as by Lemma 94 there are in expectation $\mathcal{O}(\frac{nz}{\ell})$ minimizers in $\mathcal{S}$.

Based on $\mathcal{M}_X$, we define a *minimizer* forward (resp. backward) solid factor tree as a compacted trie containing suffixes of solid factors (resp. of reversed solid factors) *starting* at position i from a string $S_j \in \mathcal{S}$ with $(i,j) \in \mathcal{M}_X$. Each leaf has a minimizer label $(i,j) \in \mathcal{M}_X$ associated to the corresponding suffix. If one same suffix corresponds to several such labels (it occurs at several minimizers from $\mathcal{S}$), we add one copy of the leaf for each such label. Since $|\mathcal{M}_X| = \mathcal{O}(\frac{nz}{\ell})$, the minimizer solid factor trees contain $\mathcal{O}(\frac{nz}{\ell})$ leaves, and therefore nodes.

Still the size of the z-estimation $\mathcal{S}$ is, by definition, always $\Theta(nz)$, which makes the total size of the index $\mathcal{O}(\frac{nz}{\ell}) + \Theta(nz) = \Theta(nz)$. We avoid this by employing the following crucial combinatorial observation on *heavy strings* [143]:

Definition 98. For any weighted string X, we call a *heavy string* H_X of X a string such that $H_X[i]$ is the letter having a largest probability in $X[i]$ (ties are broken arbitrarily).

Example 99. Let X be the weighted string of Example 91; a heavy string of X is $H_X = \mathrm{AGAAAG}$ (the tie at position 2 is broken for G and the tie at position 5 is broken for A).

Lemma 100 ([143])**.** *Let H_X be a heavy string of X. For a weight threshold $\frac{1}{z}$ and any z-solid factor U starting at position i and ending at position j of X, $d_H(U, H_X[i..j]) \leq \log_2 z$ holds.*

Example 101. Let X be the weighted string of Example 91; with $H_X = \mathrm{AGAAAG}$. If $z = 4$, $\log_2 z = 2$, so no solid factor has more than 2 mismatches with H_X at its occurrence position. The string AAGG is not valid at position 1 as it has 3 mismatches with H_X; indeed, its probability is $\frac{1}{40} < \frac{1}{4}$. The string AAAA is valid at position 1 with probability $\frac{3}{10} > \frac{1}{4}$, and has only one mismatch with H_X. The condition is however not sufficient: AGAG is not valid at position 1 (its probability is $\frac{3}{40} < \frac{1}{4}$), even if it has only one mismatch with H_X.

We directly get the following result, which allows us to avoid storing the z-estimation $\mathcal{S}$ explicitly.

Corollary 102. *Every solid factor of a weighted string X for a weight threshold $\frac{1}{z}$ can be characterized by an interval of the heavy string H_X plus the information of at most* $\log_2 z$

single mismatches. The minimizer solid factor tree can be implemented as a compacted trie whose edges store only that information, which takes $\mathcal{O}(\log z)$ extra space per edge.

We apply Corollary 102 to obtain Lemma 103. Based on this lemma, we construct the minimizer solid factor trees for X.

Lemma 103. *The (forward and backward) minimizer solid factor trees can be constructed in $\mathcal{O}(nz)$ time using $\mathcal{O}(nz)$ space. Each tree has $\mathcal{O}(\frac{nz}{\ell})$ expected nodes and its expected total size is $\mathcal{O}(\frac{nz}{\ell}\log z)$.* [1]

Proof. We apply Theorem 96 to construct a z-estimation for X in $\mathcal{O}(nz)$ time and space. The set of minimizers of any string can be computed in linear time [159]. Thus, computing $\mathcal{M}_X$ for the z-estimation takes $\mathcal{O}(nz)$ time. The compacted trie of any collection of substrings of a string can be constructed in linear time in the length of the string [53, 135], and thus the minimizer solid factor trees can be constructed in $\mathcal{O}(nz)$ time using $\mathcal{O}(nz)$ space. The number of nodes and the total size of the trees follow from Lemma 94 and Corollary 102. □

Exploiting 2D Range Reporting. We explain how to employ a geometric data structure to pair up the leaf nodes corresponding to the same minimizer position from one of the minimizer solid factor tree that we have constructed above to the other.

Let us write $\mathcal{T}_{\text{suff}}$ (resp. $\mathcal{T}_{\text{pref}}$) for the forward (resp. backward) minimizer solid factor tree. We fix an order on the leaves of both $\mathcal{T}_{\text{pref}}$ and $\mathcal{T}_{\text{suff}}$, such that for any node in one of the trees, the set of its descendant leaves forms an interval. This is possible, for example, by sorting the strings corresponding to the leaves in lexicographical order. Via this ordering, we can consider a pair of leaves from $\mathcal{T}_{\text{suff}}$ and $\mathcal{T}_{\text{pref}}$ as a point of a 2D data structure, which we call *the grid*.

We start by some definitions: (1) Given a string P, we denote by $\mathcal{I}_{\text{suff}}(P)$ (resp. $\mathcal{I}_{\text{pref}}(P)$) the (possibly empty) interval of leaves in the subtree obtained by reading P in $\mathcal{T}_{\text{suff}}$ (resp. $\mathcal{T}_{\text{pref}}$). (2) We denote by $\mathcal{N}$ the set of all those points corresponding to pairs of leaves from $\mathcal{T}_{\text{suff}}$ and $\mathcal{T}_{\text{pref}}$ with identical minimizer labels. Each point in $\mathcal{N}$ corresponds to a given minimizer $(i,j) \in \mathcal{M}_X$, and a pair of maximal solid factors in X that can be read from i, both right-wise and left-wise. (3) Given a pair P_1, P_2 of strings, we denote by $\mathcal{N}(P_1,P_2)$ the intersection of the set $\mathcal{N}$ with the rectangle $\mathcal{I}_{\text{suff}}(P_1)\times\mathcal{I}_{\text{pref}}(P_2)$. (4) Given a string P of length $m \geq \ell$, such that $f(P[1\,..\,\ell]) = \mu$, we denote $\mathcal{N}(P[\mu\,..\,m],(P[1\,..\,\mu])^r)$ by $\mathcal{N}(P)$.

Lemma 104. *For any pattern P of length m, with $n \geq m \geq \ell$, if P is a solid factor in X, then $\mathcal{N}(P)$ is nonempty. In particular, if P has a valid occurrence in X starting at position k then $\mathcal{N}(P)$ contains at least one point having label $(k-1+f(P[1\,..\,\ell]),j)$ for some $j \in [1\,..\,\lfloor z \rfloor]$.*

Proof. Let P be such a pattern, which is a solid factor in X at position k. By definition of a z-estimation, we know that P occurs at position k in some $S_j \in \mathcal{S}$. The minimizer computed for position k of S is $i = k-1+\mu$ with $\mu = f(P[1\,..\,\ell])$, since $S_j[k\,..\,k+\ell-1] = P[1\,..\,\ell]$. Therefore, the tree $\mathcal{T}_{\text{suff}}$ (resp. $\mathcal{T}_{\text{pref}}$) contains a leaf k_1 (resp. k_2) corresponding

[1] We claim $\mathcal{O}(nz)$ time and space during our construction because if $\log z > \ell$, we can abort the construction and resort to $\mathcal{O}(nz)$ size.

to the longest substring of S_j starting at position i respecting the property Π, which starts with $P[\mu\,..\,m]$ (resp. the longest reversed substring of S_j ending at position i respecting the property, which starts with $P[1\,..\,\mu]^r$). Those leaf nodes both have a label $(i,j) = (k-1+\mu, j)$, hence the corresponding point is in $\mathcal{N}(P)$, which proves the result. □

Our index (i.e., $\mathcal{T}_{\text{suff}}$, $\mathcal{T}_{\text{pref}}$, and the grid) solves ℓ-WEIGHTED INDEXING by answering 2D RANGE REPORTING queries [52]. In particular, we use Lemma 7.

Note that, even if each occurrence of a pattern can be detected with 2D RANGE REPORTING queries, the converse is not true: if a pattern U has a minimizer at position μ and both $U[1\,..\,\mu]$ and $U[\mu\,..\,|U|]$ are solid factors occurring at respective positions k and $k+\mu-1$, then a corresponding point will be detected with the 2D RANGE REPORTING queries, *even* if U is not a solid factor itself. In that case, U is by definition a substring of some $S_j \in \mathcal{S}$, but does *not* respect the property. We can simply compute all the points by 2D RANGE REPORTING, and check naively for false positives by comparing the pattern with X at the positions corresponding to these points. Conversely, one can have several points corresponding to a single occurrence, if the pattern appears in multiple strings in $\mathcal{S}$ at a same position, which could also increase the running time. To control the number of such additional checks (both for false positives and duplicate ones), we give a bound on the expected number of occurrences of a given pattern in the z-estimation $\mathcal{S}$:

Lemma 105. *For any string P chosen uniformly at random from Σ^m, there are $\mathcal{O}(nz/\sigma^m)$ points expected in $\mathcal{N}(P)$.*

Proof. $\mathcal{T}_{\text{suff}}$ and $\mathcal{T}_{\text{pref}}$ are constructed from a z-estimation $\mathcal{S}$, therefore each point in $\mathcal{N}(P)$ corresponds to an occurrence of P in $\mathcal{S}$ (it might not respect property Π however). Since $\mathcal{S}$ has $(n-m+1)\lfloor z \rfloor \leq nz$ substrings of length m, we have $\sum_{P\in\Sigma^m} |\mathcal{N}(P)| \leq nz$, and hence if P is chosen uniformly at random we obtain no more than $\frac{nz}{\sigma^m}$ points in expectation. □

Main Result. We arrive at the main result of the section:

Theorem 106. Let X be a weighted string of length n over an alphabet of size σ, $\frac{1}{z}$ be a weight threshold, and $\ell > 0$ be an integer. After $\mathcal{O}(nz)$ construction time and using $\mathcal{O}(nz)$ construction space, we can construct an index of $\mathcal{O}(n + \frac{nz}{\ell}\log z)$ expected size answering ℓ-WEIGHTED INDEXING queries of length $m \geq \ell$ in $\mathcal{O}(m + (1+\frac{nz}{\sigma^m})\log\frac{nz}{\ell})$ expected time.

