SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA
DEPARTMENT OF COMPUTER SCIENCES, SYSTEMS AND COMMUNICATIONS

PhD program in Computer Science Cycle XXXIII

COMBINATORIAL METHODS FOR THE ANALYSIS OF RELATED GENOMIC SEQUENCES

Giulia Bernardini
(*827377*)

Tutor: *Prof. Leonardo Mariani*

Supervisor: *Prof. Paola Bonizzoni*

Co-supervisor: *Prof. Nadia Pisanti*

Coordinator: *Prof. Leonardo Mariani*

ACADEMIC YEAR 2019/2020

Acknowledgements

First and foremost, I would like to thank my supervisors: **Paola**, who welcomed me in her lab three years ago and made me constantly grow over the years, and **Nadia**, who is the reason why I undertook my academic journey, supported me from the very beginning and guided me through to the end. None of this would have been possible without you.

A heartfelt thanks goes to another two great mentors of mine, **Solon** and **Paweł**, for their guidance, their invaluable insights on a wide range of subjects and for the countless enriching conversations I had with them.

I deeply thank my awesome colleagues and office mates **Marco**, **Luca**, **Simone**, **Murray** and **Stefano** for making my workplace such a fun and pleasant place to be. This last, crazy year has been a bit empty without your company. I also thank **Gianluca** and **Raffaella** for the useful discussions I had with them.

A special thanks to all the other collaborators with whom I carried out an always stimulating research and spent some quality time too: **Grigorios**, **Roberto**, **Costas**, **Alessio**, **Leen**, **Giovanna**, **Gabriele**, **Giulia**, **Garance**, **Michelle**, to name a few.

It goes without saying that I am most grateful to my whole family, which has always supported and encouraged me, and to **Chiara** and **Alessandro**, who are the bedrocks of my life.

Contents

Chapter 1

Introduction

The recent development of computational pan-genomics is posing new challenges to computer scientists, requiring, among other things, to lay solid theoretical foundations for this new kind of biological data. The availability of a large number of sequenced genomes impacts many different application domains, ranging from the study of microbial and viral genomes to human genetic diseases, cancer and phylogenies. Motivated by the importance of the applications and by the novelty of the field, the main goal of this thesis is to develop new algorithmic frameworks to deal with (i) a convenient representation of a set of similar genomes and (ii) phylogenetic data, with particular attention to the increasingly accurate tumor phylogenies.

With the exception of Chapter 6, which is more practice-oriented, the focus of the whole work is mainly theoretical, the intent being to lay firm algorithmic foundations for the problems by investigating their combinatorial aspects, rather than to provide practical tools for attacking them. The initial theoretical studies have been completed, whenever relevant, with either experimentation on real data or proof-of-concept trials (see Chapters 4, 6).

To provide deep theoretical insights on a computational problem is fundamental, as this allows a rigorous analysis of existing methods, identifying their strong and weak points, providing details on how they perform and helping to decide which problems need to be further addressed. In addition, it is often the case where new theoretical results (algorithms, data structures and reductions to other well-studied problems) can be either directly applied or adapted to fit the model of a practical problem, or at least they serve as inspiration for developing new practical tools.

In this dissertation, particular attention is devoted to exploring the connections between the considered computational biological problems and other abstract problems that arise in different contexts. The virtue of such approach is to provide, from time to time, new algorithmic techniques that take advantage of existing efficient solutions to the more general problems, or conditional lower bounds for the complexity of the original problems.

1.1 Part I

The first part of this thesis is devoted to methods for handling an *Elastic-Degenerate (ED) text*, a computational object that compactly encodes a collection of similar texts. The study of such an object in bioinformatics is motivated by the advantages of representing a collection of closely related genomes, including their variation, as a *pan-genome*. A pan-genome is, in general, any collection of genomic sequences to be analyzed jointly or to be used as a reference for a population. Traditionally, a *reference genome* is a single genomic sequence used as a representative of its species, to which, for example, fragments of newly sequenced genomes are mapped. If a single annotated reference genome generally provides a good approximation of any individual genome, in loci with polymorphic variations mapping and sequence comparison often fail their purposes, and a pan-genome would be a better reference text.

One of the advantages of representing a pan-genome as an ED text is that it can be straightforwardly obtained from files in the *Variant Call Format* (VCF), which has become the standard way of storing variants for pan-genomes and in next-generation sequencing. The VCF format was initially developed for the 1000 Genomes Project [336], and has also been adopted by other projects like UK10K [109] and dbSNP [323]. Unlike other possible representations of a pan-genome proposed in the literature, the string-like structure of ED texts makes it possible to read them sequentially, *on-line*, and thus to allow on-line algorithms for performing crucial tasks like pattern matching. The benefit of solving this problem on-line is to avoid the burden of building disk-based indexes or rebuilding them with every update in the sequences. Actually, the usage of indexes carries the assumption that the data is static or changes very infrequently, something that cannot always be assumed in the context of pan-genomics.