Proof. We first construct the minimizer solid factor trees of X in $\mathcal{O}(nz)$ time and space; the trees have size $\mathcal{O}(\frac{nz}{\ell}\log z)$ (Lemma 103). We preprocess the pairs of leaves for 2D RANGE REPORTING queries in $\mathcal{O}(\frac{nz}{\ell})$ time (Lemma 7). When a pattern P of length $m \geq \ell$ is given, we compute its leftmost minimizer μ in $\mathcal{O}(\ell)$ time [159], compute the sides of the rectangle in $\mathcal{O}(m)$ time by spelling $P[\mu\,..\,m]$ in $\mathcal{T}_{\text{suff}}$ and $(P[1\,..\,\mu])^r$ in $\mathcal{T}_{\text{pref}}$, and answer a 2D RANGE REPORTING query in $\mathcal{O}(\log\frac{nz}{\ell}(1+|\mathcal{N}(P)|))$ time (Lemma 7). Finally, we must check, for every output point (i,j), for a valid occurrence around the ith minimizer of the jth string of the z-estimation. To do this efficiently (i.e., in $\mathcal{O}(\log z)$ time per point) *without storing the z-estimation of X*, we store only the $\log_2 z$

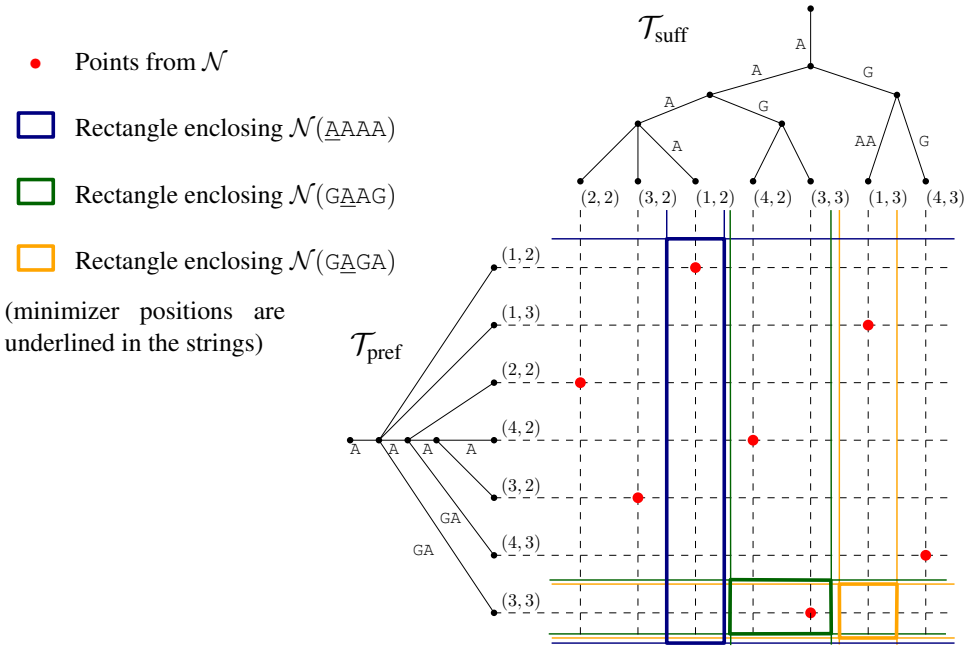


Figure 4.3: Our minimizer-based index for the weighted string from Example 91, $\frac{1}{z} = \frac{1}{4}$, and the minimizers from Example 107. The tree $\mathcal{T}_{\mathrm{suff}}$ is the forward minimizer solid factor tree and $\mathcal{T}_{\mathrm{pref}}$ is the backward one. Edges without labels are constructed for readability and mean that the parent and the children nodes correspond to the same string. Each leaf node representing the minimizer position i in string S_j is decorated with (i, j).

closest mismatching positions to the left and to the right of every minimizer in $\mathcal{M}_X$ (Lemma 100). The total verification time is thus $\mathcal{O}((\log z + \log(nz/\ell))(|\mathcal{N}(P)|) + 1) = \mathcal{O}(\log \frac{nz}{\ell}(|\mathcal{N}(P)|+1))$. By Lemma 105, we know that in expectation we have $|\mathcal{N}(P)| = \frac{nz}{|\Sigma|^m}$. We obtain an expected query time of $\mathcal{O}(m + (1 + \frac{nz}{|\Sigma|^m}) \log \frac{nz}{\ell})$. The total size is $\mathcal{O}(n + \frac{nz}{\ell} \log z)$, to store H_X plus the index. □

Example 107. Let X be the weighted string from Example 91 and $\frac{1}{z} = \frac{1}{4}$. The construction of our index is detailed in Figure 4.3, and query rectangles $\mathcal{N}(P)$ (resp. $\mathcal{N}(P')$ and $\mathcal{N}(P'')$) are constructed for patterns $P = \underline{\mathrm{A}}\mathrm{AAA}$ (resp. $P' = \mathrm{G}\underline{\mathrm{A}}\mathrm{AG}$ and $P'' = \mathrm{G}\underline{\mathrm{A}}\mathrm{GA}$) whose minimizer positions are underlined. The blue rectangle $\mathcal{N}(P)$ contains exactly one point which corresponds to a substring AAAA in S_2 that respects the property. This substring corresponds to an occurrence of P at position 1 with probability $1 \cdot \frac{1}{2} \cdot \frac{3}{4} \cdot \frac{4}{5} = 0.3 > \frac{1}{4}$ in X. The green rectangle $\mathcal{N}(P')$ contains one point, which corresponds to an occurrence of P' in S_3. However, this occurrence does not respect the property Π (because $i + |P'| - 1 = 2 + 4 - 1 = 5 > \pi_3[2] = 4$) and therefore is a false positive in X (it occurs with probability $0.15 < \frac{1}{4}$). Finally, the orange rectangle $\mathcal{N}(P'')$ does not contain any point, because the pattern does not occur in the z-estimation.

4.4 Space-efficient construction of the index

Recall that to construct the index in Section 4.3, we first construct a z-estimation, which temporarily takes $\Theta(nz)$ space during construction. In this section, we improve the space required for the construction of the index by designing a space-efficient algorithm for constructing a minimizer solid factor tree with only a moderate increase in the construction time.

Main Idea. We start the index construction by *simulating* the construction of an *extended solid factor tree*. This is a trie of the solid factors of X extended by H_X, the heavy string of X. In particular, we maintain the subtree induced by the solid factors that *start at minimizer positions* but discard the nodes that we do not need upon returning to their parents. We achieve this via traversing the full tree in a DFS order. Thus, even though the full tree size is $\mathcal{O}(nz)$ (Lemma 108), at any moment, we store *only* the current leaf-to-root path plus the actual output. Therefore, we use only $\mathcal{O}(n+\frac{nz}{\ell}\log z)$ expected space at a cost of $\mathcal{O}(nz\log\ell)$ time (Lemma 109). We next reverse this tree (the solid factors are read from leaf to root there, while in the minimizer solid factor tree those are read from root to leaf), in $\mathcal{O}(\frac{nz}{\ell}\log\frac{nz}{\ell}\log z)$ expected time, using $\mathcal{O}(\frac{nz}{\ell}\log z)$ expected space (Theorem 110). *Our approach thus trades construction time for less space*: in total, it adds up to $\mathcal{O}(nz\log\ell+\frac{nz}{\ell}\log\frac{nz}{\ell}\log z)$ expected construction time (instead of $\mathcal{O}(nz)$) but $\mathcal{O}(n+\frac{nz}{\ell}\log z)$ expected construction space (instead of $\mathcal{O}(nz)$). We thus arrive to Contribution 2.

Key Concepts. The string $U\cdot H_X[j+1\,..\,n]$ (resp. $(H_X[1\,..\,i-1]\cdot U)^r$) is called the *right extension of the solid factor U* (resp. *left extension of the solid factor U*), if U is a solid factor of X starting at position i and ending at position $j\geq i-1$.

For such a U, we define a *forward extended solid factor tree* of X as a trie of all the reversals of $U\cdot H_X[j+1\,..\,n]$, and a *backward extended solid factor tree* of X as a trie of all $H_X[1\,..\,i-1]\cdot U$. These trees can be constructed by looking only at the extensions of *maximal* solid factors, namely those that cannot be extended into a longer solid factor [35]. We make use of the following lemma to bound the size of the trees:

Lemma 108 ([35])**.** *The extended solid factor trees have $\mathcal{O}(nz)$ nodes.*

We next describe our algorithm (see Figure 4.4 for pseudocode).

Construction. We start by constructing the minimizer versions of the extended solid factor trees – that is for the solid factors trimmed to their parts starting (resp. ending) at the position of their minimizer. In particular, we show how to construct the forward extended solid factor tree (see Fig. 4.5) – the backward one can be constructed by simply doing the same operations on the reversed string, except that the minimizers will be computed on the reversed substrings.

Initialization. We construct the tree with a DFS traversal of the full (non-minimizer) extended solid factor tree, starting from the root, which corresponds to ε, the empty string. Each node corresponds to the right extension of a solid factor U of X starting at a position i (recall that U can be empty, in which case its right extension is $H_X[i\,..\,n]$). We assume that H_X and PP_H are computed before running the main algorithm and can be accessed just like X and $n=|X|$.

First Visit to a Node. When a node u that corresponds to a solid factor U starting at position i of X and ending at position j is created, we keep a pair of labels (i,

Algorithm 3 Construct-$\mathcal{T}$

1: **Global variables:** $j = n$, $p = 1.0$, string $S = \varepsilon$, set Diff $= \emptyset$, set Minimizers $= \emptyset$.
2: create a node *root*
3: run Augment-$\mathcal{T}(n+1, root)$
4: **return** *root* ▷ The minimizer extended solid factor tree

Algorithm 4 DOWN(i, u, α)

1: $S \leftarrow \alpha S$
2: **if** $\alpha \neq H_X[i]$ **then** add (i, α) to Diff
3: **end if**
4: **if** $|S| \geq \ell$ and $p \cdot PP_H[i-1+\ell]/PP_H[j] \geq \frac{1}{z}$ **then** ▷ $S[1..\ell]$ is solid
5: Minimizers $\leftarrow$ Minimizers $\cup$ $\{i + f(S[1..\ell]) - 1\}$
6: **end if**
7: add a node v as a child of u
8: run Augment-$\mathcal{T}(i, v)$

Algorithm 5 Augment-$\mathcal{T}(i, u)$

1: **for** $\alpha \in \Sigma$ **do**
2: **if** $p = 1$ and $\alpha = H_X[i-1]$ **then** ▷ If U is empty
3: $j \leftarrow j - 1$
4: run DOWN$(i-1, u, \alpha)$
5: $j \leftarrow j + 1$
6: **else if** $p \cdot p_{i-1}[\alpha] \geq \frac{1}{z}$ **then**
7: $p \leftarrow p \cdot p_{i-1}[\alpha]$
8: run DOWN$(i-1, u, \alpha)$
9: **end if**
10: **end for**
11: **if** $i < n+1$ **then** run UP(i, u)
12: **end if**

Algorithm 6 UP(i, u):

1: **if** $i \in$ Minimizers **then**
2: remove i from Minimizers
3: set label of u to (i, Diff)
4: **else if** u has at most one child **then** merge u with PARENT(u)
5: **end if**
6: $p \leftarrow \min(1, p \cdot p_i[S[1]]^{-1})$
7: **if** $(i, S[1]) \in$ Diff **then** remove $(i, S[1])$ from Diff
8: **end if**
9: remove the first letter from S

Figure 4.4: The space-efficient algorithm for constructing the minimizer extended solid factor tree of a weighted string X.

Diff), where Diff is the sequence (list) of mismatches between U and $H_X[i..j]$. By Lemma 100, the label of a given node has size $\mathcal{O}(\log z)$. Note, also, that for any ancestor of a node u its list of mismatches will be a suffix of Diff.

A single node and equivalently such a pair of labels can still represent multiple solid factors (for different values of j – if the suffix of the solid factor matches the heavy string); henceforth, by U we mean the shortest such solid factor: j is the largest element of Diff or $j = i - 1$ if Diff is empty.

Additionally, given a node u representing $S = U \cdot H_X[j+1..n]$, we check if the longest solid prefix of S has length at least ℓ; we check this in $\mathcal{O}(1)$ time using value p – a global variable that denotes the probability of the current node – that is the weight of U and the precomputed array PP_H of prefix products of H_X for the heavy part. If this is the case, we ask for the minimizer μ of this solid factor, and mark the $(\mu-1)$th ancestor of u as a minimizer node. Such minimizer can be found in $\mathcal{O}(1)$ time using a heap data structure [71], which stores information about the length-k substrings of the length-ℓ prefix of $S = U \cdot H_X[j+1..n]$ and is updated in $\mathcal{O}(\log \ell)$ time in each step of the traversal.

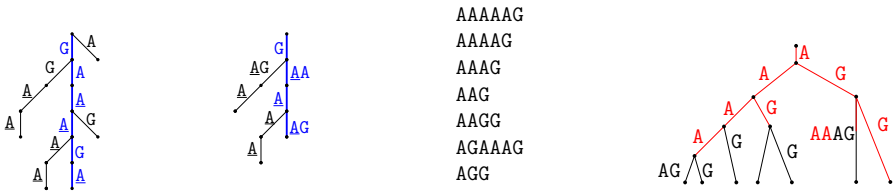


Figure 4.5: (a) The forward extended solid factor tree with X from Example 91. The blue edges correspond to the heavy string $H_X = \mathrm{AGAAAG}$ (reversed). The minimizer positions are underlined. (b) The minimizer extended solid factor tree. The edges without any minimizer descendant nodes are pruned and the non-minimizer nodes are made implicit. (c) The lexicographically sorted strings corresponding to each path from a minimizer node to a root. (d) The minimizer solid factor tree constructed by Theorem 110. It contains the forward (top) tree from Figure 4.3 as a red subtree. The edge with no label is added in the figure to stress that AAG also has a corresponding leaf. In the algorithm, we simply make the corresponding internal node explicit and treat it as a leaf node.

Stepping Down to a Child Node. If U is empty, then the node u corresponds to a string $H_X[i\,..\,n]$. In this case, p is not updated when creating its child v corresponding to $H_X[i-1\,..\,n]$. This way, we ensure by induction that $p = 1$ at the creation of each such node, and only for such a node, so that this can be checked in $\mathcal{O}(1)$ time. We now assume that the created child does not correspond to $H_X[i-1\,..\,n]$. To create a child v of the node u, corresponding to the right extension of the string $\alpha \cdot U$ for some letter $\alpha \in \Sigma$, one needs to check if $\alpha \cdot U$ is valid by computing its probability (U is nonempty, or the letter α is different from $H_X[i-1]$). This is done using p, which we multiply by $p_{i-1}[\alpha]$. In any case, the labels of v are computed from the labels of u by decreasing the starting position and potentially adding a new label to Diff.

Returning to the Parent Node. In the DFS we traverse the full extended solid factor tree, but we are only interested in the strings that start in those minimizer nodes. Thus, we remove the nodes which correspond to letters of the solid factors that appear before the position of the first minimizer and recursively compactify the tree during the traversal.

After all descendants of u are created, we keep u explicit if it is a minimizer node or if it has more than one (not removed) child. Otherwise, the node u is made implicit by merging it with its parent. Finally, upon returning to the parent of u we update p by dividing it by $p_i[U[1]]$ (if $p < 1$) and the list of differences by removing position i if $i \in$ Diff.

Complexity Analysis. By Lemma 103 the final tree has $\mathcal{O}(\frac{nz}{\ell})$ nodes in expectation. As for the construction space, observe that while a node u is being processed, only the path between u and the root is uncompacted, and contains at most n nodes. All the other global variables also have size $\mathcal{O}(n)$, therefore the total expected work space needed is $\mathcal{O}(\frac{nz}{\ell}\log z + n)$. As for the construction time, note that the set of created nodes is exactly the set of nodes from the original extended solid factor tree (namely, without minimizers), which has size $\mathcal{O}(nz)$ by Lemma 108. Reading X and creating H_X costs $n\sigma = \mathcal{O}(n)$ time [54]; after that the total number of probabilities read is bounded by the number of tree nodes as for a single position we can read

those in the order of non-increasing probabilities.

During the construction, all the operations cost $\mathcal{O}(1)$ time with the exception of updating the minimizer heap which takes $\mathcal{O}(\log \ell)$ time and storing a copy of the list of differences for each minimizer node in $\mathcal{O}(\log z)$ time. Note that the last type of the operation does not influence the worst-case running time as we can abandon the computation upon learning that the total size of those lists reaches nz – in which case the classic (non-minimizer) data structure is more efficient. We have thus proved the following lemma.

Lemma 109. *For any weighted string of length n, any weight threshold $\frac{1}{z}$, and any integer $\ell > 0$, we can construct a representation of the minimizer extended solid factor trees in $\mathcal{O}(nz \log \ell)$ time using $\mathcal{O}(n + \frac{nz \log z}{\ell})$ expected space.*

Main Result. The minimizer solid factors tree can be constructed from the minimizer extended solid factor tree in $\mathcal{O}(\frac{nz}{\ell} \log \frac{nz}{\ell} \log z)$ expected time and $\mathcal{O}(\frac{nz}{\ell} \log z)$ expected space (as proven below), hence we arrive at the main result of the section:

Theorem 110. For any weighted string X of length n, any weight threshold $\frac{1}{z}$, and any integer $\ell > 0$, we can construct the minimizer solid factor tree in expected time $\mathcal{O}(nz \log \ell + \frac{nz}{\ell} \log \frac{nz}{\ell} \log z)$ and space $\mathcal{O}(n + \frac{nz}{\ell} \log z)$.

Proof. As stated above it remains to show how to construct the minimizer solid factor tree from the minimizer extended solid factor tree. We achieve that by reversing the tree, that is, by creating the trie of all the strings from the minimizer extended solid factor tree read from leaf to root (corresponding to strings $U \cdot H_X[j+1..n]$). Note that for two such strings we can find their longest common prefix (LCP), and hence also compare them in $\mathcal{O}(\log z)$ time with a use of an LCP data structure for H_X [150] (comparison of $\mathcal{O}(\log z)$ intervals of H_X and $\mathcal{O}(\log z)$ differences).

We first sort those strings in lexicographic order. Since there are in expectation $\mathcal{O}(\frac{nz}{\ell})$ of them, and a single comparison takes $\mathcal{O}(\log z)$ time, this takes $\mathcal{O}(\frac{nz}{\ell} \log \frac{nz}{\ell} \log z)$ time in total using any optimal comparison-based sorting algorithm [71]. Now we construct the compacted trie of those strings node by node in the order of a DFS. Each edge will be labeled with an interval of H_X and a list of at most $\log_2 z$ differences.

We start from creating a single edge from root to a leaf representing the first string. Now we iterate over all remaining strings in lexicographic order – we first compute the length of the LCP of this string, and the previous one, next starting from the leaf representing the previous string we move up the tree node by node to find its ancestor at depth equal to the length of this LCP. If this node turns out to be an implicit one, then we make it explicit by dividing the edge (and hence also the interval of H_X and the list of differences). We finish by creating a new child of the reached node – this child becomes the leaf representing the new string.

Unlike the construction from [35] we do not need to trim the H_X parts after constructing the tree, as in our query algorithm we must verify the weight for each match anyway.

□

4.5 PRACTICALLY FAST QUERYING WITHOUT A GRID

In this section, we describe a simple and fast querying algorithm that does not require the grid to be constructed on top of the trees. This querying algorithm has worse guarantee than that in Theorem 106 but performs much better in practice, due to its simplicity, as we show in the experimental evaluation.

Like in the previous construction let $\mu = f(P[1..\ell])$. Without loss of generality we assume that $\mu \leq \frac{m}{2}$ (otherwise we swap the roles of the parts of P and of trees $\mathcal{T}_{\text{suff}}$ and $\mathcal{T}_{\text{pref}}$). Let u be the node reached by reading $P[\mu..m]$ in $\mathcal{T}_{\text{suff}}$. We can separately check each leaf in the subtree of u as a potential candidate in $\mathcal{O}(m)$ time: we can do this assuming we have random access to X. This time we cannot use the $\frac{nz}{\sigma^m}$ bound on the expected number of candidates from Lemma 105. However, $P[\mu..m]$ has length at least $m/2$, and hence the expected number of candidates can still be bounded by $\sum_{k=\lceil m/2 \rceil}^{m} \frac{nz}{\sigma^k} \leq 2\frac{nz}{\sigma^{m/2}}$ using a similar argument. Thus we can answer a query in $\mathcal{O}(m \cdot (1 + \frac{nz}{\sigma^{m/2}}))$ expected time.

Example 111. For the weighted string from Example 91 and pattern $P' = \mathrm{B}\underline{\mathrm{A}}\mathrm{AB}$ the grid based construction finds a single candidate $(3,3)$ to check (see Figure 4.3 and Example 107). In case of the simpler querying algorithm we only locate one part of the pattern in one of the trees, in this case $|\mathrm{G}\underline{\mathrm{A}}| < |\underline{\mathrm{A}}\mathrm{AB}|$, hence we locate AAB in $\mathcal{T}_{\text{suff}}$. We find two leaves with labels $(3,3)$ and $(4,2)$, that is candidate occurrences starting at positions $2 = 3 - |\mathrm{G}\underline{\mathrm{A}}| + 1$ and $3 = 4 - |\mathrm{G}\underline{\mathrm{A}}| + 1$. Each of those can be checked naively to get the occurrence probabilities $3/20$ and $3/40$ respectively; each probability is smaller than $1/4$.

4.6 RELATED WORK

The *Weighted Indexing* problem was introduced by the work of Iliopoulos et al. [123]; the authors gave a tree-based index supporting $\mathcal{O}(m + |\mathrm{Occ}|)$-time queries. The construction time, space, and size of the index was, however, $\mathcal{O}(n\sigma^z \log z)$. Their index is essentially a compacted trie of all the solid factors of X. Amir et al. [18] then reduced the Weighted Indexing problem to the so-called *Property Indexing* problem in a standard string of length $\mathcal{O}(nz^2 \log z)$. For the latter problem, Amir et al. proposed a super-linear-time construction and $\mathcal{O}(m + |\mathrm{Occ}|)$-time queries. Later, it was shown that Property Indexing can be performed in linear time [34]. This directly gives a solution to the Weighted Indexing problem with construction time, space, and size $\mathcal{O}(nz^2 \log z)$, preserving the optimal query time. The state-of-the-art indexes for Weighted Indexing are the weighted suffix tree (WST) [34, 35] and the weighted suffix array (WSA) [53]; see the Introduction for more details. To conclude, *there are currently no practical indexing schemes for Weighted Indexing mainly due to their prohibitive size and space requirements*.

There is substantial work on practical indexing schemes for probabilistic/uncertain data; e.g., for range queries [10, 58, 61, 200, 201], top-k queries [120, 179, 217, 218], nearest neighbor queries [9, 59, 60], sql-like queries [178], inference queries [132], and probabilistic equality threshold queries [193]. These indexes were developed for different uncertainty data models, such as tuple uncertainty and attribute uncertainty [194]. Under tuple uncertainty, the presence of a tuple in a relation is

probabilistic, while under attribute uncertainty a tuple is certainly present in a database but one or more of its attributes are not known with certainty. Several indexes are built on R-trees or inverted indexes (e.g., [75, 193]), while others are built on R*-trees [200]. There are also specialized indexes, e.g., for probabilistic XML queries [107] or uncertain graphs [198, 216]. Our work differs substantially from the above in the supported data model and query type.

4.7 EXPERIMENTAL EVALUATION

4.7.1 DATA AND SETUP

Data. We used three real weighted strings which model variations found in the DNA ($\sigma = 4$) of different samples of the same species. The chromosomal or genomic location of a gene or any other genetic element is called a *locus* and alternative DNA sequences at a locus are called *alleles*. Allele frequency, or gene frequency, is the relative frequency of an allele at a particular locus in a population, expressed as a fraction or percentage. Thus, alleles have a natural representation as weighted strings: we model the probability $p_i(\alpha)$ in these strings as the relative frequency of letter α at position i among the different samples.

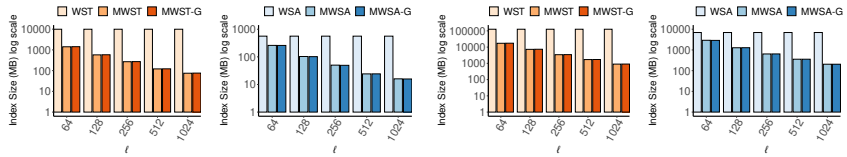
We next describe the datasets we used (see also Table 4.1):

- SARS: The full genome of *SARS-CoV-2* (isolate Wuhan-Hu-1) [1] combined with a set of single nucleotide polymorphisms (SNPs) [2] taken from $1,181$ samples [205].
- EFM: The full chromosome of *Enterococcus faecium* Aus0004 strain (CP003351) [3] combined with a set of SNPs [4] taken from $1,432$ samples [69].
- HUMAN: The full chromosome 22 of the *Homo sapiens* genome (v. GRCh37) [5] combined with a set of SNPs [6] taken from the final phase of the 1000 Genomes Project (phase 3) representing $2,504$ samples [90].

The percentage of positions where more than one letter has a probability of occurrence larger than 0 is denoted by Δ.

We also used another real weighted string comprised of Received Signal Strength Indicator (RSSI) (i.e., signal strength) values of sensors; $p_i(\alpha)$, $i \in [1..n]$, corresponds to the ratio of IEEE 802.15.4 channels that received an RSSI value α at time i, and $\sigma = 91$ [171]. Its characteristics are in Table 4.1. Furthermore, we used a family of weighted strings, generated from RSSI. Each such string is denoted by $\text{RSSI}_{n,\sigma}$, where $n \in \{2,4,6,8\}$ is the number of times the length of this string is larger than that of RSSI and $\sigma \in \{16,32,64\}$ is its alphabet size. To increase n, we appended RSSI to itself. To reduce σ, we replaced each RSSI value (integer) v with $v \mod y$, where $y \in \{16,32,64\}$ is the desired alphabet size. For all these datasets, $\Delta = 100\%$.

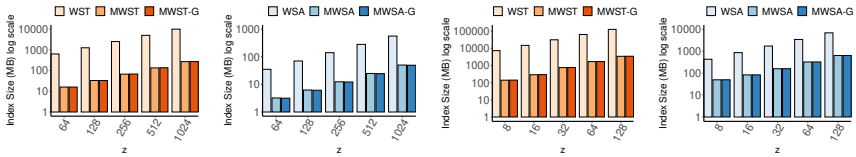
Parameters. For every weighted string of length n, every pattern length $m \in \{64,128,256,512,1024\}$, and every z we used, we selected $\lfloor nz/200 \rfloor$ patterns from the z-estimation of the weighted string, uniformly at random, to account for the different n and z values. For example, for HUMAN whose length is $n = 35,194,566$, and for $z = 32$, we have selected $5,631,130$ patterns uniformly at random from its



(a) SARS (b) SARS (c) EFM (d) EFM

(e) HUMAN (f) HUMAN

Figure 4.6: Index size (log scale, MB) vs. ℓ.



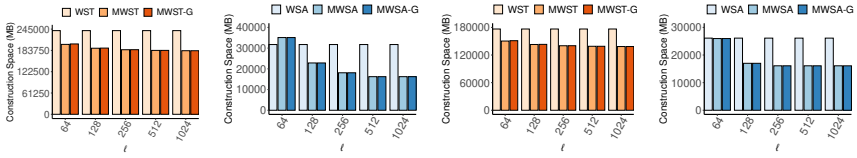
(a) SARS (b) SARS (c) EFM (d) EFM

(e) HUMAN (f) HUMAN

Figure 4.7: Index size (log scale, MB) vs. z. The tree-based indexes for HUMAN (Fig. 4.7e) needed $>$ 252GB of space when $z \geq 16$ and hence could not be constructed.

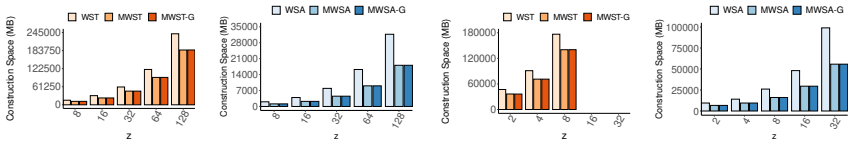
Table 4.1: Characteristics of the real datasets we used.

Dataset	# of samples	Length n	Δ as percentage of n	Size of z-estimation for the default z (MBs)
SARS	1,181	29,903	3.6%	31
EFM	1,432	2,955,294	6%	378
HUMAN	2,504	35,194,566	3.2%	282
RSSI	N/A	6,053,462	100%	96.9



(a) EFM (b) EFM (c) HUMAN (d) HUMAN

Figure 4.8: Construction space (MB) vs. ℓ.



(a) EFM (b) EFM (c) HUMAN (d) HUMAN

Figure 4.9: Construction space (MB) vs. z. The tree-based indexes for HUMAN (Fig. 4.9c) needed $>$ 252GB when $z \geq 16$ and hence could not be constructed.

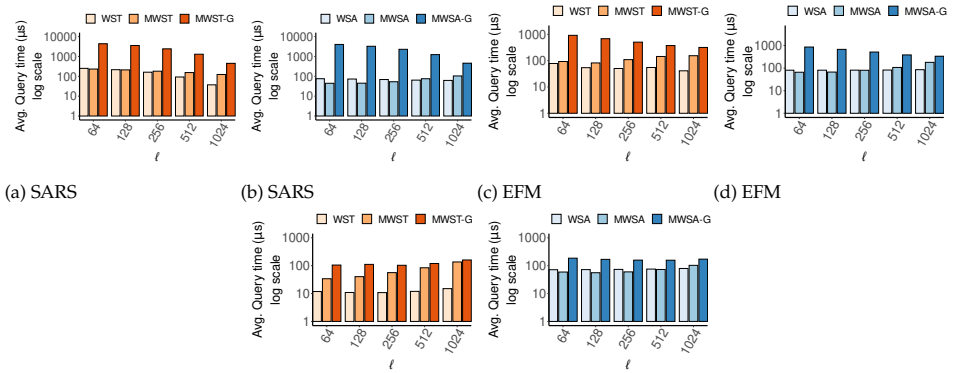
32-estimation. The default z for SARS, EFM, HUMAN, RSSI, and RSSI$_{n,\sigma}$ was 1024, 128, 8, 16, and 16, and led to z-estimations with sizes of several MBs; see Table 4.1. The parameter ℓ was set to m, and the default m value was 256.

Implementations. We used the implementations of the state-of-the-art indexes WST and WSA from [34] and [53], respectively. We implemented: (1) MWST-G, the algorithm underlying our Theorem 106. (2) MWST, a simplified version of MWST-G that drops the 2D grid and performs pattern matching as described in Section 4.5. (3) MWSA and MWSA-G, the *array-based* versions of MWST and MWST-G, respectively. Note that an in-order DFS traversal of the tree gives the array. (4) MWST-SE, the space-efficient construction of MWST underlying Theorem 110. In all implementations, we used Karp-Rabin fingerprints [134] to compute the minimizers.

Measures. We used all four relevant measures of efficiency (see Introduction): index size; query time; construction space; and construction time. To measure the query and construction time, we used the chrono C++ library. To measure the index size, we used the malloc2 C++ function. To measure the construction space, we recorded the maximum resident set size using the /usr/bin/time -v command.

Environment. All experiments ran using a single AMD EPYC 7282 CPU at 2.8GHz with 252GB RAM under GNU/Linux. All methods were implemented in C++ and compiled with g++ (v. 12.2.1) at optimization level -O3.

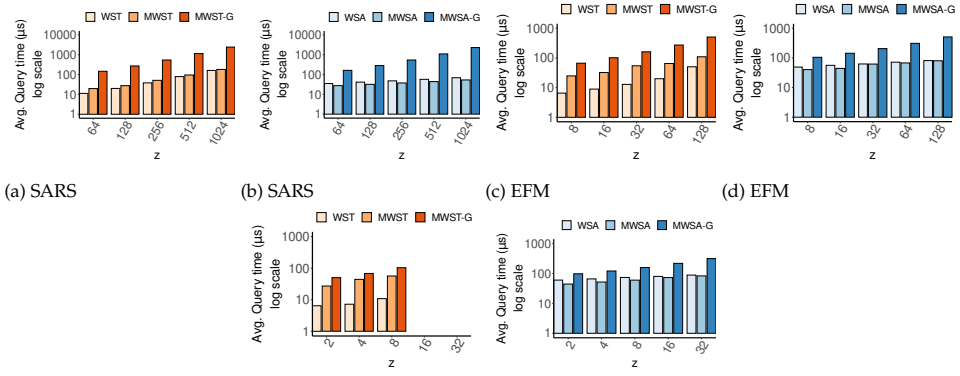
Code and Datasets. The code and all datasets are available at https://github.com/



(a) SARS (b) SARS (c) EFM (d) EFM

(e) HUMAN (f) HUMAN

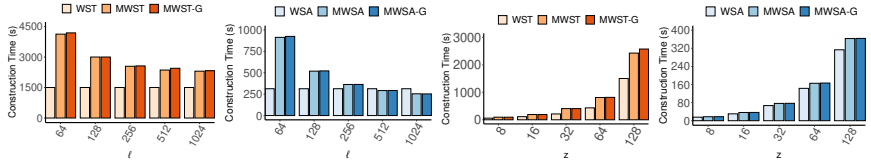
Figure 4.10: Average query time (log scale, μs) vs. ℓ.



(a) SARS (b) SARS (c) EFM (d) EFM

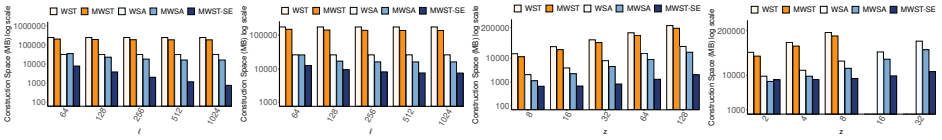
(e) HUMAN (f) HUMAN

Figure 4.11: Average query time (log scale, μs) vs. z. The tree-based indexes for HUMAN (Fig. 4.11e) needed $>$ 252GB when $z \geq 16$ and hence could not be constructed.



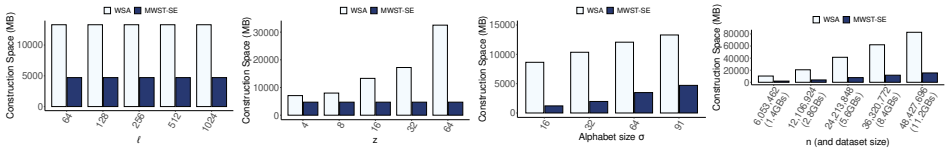
(a) EFM (b) EFM (c) EFM (d) EFM

Figure 4.12: (a, b) Construction time (s) vs. ℓ for EFM. (c, d) Construction time (s) vs. z for EFM. The results for SARS and HUMAN were analogous.



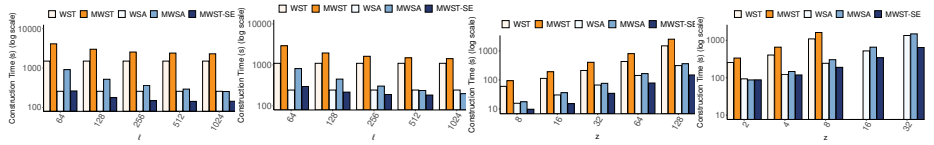
(a) EFM (b) HUMAN (c) EFM (d) HUMAN

Figure 4.13: Construction space (log scale, MB) vs: (a, b) ℓ. (c, d) z. WST and MWST for HUMAN (Fig. 4.13d) needed $>$ 252GB when $z \geq 16$ and hence could not be constructed. The results for SARS were analogous.



(a) RSSI (b) RSSI (c) $\text{RSSI}_{1,16},\ldots,\text{RSSI}$ (d) $\text{RSSI}_{1,32},\ldots,\text{RSSI}_{8,32}$

Figure 4.14: Construction space (MBs) vs. (a) ℓ, (b) z, (c) σ, and (d) n (and dataset size).



(a) EFM (b) HUMAN (c) EFM (d) HUMAN

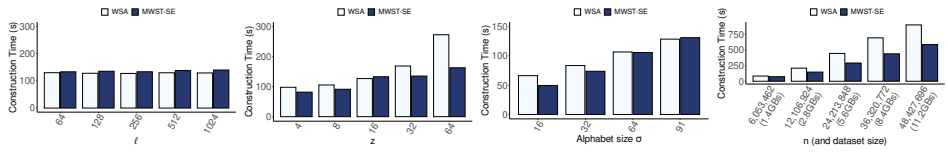
Figure 4.15: Construction time (log scale, s) vs: (a, b) ℓ. (c, d) z. WST and MWST for HUMAN (Fig. 4.15d) needed $>$ 252GB when $z \geq 16$ and hence could not be constructed. The results for SARS were analogous.

solonas13/ius under GNU GPL v3.0.

4.7.2 EVALUATING OUR MINIMIZER-BASED INDEXES

Index Size. Figs. 4.6 and 4.7 show that our tree-based (resp. array-based) indexes occupy *up to two orders of magnitude less space* than WST (resp. WSA). The size of our indexes decreases with ℓ and increases with z (see Theorem 106). Furthermore, the array-based indexes occupy several times less space than the tree-based ones, as it is widely known [133]. For example, note from Figs. 4.6c and 4.6d that for $\ell = 1024$, WST occupied 126GB of space, whereas our MWST 900MB and MWSA only 204MB! As expected, our grid-based indexes MWST-G and MWSA-G occupy a slight amount of extra space compared to MWST and MWSA, respectively.

Construction Space. Figs. 4.8 and 4.9 show that our tree-based (resp. array-based) indexes outperform WST (resp. WSA) *by* 27% *(resp.* 61%*)* on average; the results for SARS are analogous to those for EFM. Although our construction algorithm (see Theorem 106) takes $\Theta(nz)$ space in any case, it carries lower constant factors than that of WST. That is, in practice, the index construction space for our tree-based indexes decreases as ℓ increases and increases with z – see Lemmas 103 and 7, which show a clear dependency on the number $\mathcal{O}(\frac{nz}{\ell})$ of sampled minimizers. The same explanation holds for WSA and our array-based indexes. Again, as it is widely



(a) RSSI (b) RSSI (c) $\mathrm{RSSI}_{1,16},\ldots,\mathrm{RSSI}$ (d) $\mathrm{RSSI}_{1,32},\ldots,\mathrm{RSSI}_{8,32}$

Figure 4.16: Construction time (s) vs. (a) ℓ, (b) z, (c) σ, and (d) n (and dataset size).

known [133], the array-based indexes outperform the tree-based ones in terms of space; and, as expected, MWST-G and MWSA-G need a very slightly larger construction space than MWST and MWSA, respectively.

Query Time. Figs. 4.10 and 4.11 show that MWST is generally slower than WST because its search operation is more costly than that of WST (see Theorem 106). However, MWSA is competitive to WSA since, due to the smaller size of the former, the binary search operation used in query answering (pattern matching) [163] becomes faster. This is *very encouraging* given its substantially smaller index size and index construction space across all z and ℓ values. Furthermore, MWST and MWSA outperform MWST-G and MWSA-G, respectively. This is in line with the findings of [28, 159], which show that simple verification schemes like the one developed by us in Section 4.5, are faster than grid approaches, even if the theoretical guarantees provided by the former are weaker. Note in Fig. 4.10 that the query time of the grid-based indexes is not negatively affected by increasing ℓ, unlike MWST and MWSA. In particular, Fig. 4.10 shows that, although the grid-based indexes are slower, the difference in performance *decreases* as ℓ grows. This is because, as ℓ grows, the grid becomes smaller and the simple verification schemes become more expensive, which highlights the benefit of Theorem 106. The query time of all indexes increases with z, as expected by their time complexities. The query time of WST and WSA does not depend on ℓ, as expected by their time complexities.

Construction Time. Fig. 4.12 shows that WST and WSA can be constructed in less time than our tree-based and array-based indexes, respectively. This is expected as our construction is much more complex than that of WST and WSA [34, 53]. In particular, although our construction algorithm (see Theorem 106) takes $\Theta(nz)$ time in any case, it carries higher constant factors than those of WST and WSA. This is expected as, in some sense, our construction largely follows the one of WST and WSA but it does additional work implied by the sampling mechanism. In practice, the construction time decreases as ℓ increases and increases with z – see Lemmas 103 and 7, which show a clear dependency on the number $\mathcal{O}(\frac{nz}{\ell})$ of sampled minimizers. On average, MWST requires 70% (resp. MWSA requires 32%) more time to be constructed than WST (resp. WSA). MWST-G and MWSA-G have similar construction time to MWST and MWSA, respectively.

4.7.3 Evaluating our space-efficient index construction

Construction Space. Fig. 4.13 shows that the construction space of MWST-SE is up to *one order of magnitude smaller* than that of WSA and 52 *times smaller* than that of MWST

on average. The construction space of MWST-SE decreases with ℓ and increases with z, as expected by Theorem 110. For example, in Fig. 4.13a MWST-SE needs only 772MB of memory to be constructed when $\ell = 1024$, while WSA and MWST need over 32GBs and 183GBs, respectively. Even for $\ell = 64$, the construction space of MWST-SE is 4 times smaller than that of WSA and more than 25 times smaller than that of MWST. On the RSSI datasets, MWST-SE substantially outperformed all other indexes. Fig. 4.14 shows the results against the best competitor, WSA. MWST-SE needed 4 times less space than WSA on average. Although z and ℓ appear in the space bound of Theorem 110, and σ does not, the impact of the latter is larger on the construction space. The reason is that the maximum resident set size we measure includes the memory for reading the dataset file, which increases with σ for all indexes. Note that the construction space for both indexes scaled linearly with n, as can be seen in Fig. 4.14d.

Construction Time. Fig. 4.15 shows that the construction time of MWST-SE is on average 53% smaller than that of WSA, the next fastest index. This is *very encouraging*, as MWST-SE is quite complex. The construction time of MWST-SE decreases with ℓ and increases with z, as expected by Theorem 110. For example, for $\ell = 1024$ and $z = 128$ in Fig. 4.15a, the construction time of MWST-SE is smaller by 31% (resp. 16 times smaller) compared to that of WSA (resp. MWST). This faster construction is a consequence of WSA and WST being always of $\Theta(nz)$ size (producing copies of solid factors), while in the extended solid factor trees each solid factor is considered only once. The extra $\mathcal{O}(\log \ell)$ cost for heap operations is very optimized and in practice comparable with the large constants of the other constructions for reasonable ℓ values.

On the RSSI datasets, MWST-SE substantially outperformed all other indexes. Fig. 4.16 shows the results against the best competitor, WSA. Now the term $n\sigma = \mathcal{O}(n)$ prevails in the construction time bound of Theorem 110. Thus, the construction time increased linearly with σ (Fig. 4.16c), while the impact of ℓ and z was smaller. MWST-SE scaled linearly with n (Fig. 4.16d). The input datasets we used are in the order of GBs, thus their reading takes much of the construction time.

4.7.4 Conclusion of our experimental evaluation

The most practical solution to ℓ-Weighted Indexing is to use the MWST-SE algorithm, which requires the smallest construction space and time (see Figs. 4.13, 4.14, 4.15, and 4.16) to construct MWST and then infer MWSA, the array-based version of MWST via a standard in-order DFS traversal on MWST [163], as MWSA has the smallest index size and a competitive query time to WSA (see Figs. 4.6, 4.7, 4.10, and 4.11).

4.8 Conclusion and future work

The size of our indexes is $\mathcal{O}(\frac{nz}{\ell}\log z)$ *in expectation* because minimizers have no worst-case guarantees; e.g., in string $\mathrm{abcdefg}\ldots$, every position is a minimizer. This could be avoided by using *difference covers* [133] instead. Difference covers are practical to construct, but slash the set of positions by $\sqrt{\ell}$ instead of ℓ (the factor that minimizers

slash by in real-world datasets). As our focus is on practical schemes for reducing the space, we have resorted to minimizers. There are other sampling schemes slashing the set of positions by ℓ in the worst case [137], such as *string synchronizing sets*; these are however impractical to construct (see [79], for example) and not directly applicable to the uncertain setting. Recently, *partitioning sets* [145] were used to give worst case guarantees on indexes for standard texts [29]. It remains to be investigated whether this could be adapted to the uncertain case. This result would unlikely be relevant for practical applications, as authors claim that partitioning sets are already unpractical for standard texts. However, it would be interesting on a theoretical aspect.

Our methods do not explicitly take advantage of the n probability distributions. The following may be explored to improve our indexes: (i) use the percentage Δ of positions where more than one letter has a probability of occurrence larger than 0 as a complexity parameter; (ii) use lower or upper bounds on the letter probabilities depending on the underlying application. For instance, in sequencing data, a quality score is assigned to a single nucleotide in every position i, which is then translated to a probability p. The other 3 nucleotides can be assigned to probability $(1-p)/3$. However, based on the vicinity of the ith position or on domain knowledge, we could rather use upper or lower bounds for the remaining letters.

Our indexes rely explicitly or implicitly on the z-estimation of the input weighted string. This is a family $\mathcal{S}$ of $\lfloor z \rfloor$ strings of length n. Although the construction of $\mathcal{S}$ is combinatorially correct [34] it is oblivious to domain knowledge. It may thus generate solid factors that are completely impossible to occur in a real-world dataset. For instance, in biology, such implausible factors are termed *absent* or *avoided* [14]. If one had this knowledge, they could easily trim these factors from the index by first querying them during a post-processing step. It seems much more challenging though to amend our data model to include this knowledge while constructing the index.

5

Conclusion

Pattern matching, *comparison*, and *indexing* are three core tasks in sequence analysis. This thesis investigates generalizations of these tasks for data structures that model collections of related sequences—such as *pangenomes*—rather than individual strings. In addition to the general summary in introduction, and to chapter-specific open problems that can be found in the end of each chapter, we conclude with some more general future research directions.

Beyond these three tasks, sequence analysis often involves identifying *regularities* or *structural properties* in texts, such as periods, symmetries, repetitions, or palindromes [158]. Those structural properties often have crucial applications, for example to pattern matching [112]. While structural properties have been studied for degenerate strings (e.g., [17, 23, 77, 97, 125]), their generalization to more general variable strings are still, for a large part, open. Though recent work has begun exploring their extension to more general variable strings [148], this direction is for a large part open.

The work of classification started in Section 2.1 could be extended to many different problems presented in this thesis, for example approximate pattern matching and comparison. In particular, the generalization of our works on approximate pattern matching and comparison to graph-based representations could be investigated.

Indexing variable texts is also a crucial task. Despite a first discouraging result for ED strings [105], one could hope to design indexes for more restricted variable strings, or that, despite theoretical limitations, have good practical value. A recent work in that direction generalized the *Burrows-Wheeler transform*, a widely-used practical compression scheme, to ED strings [62].

Finally, a major open question involves the *construction* of ED strings. Given a set of sequences, formalizing criteria for a "good" ED string representation—and developing algorithms to compute it—remains unresolved. Current methods rely on simple heuristics, leaving room for theoretical or practical improvement.

Bibliography

URLs in this thesis have been archived on Archive.org. Their link target in digital editions refers to this timestamped version.

References

[1] https://www.ncbi.nlm.nih.gov/nuccore/MN908947.3.

[2] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8363274/bin/elife-66857-supp2.txt.

[3] https://www.ncbi.nlm.nih.gov/nuccore/CP003351.

[4] https://github.com/francesccoll/powerbacgwas/blob/main/data/efm_clade_all.vcf.gz.

[5] https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.13/.

[6] https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL.chr22.phase3_shapeit2_mvncall_integrated_v5b.20130502.genotypes.vcf.gz.

[7] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014.

[8] Karl R. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[9] Pankaj K. Agarwal, Boris Aronov, Sariel Har-Peled, Jeff M. Phillips, Ke Yi, and Wuzhou Zhang. Nearest-neighbor searching under uncertainty II. *ACM Transactions on Algorithms*, 13(1):3:1–3:25, 2016.

[10] Pankaj K. Agarwal, Siu-Wing Cheng, Yufei Tao, and Ke Yi. Indexing uncertain data. In Jan Paredaens and Jianwen Su, editors, *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, pages 137–146. ACM, 2009.

[11] Charu C. Aggarwal. On unifying privacy and uncertain data models. In *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, pages 386–395. IEEE Computer Society, 2008.

[12] Charu C. Aggarwal. *Managing and Mining Uncertain Data*, volume 35 of *Advances in Database Systems*. Kluwer, 2009.

[13] Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi. Subset wavelet trees. In *21st International Symposium on Experimental Algorithms (SEA)*, volume 265 of *LIPIcs*, pages 4:1–4:14, 2023.

[14] Yannis Almirantis, Panagiotis Charalampopoulos, Jia Gao, Costas S. Iliopoulos, Manal Mohamed, Solon P. Pissis, and Dimitris Polychronopoulos. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology*, 12(1):5:1–5:12, 2017.

[15] Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Degenerate string comparison and applications. In Laxmi Parida and Esko Ukkonen, editors, *18th International Workshop on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPIcs*, pages 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[16] Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundamenta Informaticae*, 175(1-4):41–58, 2020.

[17] Mai Alzamel, Christopher Hampson, Costas S. Iliopoulos, Zara Lim, Solon Pissis, Dimitrios Vlachakis, and Steven Watts. Maximal degenerate palindromes with gaps and mismatches. *Theoretical Computer Science*, 978:114182, 2023.

[18] Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. In Moshe Lewenstein and Gabriel Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 188–199. Springer, 2006.

[19] Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. *Theoretical Computer Science*, 395(2-3):298–310, 2008.

[20] Amihood Amir and Michael Itzhaki. Reconstructing General Matching Graphs. In *35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)*, volume 296 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:15, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[21] Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *Journal of Algorithms*, 37(2):309–325, 2000.

[22] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004.

[23] Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos, Inuka Jayasekera, and Gad M. Landau. Conservative string covering of indeterminate strings. In *Proceedings of the Prague Stringology Conference*, pages 108–115, 2008.

[24] Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster online elastic degenerate string matching. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *29th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 105 of *LIPIcs*, pages 9:1–9:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[25] Mozhdeh Ariannezhad, Ali Montazeralghaem, Hamed Zamani, and Azadeh Shakery. Improving retrieval performance for verbose queries via axiomatic analysis of term discrimination heuristic. In Noriko Kando, Tetsuya Sakai, Hideo Joho, Hang Li, Arjen P. de Vries, and Ryen W. White, editors, *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017*, pages 1201–1204. ACM, 2017.

[26] Rocco Ascone, Giulia Bernardini, Alessio Conte, Massimo Equi, Estéban Gabory, Roberto Grossi, and Nadia Pisanti. A Unifying Taxonomy of Pattern Matching in Degenerate Strings and Founder Graphs. In Solon P. Pissis and Wing-Kin Sung, editors, *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, volume 312 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:21, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[27] Lorraine A. K. Ayad, Carl Barton, and Solon P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognition Letters*, 88:81–87, 2017.

[28] Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text indexing for long patterns: Anchors are all you need. *Proceedings of the VLDB Endowment*, 16(9):2117–2131, 2023.

[29] Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text indexing for long patterns using locally consistent anchors, 2024.

[30] Jasmijn A. Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Natural Computing*, 21(1):81–108, 2022.

[31] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *CoRR*, abs/1412.0348, 2014.

[32] Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In Irit Dinur, editor, *57th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 457–466. IEEE Computer Society, 2016.

[33] Ricardo Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011.

[34] Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *Information and Computation*, 270, 2020.

[35] Carl Barton, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Efficient index for weighted sequences. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 4:1–4:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[36] Christian Baudet, Claire Lemaitre, Zanoni Dias, Christian Gautier, Eric Tannier, and Marie-France Sagot. Cassis: detection of genomic rearrangement breakpoints. *Bioinformatics*, 26(15):1897–1898, 2010.

[37] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[38] Michael Bendersky and W. Bruce Croft. Discovering key concepts in verbose queries. In Sung-Hyon Myaeng, Douglas W. Oard, Fabrizio Sebastiani, Tat-Seng Chua, and Mun-Kew Leong, editors, *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2008, Singapore, July 20-24, 2008*, pages 491–498. ACM, 2008.

[39] Giulia Bernardini, Alessio Conte, Estéban Gabory, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, Giulia Punzi, and Michelle Sweering. On Strings Having the Same Length- k Substrings. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, volume 223 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[40] Giulia Bernardini, Estéban Gabory, Solon P. Pissis, Leen Stougie, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string matching with 1 error. In Armando Castañeda and Francisco Rodríguez-Henríquez, editors, *15th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 13568 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2022.

[41] Giulia Bernardini, Estéban Gabory, Solon P. Pissis, Leen Stougie, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string matching with 1 error or mismatch. *Theory of Computing Systems*, Sep 2024.

[42] Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPIcs*, pages 21:1–21:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[43] Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM Journal on Computing*, 51(3):549–576, 2022.

[44] Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In *24th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 10508 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2017.

[45] Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Approximate pattern matching on elastic-degenerate text. *Theoretical Computer Science*, 812:109–122, 2020.

[46] Philip Bille, Inge Li Gørtz, and Tord Stordalen. Rank and select on degenerate strings. In *2024 Data Compression Conference (DCC)*, pages 283–292, 2024.

[47] Sudip Biswas, Manish Patil, Sharma V. Thankachan, and Rahul Shah. Probabilistic threshold indexing for uncertain strings. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, pages 401–412. OpenProceedings.org, 2016.

[48] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping, 2015.

[49] Thomas Büchler, Jannik Olbrich, and Enno Ohlebusch. Efficient short read mapping to a pangenome that is represented by a graph of ED strings. *Bioinformatics*, 39(5):btad320, 05 2023.

[50] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In Jianer Chen and Fedor V. Fomin, editors, *Parameterized and Exact Computation, 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10-11, 2009, Revised Selected Papers*, volume 5917 of *Lecture Notes in Computer Science*, pages 75–85. Springer, 2009.

[51] Vincenzo Carletti, Pasquale Foggia, Erik Garrison, Luca Greco, Pierluigi Ritrovato, and Mario Vento. Graph-based representations for supporting genome data analysis and visualization: Opportunities and challenges. In Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia, editors, *Graph-Based Representations in Pattern Recognition - 12th IAPR-TC-15 International Workshop,*

GbRPR 2019, Tours, France, June 19-21, 2019, Proceedings, volume 11510 of *Lecture Notes in Computer Science*, pages 237–246. Springer, 2019.

[52] Timothy M. Chan, Kasper Green Larsen, and Mihai Puatracscu. Orthogonal range searching on the RAM, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011.

[53] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Property suffix array with applications in indexing weighted sequences. *ACM J. Exp. Algorithmics*, 25:1–16, 2020.

[54] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. online weighted pattern matching. *Information and Computation*, 266:49–59, 2019.

[55] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 978–989. IEEE, 2020.

[56] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster pattern matching under edit distance : A reduction to dynamic puzzle matching and the seaweed monoid of permutation matrices. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 698–707. IEEE, 2022.

[57] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[58] Lu Chen, Yunjun Gao, Aoxiao Zhong, Christian S. Jensen, Gang Chen, and Baihua Zheng. Indexing metric uncertain data for range queries and range joins. *VLDB J.*, 26(4):585–610, 2017.

[59] Reynold Cheng, Lei Chen, Jinchuan Chen, and Xike Xie. Evaluating probability threshold k-nearest-neighbor queries over uncertain data. In Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 672–683. ACM, 2009.

[60] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Querying imprecise data in moving object environments. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1112–1127, 2004.

[61] Reynold Cheng, Yuni Xia, Sunil Prabhakar, Rahul Shah, and Jeffrey Scott Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the*

Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004, pages 876–887. Morgan Kaufmann, 2004.

[62] Lapo Cioni, Veronica Guerrini, and Giovanna Rosone. The burrows-wheeler transform of an elastic-degenerate string. In Ugo de'Liguoro, Matteo Palazzo, and Luca Roversi, editors, *Proceedings of the 25th Italian Conference on Theoretical Computer Science, Torino, Italy, September 11-13, 2024*, volume 3811 of *CEUR Workshop Proceedings*, pages 66–80. CEUR-WS.org, 2024.

[63] Aleksander Cislak, Szymon Grabowski, and Jan Holub. Sopang: online text searching over a pan-genome. *Bioinformatics*, 34(24):4290–4292, 2018.

[64] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004.

[65] Richard Cole and Ramesh Hariharan. Tree pattern matching and subset matching in randomized o(n $\log^3$m) time. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing (STOC)*, pages 66–75. ACM, 1997.

[66] Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31(6):1761–1782, 2002.

[67] Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 592–601. ACM, 2002.

[68] Richard Cole and Ramesh Hariharan. Tree pattern matching to subset matching in linear time. *SIAM Journal on Computing*, 32(4):1056–1066, 2003.

[69] Francesc Coll, Theodore Gouliouris, Sebastian Bruchmann, Jody Phelan, Kathy E. Raven, Taane G. Clark, Julian Parkhill, and Sharon J. Peacock. PowerBacGWAS: a computational pipeline to perform power calculations for bacterial genome-wide association studies. *Communications Biology*, 5(266), 2022.

[70] The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings Bioinformatics*, 19(1):118–135, 2018.

[71] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[72] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[73] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Covering problems for partial words and for indeterminate strings. *Theoretical Computer Science*, 698:25–39, 2017.

[74] Nicole B Crux and Shokrollah Elahi. Human leukocyte antigen (HLA) and immune regulation: How do classical and non-classical HLA alleles modulate immune response to human immunodeficiency virus and hepatitis C virus infections? *Frontiers in Immunology*, 8:832, July 2017.

[75] Xiangyuan Dai, Man Lung Yiu, Nikos Mamoulis, Yufei Tao, and Michail Vaitis. Probabilistic spatial queries on existentially uncertain data. In Claudia Bauzer Medeiros, Max J. Egenhofer, and Elisa Bertino, editors, *Advances in Spatial and Temporal Databases, 9th International Symposium, SSTD 2005, Angra dos Reis, Brazil, August 22-24, 2005, Proceedings*, volume 3633 of *Lecture Notes in Computer Science*, pages 400–417. Springer, 2005.

[76] Jacqueline W. Daykin, Richard Groult, Yannick Guesnet, Thierry Lecroq, Arnaud Lefebvre, Martine Léonard, Laurent Mouchard, Élise Prieur, and Bruce W. Watson. Efficient pattern matching in degenerate strings with the burrows-wheeler transform. *Information Processing Letters*, 147:82–87, 2019.

[77] Jacqueline W. Daykin and Bruce W. Watson. Indeterminate string factorizations and degenerate text transformations. *Mathematics in Computer Science*, 11(2):209–218, 2017.

[78] Robert B. Dial. Algorithm 360: shortest-path forest with topological ordering [H]. *Communcations of the ACM*, 12(11):632–633, 1969.

[79] Patrick Dinklage, Johannes Fischer, Alexander Herlez, Tomasz Kociumaka, and Florian Kurpicz. Practical performance of space efficient data structures for longest common extensions. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 39:1–39:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[80] N. Rex Dixon and Thomas B. Martin. *Automatic Speech and Speaker Recognition*. John Wiley & Sons, Inc., USA, 1979.

[81] Daniel Dorey-Robinson, Giuseppe Maccari, and John A. Hammond. Igmat: immunoglobulin sequence multi-species annotation tool for any species including those with incomplete antibody annotation or unusual characteristics. *BMC Bioinformatics*, 24(1):491, 2023.

[82] E.Garrison, J.Sirén, A.M.Novak, G.Hickey, J.M.Eizenga, E.T.Dawson, W.Jones, S.Garg, C.Markello, M.F.Lin MF, B.Paten B, and R.Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, October 2018. tex.ids: garrisonVariationGraphToolkit2018, garrisonVariationGraphToolkit2018a, garrisonVariationGraphToolkit2018c.

[83] Jordan M. Eizenga, Adam M. Novak, Emily Kobayashi, Flavia Villani, Cecilia Cisar, Simon Heumos, Glenn Hickey, Vincenza Colonna, Benedict Paten,

and Erik Garrison. Efficient dynamic variation graphs. *Bioinformatics*, 36(21):5139–5144, 2021.

[84] Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In *47th International Conference on Current Trends in Theory and Practice of Computer Science, (SOFSEM)*, volume 12607 of *Lecture Notes in Computer Science*, pages 608–622. Springer, 2021.

[85] Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. *Theoretical Computer Science*, 975:114128, 2023.

[86] Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Transactions on Algorithms*, 19(3):21:1–21:25, 2023.

[87] Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing elastic founder graphs. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation (ISAAC)*, volume 212 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[88] Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing founder graphs. *Algorithmica*, 85(6):1586–1623, 2023.

[89] Liao et al. A draft human pangenome reference. *Nature*, 617(7960):312 – 324, 2023.

[90] Susan Fairley, Ernesto Lowy-Gallego, Emily Perry, and Paul Flicek. The International Genome Sample Resource (IGSR) collection of open human genomic variation resources. *Nucleic Acids Research*, 48(D1):D941–D947, 10 2019.

[91] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997.

[92] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[93] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[94] Estéban Gabory, Chang Liu, Grigorios Loukides, Solon P. Pissis, and Wiktor Zuba. Space-efficient indexes for uncertain strings. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4828–4842, 2024.

[95] Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Comparing elastic-degenerate strings: Algorithms, lower bounds, and applications. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 259 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[96] Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Pangenome comparison via ED strings. *Frontiers Bioinformatics*, 4 - 2024, 2024.

[97] Estéban Gabory, Eric Rivals, Michelle Sweering, Hilde Verbeek, and Pengfei Wang. Periodicity of degenerate strings. In *Prague Stringology Conference 2023*, page 42, 2023.

[98] Estéban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string comparison. *Information and Computation*, 304:105296, 2025.

[99] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):2:1–2:54, 2020.

[100] Z Galil and R Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, mar 1986.

[101] Younan Gao, Meng He, and Yakov Nekrich. Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 54:1–54:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[102] Pawel Gawrychowski, Samah Ghazawi, and Gad M. Landau. On indeterminate strings matching. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 161 of *LIPIcs*, pages 14:1–14:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[103] Paweł Gawrychowski and Przemysław Uznański. Towards unified approximate pattern matching for Hamming and l_1 distance. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 62:1–62:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[104] Tingjian Ge and Zheng Li. Approximate substring matching over uncertain strings. *Proceedings of the VLDB Endowment*, 4(11):772–782, 2011.

[105] Daniel Gibney. An efficient elastic-degenerate text index? not likely. In Christina Boucher and Sharma V. Thankachan, editors, *27th International Symposium on String Processing and Information Retrieval*, volume 12303 of *Lecture Notes in Computer Science*, pages 76–88. Springer, 2020.

[106] Daniel Gibney, Gary Hoppenworth, and Sharma V. Thankachan. Simple reductions from formula-sat to pattern matching on labeled graphs and subtree isomorphism. In *4th SIAM Symposium on Simplicity in Algorithms (SOSA)*, pages 232–242, 2021.

[107] Jian Gong, Reynold Cheng, and David W. Cheung. Efficient management of uncertainty in XML schema matching. *VLDB J.*, 21(3):385–409, 2012.

[108] Szymon Grabowski and Marcin Raniszewski. Sampled suffix array with minimizers. *Software: Practice and Experience*, 47(11):1755–1771, 2017.

[109] Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. online pattern matching on similar texts. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 78 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[110] Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In *Proceedings of the 5th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 697–704, 1994.

[111] Manish Gupta and Michael Bendersky. Information retrieval with verbose queries. In Ricardo Baeza-Yates, Mounia Lalmas, Alistair Moffat, and Berthier A. Ribeiro-Neto, editors, *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015*, pages 1121–1124. ACM, 2015.

[112] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[113] Kersten Hall and Neeraja Sankaran. Dna translated: Friedrich miescher's discovery of nuclein in its original context. *The British Journal for the History of Science*, 54(1):99–107, February 2021.

[114] Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978.

[115] Monika Rauch Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In Efthimis N. Efthimiadis, Susan T. Dumais, David Hawking, and Kalervo Järvelin, editors, *SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, Washington, USA, August 6-11, 2006*, pages 284–291. ACM, 2006.

[116] Jan Holub, William F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *Journal of Discrete Algorithms*, 6(1):37–50, 2008.

[117] https://gnomad.broadinstitute.org/, 2024.

[118] https://topmed.nhlbi.nih.gov/, 2024.

[119] https://www.internationalgenome.org/, 2024.

[120] Ming Hua, Jian Pei, and Xuemin Lin. Ranking queries on uncertain data. *VLDB J.*, 20(1):129–153, 2011.

[121] Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate texts. In *11th International Conference on Language and Automata Theory and Applications (LATA)*, volume 10168 of *Lecture Notes in Computer Science*, pages 131–142, 2017.

[122] Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate strings. *Information and Computation*, 279:104616, 2021.

[123] Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios K. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundamenta Informaticae*, 71(2-3):259–277, 2006.

[124] Costas S. Iliopoulos, Laurent Mouchard, and Mohammad Sohel Rahman. A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching. *Mathematics in Computer Science*, 1(4):557–569, 2008.

[125] Costas S. Iliopoulos and Jakub Radoszewski. Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties. In *27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPIcs*, pages 8:1–8:12, 2016.

[126] R. Impagliazzo and R. Paturi. Complexity of k-sat. In *Proceedings. Fourteenth Annual IEEE Conference on Computational Complexity (Formerly: Structure in Complexity Theory Conference) (Cat.No.99CB36317)*, pages 237–240, 1999.

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

[128] Kayla Jacobs, Alon Itai, and Shuly Wintner. Acronyms: identification, expansion and disambiguation. *Annals of Mathematics and Artificial Intelligence*, 88(5-6):517–532, 2020.

[129] Chirag Jain, Arang Rhie, Nancy Hansen, Sergey Koren, and Adam M. Phillippy. Long-read mapping to repetitive reference sequences using winnowmap2. *Nat Methods*, 19:705–710, 2022.

[130] Jeffrey Jestes, Feifei Li, Zhepeng Yan, and Ke Yi. Probabilistic string similarity joins. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 327–338. ACM, 2010.

[131] Jiaojiao Jiang, Steve Versteeg, Jun Han, Md. Arafat Hossain, Jean-Guy Schneider, Christopher Leckie, and Zeinab Farahmandpour. P-gram: Positional n-gram for the clustering of machine-generated messages. *IEEE Access*, 7:88504–88516, 2019.

[132] Bhargav Kanagal and Amol Deshpande. Indexing correlated probabilistic databases. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 455–468. ACM, 2009.

[133] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[134] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[135] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, pages 181–192, 2001.

[136] A.E. Kel, E. Gössling, I. Reuter, E. Cheremushkin, O.V. Kel-Margoulis, and E. Wingender. MATCHTM: a tool for searching transcription factor binding sites in DNA sequences. *Nucleic Acids Research*, 31(13):3576–3579, 07 2003.

[137] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 756–767. ACM, 2019.

[138] Dominik Kempa and Tomasz Kociumaka. Breaking the O(n)-barrier in the construction of compressed suffix arrays and suffix trees. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 5122–5202. SIAM, 2023.

[139] Katrin Kirchhoff and Anne M. Turner. Unsupervised resolution of acronyms and abbreviations in nursing notes using document-level context models. In Cyril Grouin, Thierry Hamon, Aurélie Névéol, and Pierre Zweigenbaum, editors, *Proceedings of the Seventh International Workshop on Health Text Mining and Information Analysis, Louhi@EMNLP 2016, Austin, TX, USA, November 5, 2016*, pages 52–60. Association for Computational Linguistics, 2016.

[140] Jon M. Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2006.

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

[142] Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Pattern matching and consensus problems on weighted sequences and profiles. In Seok-Hee Hong, editor, *27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia*, volume 64 of *LIPIcs*, pages 46:1–46:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[143] Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Pattern matching and consensus problems on weighted sequences and profiles. *Theory of Computing Systems*, 63(3):506–542, 2019.

[144] Janne H. Korhonen, Petri Martinmäki, Cinzia Pizzi, Pasi Rastas, and Esko Ukkonen. MOODS: fast search for position weight matrix matches in DNA sequences. *Bioinformatics*, 25(23):3181–3182, 2009.

[145] Dmitry Kosolobov and Nikita Sivukhin. Construction of sparse suffix trees and LCE indexes in optimal time and space, 2024.

[146] Divesh R. Kubal and Apurva Nagvenkar. Effective ensembling of transformer based language models for acronyms identification. In Amir Pouran Ben Veyseh, Franck Dernoncourt, Thien Huu Nguyen, Walter Chang, and Leo Anthony Celi, editors, *Proceedings of the Workshop on Scientific Document Understanding co-located with 35th AAAI Conference on Artificial Inteligence, SDU@AAAI 2021, Virtual Event, February 9, 2021*, volume 2831 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021.

[147] Cheng-Ju Kuo, Maurice H. T. Ling, Kuan-Ting Lin, and Chun-Nan Hsu. BIOADI: a machine learning approach to identifying abbreviations and definitions in biological literature. *BMC Bioinformatics*, 10(S-15):7, 2009.

[148] Dominik Köppl and Jannik Olbrich. Hardness results on characteristics for elastic-degenerated strings, 2024.

[149] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

[150] Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[151] Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.

[152] Gad M Landau, Uzi Vishkin, and Ruth Nussinov. An efficient string matching algorithm with k differences for nucleotide and amino acid sequences. *Nucleic Acids Research*, 14(1):31–46, 1986.

[153] Mark V. Lawson. *Finite Automata*. Chapman and Hall/CRC, 2004.

[154] Chris-André Leimeister and Burkhard Morgenstern. kmacs: the *k*-mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014.

[155] Yuxuan Li, James Bailey, Lars Kulik, and Jian Pei. Efficient matching of substrings in uncertain sequences. In Mohammed Javeed Zaki, Zoran Obradovic, Pang-Ning Tan, Arindam Banerjee, Chandrika Kamath, and Srinivasan Parthasarathy, editors, *Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014*, pages 767–775. SIAM, 2014.

[156] Jie Liu, Caihua Liu, and Yalou Huang. Multi-granularity sequence labeling model for acronym expansion identification. *Information Sciences*, 378:462–474, 2017.

[157] Glennis A. Logsdon, Mitchell R. Vollger, and Evan E. Eichler. Long-read human genome sequencing and its applications. *Nature Reviews Genetics*, 21(10):597–614, 2020.

[158] M. Lothaire. *Combinatorics on Words*. Advanced Book Program. Addison-Wesley, Advanced Book Program, World Science Division, 1983.

[159] Grigorios Loukides and Solon P. Pissis. Bidirectional string anchors: A new string sampling mechanism. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 64:1–64:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[160] Felipe A. Louza, Neerja Mhaskar, and W. F. Smyth. A new approach to regular & indeterminate strings. *Theoretical Computer Science*, 854:105–115, 2021.

[161] Veli Mäkinen, Bastien Cazaux, Massimo Equi, Tuukka Norri, and Alexandru I. Tomescu. Linear time construction of indexable founder block graphs. In Carl Kingsford and Nadia Pisanti, editors, *20th International Conference on Algorithms in Bioinformatics (WABI)*, volume 172 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[162] Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In José R. Correa, Alejandro Hevia, and Marcos A. Kiwi, editors, *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer, 2006.

[163] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for online string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[164] Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4):191–197, 1994.

[165] Olena Medelyan and Ian H. Witten. Thesaurus based automatic keyphrase indexing. In Gary Marchionini, Michael L. Nelson, and Catherine C. Marshall, editors, *ACM/IEEE Joint Conference on Digital Libraries, JCDL 2006, Chapel Hill, NC, USA, June 11-15, 2006, Proceedings*, pages 296–297. ACM, 2006.

[166] Friedrich Miescher. Ueber die chemische zusammensetzung der eiterzellen. *Medizinisch-chemische Untersuchungen*, 4:441–460, 1871.

[167] Njagi Moses Mwaniki, Erik Garrison, and Nadia Pisanti. Fast exact string to D-texts alignments. In Hesham Ali, Ning Deng, Ana L. N. Fred, and Hugo Gamboa, editors, *16th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC)*, pages 70–79. SCITEPRESS, 2023.

[168] Njagi Moses Mwaniki and Nadia Pisanti. Optimal sequence alignment to ED-strings. In Mukul S. Bansal, Zhipeng Cai, and Serghei Mangul, editors, *18th International Symposium Bioinformatics Research and Applications (ISBRA)*, volume 13760 of *Lecture Notes in Computer Science*, pages 204–216. Springer, 2022.

[169] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(1-3):87–101, 2003.

[170] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[171] Claro Noda, Shashi Prabh, Mário Alves, Thiemo Voigt, and Carlo Alberto Boano. CRAWDAD cister/rssi. https://dx.doi.org/10.15783/C7WC75, 2022.

[172] Benedict Paten, Adam M. Novak, Jordan M. Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome Research*, 27(5):665–676, 2017.

[173] Nadia Pisanti, Henry Soldano, and Mathilde Carpentier. Incremental inference of relational motifs with a degenerate alphabet. In *16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3537 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2005.

[174] Nadia Pisanti, Henry Soldano, Mathilde Carpentier, and Joël 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.

[175] Solon P. Pissis and Ahmad Retha. Dictionary matching in elastic-degenerate texts with applications in searching VCF files online. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA)*,

volume 103 of *LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[176] Cinzia Pizzi, Pasi Rastas, and Esko Ukkonen. Finding significant matches of position weight matrices in linear time. *IEEE ACM Transactions on Computational Biology and Bioinformatics*, 8(1):69–79, 2011.

[177] Petr Procházka, Ondrej Cvacho, Lubos Krcál, and Jan Holub. Backward pattern matching on elastic-degenerate strings. *SN Compouter Science*, 4(5):442, 2023.

[178] Yinian Qi, Rohit Jain, Sarvjeet Singh, and Sunil Prabhakar. Threshold query optimization for uncertain data. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 315–326. ACM, 2010.

[179] Niranjan Rai and Xiang Lian. Distributed probabilistic top-k dominating queries over uncertain databases. *Knowledge and Information Systems*, 65(11):4939–4965, 2023.

[180] Goran Rakocevic, Vladimir Semenyuk, Wan-Ping Lee, James Spencer, John Browning, Ivan J. Johnson, Vladan Arsenijevic, Jelena Nadj, Kaushik Ghose, Maria C. Suciu, Sun-Gou Ji, Gülfem Demir, Lizao Li, Berke Ç. Toptaş, Alexey Dolgoborodov, Björn Pollex, Iosif Spulber, Irina Glotova, Péter Kómár, Andrew L. Stachyra, Yilong Li, Milos Popovic, Morten Källberg, Amit Jain, and Deniz Kural. Fast and accurate genomic analyses using genome graphs. *Nature Genetics*, 51:354–362, 2019.

[181] Nicola Rizzo, Massimo Equi, Tuukka Norri, and Veli Mäkinen. Elastic founder graphs improved and enhanced. *Theoretical Computer Science*, 982:114269, 2024.

[182] Nicola Rizzo and Veli Mäkinen. Indexable elastic founder graphs of minimum height. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 223 of *LIPIcs*, pages 19:1–19:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[183] Nicola Rizzo and Veli Mäkinen. Linear time construction of indexable elastic founder graphs. In Cristina Bazgan and Henning Fernau, editors, *33rd International Workshop on Combinatorial Algorithms (IWOCA)*, volume 13270 of *Lecture Notes in Computer Science*, pages 480–493. Springer, 2022.

[184] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[185] Consuelo Romero-Sánchez, Natalia Hernández, Lorena Chila-Moreno, Karen Jiménez, Diana Padilla, Juan Manuel Bello-Gualtero, and Wilson Bautista-Molano. HLA-B allele, genotype, and haplotype frequencies in a group of healthy individuals in colombia. *Journal of Clinical Rheumatology*, 27(6S):S148–S152, September 2021.

[186] Milan Ruzic. Constructing efficient dictionaries in close to sorting time. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2008.

[187] Marie-France Sagot, Alain Viari, and Henry Soldano. Multiple sequence comparison: A peptide matching approach. In *6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 937 of *Lecture Notes in Computer Science*, pages 366–385. Springer, 1995.

[188] Marie-France Sagot, Alain Viari, and Henry Soldano. Multiple sequence comparison - A peptide matching approach. *Theoretical Computer Science*, 180(1-2):115–137, 1997.

[189] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. Winnowing: Local algorithms for document fingerprinting. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 76–85. ACM, 2003.

[190] Ariel S. Schwartz and Marti A. Hearst. A simple algorithm for identifying abbreviation definitions in biomedical text. In Russ B. Altman, A. Keith Dunker, Lawrence Hunter, and Teri E. Klein, editors, *Proceedings of the 8th Pacific Symposium on Biocomputing, PSB 2003, Lihue, Hawaii, USA, January 3-7, 2003*, pages 451–462, 2003.

[191] Qingmin Shi and Joseph F. JáJá. Novel transformation techniques using q-heaps with applications to computational geometry. *SIAM Journal on Computing*, 34(6):1474–1492, 2005.

[192] Ariel Shiftan and Ely Porat. Set intersection and sequence matching with mismatch counting. *Theoretical Computer Science*, 638:3–10, 2016.

[193] Sarvjeet Singh, Chris Mayfield, Sunil Prabhakar, Rahul Shah, and Susanne E. Hambrusch. Indexing uncertain categorical data. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 616–625. IEEE Computer Society, 2007.

[194] Sarvjeet Singh, Chris Mayfield, Rahul Shah, Sunil Prabhakar, Susanne E. Hambrusch, Jennifer Neville, and Reynold Cheng. Database support for probabilistic attributes and tuples. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 1053–1061. IEEE Computer Society, 2008.

[195] Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. In *18th International Conference on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPIcs*, pages 4:1–4:13, 2018.

[196] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

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

[198] Zitan Sun, Xin Huang, Jianliang Xu, and Francesco Bonchi. Efficient probabilistic truss indexing on uncertain graphs. In Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia, editors, *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, pages 354–366. ACM / IW3C2, 2021.

[199] Kazem Taghva and Jeff Gilbreth. Recognizing acronyms and their definitions. *International Journal on Document Analysis and Recognition*, 1(4):191–198, 1999.

[200] Yufei Tao, Reynold Cheng, Xiaokui Xiao, Wang Kay Ngai, Ben Kao, and Sunil Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 922–933. ACM, 2005.

[201] Yufei Tao, Xiaokui Xiao, and Reynold Cheng. Range search on multidimensional uncertain data. *ACM Transactions on Database Systems*, 32(3):15, 2007.

[202] Chris Thachuk. Indexing hypertext. *Journal of Discrete Algorithms*, 18:113–122, 2013.

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

[204] Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Ambujam Krishnan, and Srinivas Aluru. A greedy alignment-free distance estimator for phylogenetic inference. *BMC Bioinformatics*, 18(8):238:1–238:8, 2017.

[205] Gerry Tonkin-Hill, Inigo Martincorena, Roberto Amato, Andrew RJ Lawson, Moritz Gerstung, Ian Johnston, David K Jackson, Naomi Park, Stefanie V Lensing, Michael A Quail, Sónia Gonçalves, Cristina Ariani, Michael Spencer Chapman, William L Hamilton, Luke W Meredith, Grant Hall, Aminu S Jahun, Yasmin Chaudhry, Myra Hosmillo, Malte L Pinckert, Iliana Georgana, Anna Yakovleva, Laura G Caller, Sarah L Caddy, Theresa Feltwell, Fahad A Khokhar, Charlotte J Houldcroft, Martin D Curran, Surendra Parmar, The COVID-19 Genomics UK (COG-UK) Consortium, Alex Alderton, Rachel Nelson, Ewan M

Harrison, John Sillitoe, Stephen D Bentley, Jeffrey C Barrett, M Estee Torok, Ian G Goodfellow, Cordelia Langford, Dominic Kwiatkowski, and Wellcome Sanger Institute COVID-19 Surveillance Team. Patterns of within-host genetic diversity in SARS-CoV-2. *eLife*, 10:e66857, aug 2021.

[206] Igor Ulitsky, David Burstein, Tamir Tuller, and Benny Chor. The average common substring approach to phylogenomic reconstruction. *Journal of Computational Biology*, 13(2):336–350, 2006.

[207] Kazutoshi Umemoto, Ruihua Song, Jian-Yun Nie, Xing Xie, Katsumi Tanaka, and Yong Rui. Search by screenshots for universal article clipping in mobile apps. *ACM Transactions on Information Systems.*, 35(4):34:1–34:29, 2017.

[208] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 645–654. IEEE Computer Society, 2010.

[209] Amir Pouran Ben Veyseh, Franck Dernoncourt, Quan Hung Tran, and Thien Huu Nguyen. What does this acronym mean? Introducing a new dataset for acronym identification and disambiguation. In Donia Scott, Núria Bel, and Chengqing Zong, editors, *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020*, pages 3285–3301. International Committee on Computational Linguistics, 2020.

[210] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[211] James D Watson and Francis HC Crick. Molecular structure of nucleic acids: a structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, 1953.

[212] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.

[213] Aaron M. Wenger et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nature biotechnology*, 37:1155–1162, 2019.

[214] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005.

[215] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega, 2023.

[216] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. Index-based optimal algorithm for computing k-cores in large uncertain graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 64–75. IEEE, 2019.

[217] Ke Yi, Feifei Li, George Kollios, and Divesh Srivastava. Efficient processing of top-k queries in uncertain databases with x-relations. *IEEE Transactions on Knowledge and Data Engineering*, 20(12):1669–1682, 2008.

[218] Liming Zhan, Ying Zhang, Wenjie Zhang, and Xuemin Lin. Identifying top k dominating objects over uncertain data. In Sourav S. Bhowmick, Curtis E. Dyreson, Christian S. Jensen, Mong-Li Lee, Agus Muliantara, and Bernhard Thalheim, editors, *Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part I*, volume 8421 of *Lecture Notes in Computer Science*, pages 388–405. Springer, 2014.

[219] Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Improved design and analysis of practical minimizers. *Bioinformatics*, 36(Supplement_1):i119–i127, 07 2020.