In Chapter 2 (paper [58]) we address the fundamental problem of matching a pattern on an ED text on-line. We start by conducting an exhaustive and enlightening analysis of the problem, by proving a conditional lower bound both for the problem itself and for the sub-problem constituting the core of all the on-line algorithmic frameworks proposed in the literature so far. Not only give these results very detailed insights on the inherent complexity of the problem, they have also inspired us to design an even faster algorithm that combines string periodicity, fast Fourier transform and fast matrix multiplication. As this is the first time that these tools are combined together in string algorithms, and an important building block in our solution is a new substring-selection method that might find applications in other problems, the scope of this result seems to extend beyond the mere solution of the pattern matching problem.

As genomic sequences are typically endowed with polymorphisms and sequencing errors, the existence of an exact occurrence of a pattern in an ED text encoding a set of similar genomes is often too strong an assumption. This issue brought us to consider the approximate version of the problem, and to develop a combinatorial algorithm to perform approximate pattern matching on an ED text. The resulting publication is presented in Chapter 3 (paper [60]).

After working on the problem of matching a string in an ED text, it was only natural to consider the problem of matching two ED texts. The work presented in Chapter 4 (papers [27, 28]) is a first step in that direction, as it attacks the problem of comparing two *Generalized degenerate (GD) strings*, a restricted version of ED text that models

a gapless multiple sequence alignment. The main result, an asymptotically optimal algorithm for deciding whether two GD strings have a non-empty intersection, can be used as a basic tool in various GD string processing applications. We apply it to compute all the palindromes of a GD string, a task needed in biological applications to identify *hairpins*, patterns that occur in single-stranded DNA or, more commonly, in RNA. In the spirit of exploring the connections with other problems, we complement our algorithm for decomposing a GD string into palindromic factors with a non-trivial conditional lower bound, and give an alternative solution to the GD string intersection problem that employs an automata-based approach.

1.2 Part II

The second part of this thesis focuses on another kind of biological data: phylogenetic trees. A phylogeny is meant to describe the evolutionary relationships among a group of items, be they species of living beings, genes, natural languages, ancient manuscripts or cancer cells. Depending on the characteristics of the evolutionary relationships one needs to describe, phylogenies are modelled, on a case-by-case basis, as trees (either rooted or not) or directed acyclic graphs. The simplest mathematical object that has been historically used to describe the evolution over time of a set of species is a rooted, leaf-labelled tree, whose leaves are in a one-to-one correspondence with the species.

While a leaf-labelled tree is a reasonable description of the evolutionary history of a set of living species whose ancestors (corresponding to internal nodes) are extinct, this model does not fit well with the evolution of a tumor, in which cancer cells with "ancestral" mutations are alive at the very same time as their descendants. Instead, the simplest mathematical object that may be used to represent the evolution of a tumor is a rooted *fully-labelled* tree, whose entire set of nodes is in a one-to-one correspondence with the mutations of cancer cells. The motivation for studying such object is that recent methods to infer the evolution and progression of cancer have made it possible to develop targeted therapies for treating the disease. As discussed in several studies, understanding the history of accumulation and the prevalence of somatic mutations during cancer progression is a fundamental step to devise new treatment strategies. Given the importance of the task, a multitude of methods for cancer phylogeny reconstruction have been developed over the years, encouraged by the diversity of the available data; in particular, we are witnessing a shift from bulk sequencing data towards single-cell data and hybrid approaches.

Having many different tools accomplishing the same task requires solid methods to compare their results. While there is an ample literature on measures of distance for classical phylogenies (modelled as leaf-labelled trees), the investigation of methods for comparing tumor phylogenies is still in its infancy. Chapters 5 and 6 study the problem of comparing tumor phylogenies under two different points of view. The results reported in Chapter 5 (papers [52, 51]) attack the problem from a purely combinatorial point of view. In [52] we take the simplest possible model of tumor phylogeny, a fully-labelled rooted tree: we assume that the two trees to be compared have the same number of nodes and are labelled by the same set of mutations, meaning that there is a bijection between the nodes of each tree and the set of labels. We define a measure of distance based on two operations on the trees. Like other similar-in-spirit rearrangement

distances for leaf-labelled trees, such distance turns out to be NP-hard to decide, while the distances defined with respect to, in turn, only one of the two operations, are polinomially computable.

The follow-up paper [51] stemmed from a meaningful discussion that took place after the presentation of [52] at the conference CPM 2019. The contribution of the second paper is twofold. We start by studying in depth one of the restricted versions of the rearrangement distance, providing a two-way fine-grained reduction from (and to) the problem of computing the maximum cardinality matching in multiple bipartite graphs. In one direction, this reduction provides a method for computing the distance that takes advantage from the highly-efficient solutions to the maximum matching problem; in the other direction, the reduction gives insights on the time complexity of the problem. The other main contribution is a constant-factor approximation algorithm for the actual rearrangement distance, which has the additional merit of offering a clearer understanding of the structure of the problem.

If in Chapter 5 we took the simplest (and perhaps unrealistic) model of tumor phlyogeny, as the main intent was to unveil the combinatorial structure of the problem, rather than to provide an actual method for comparing real phylogenies, in Chapter 6 (paper [100]) we radically change our point of view, and aim at designing a comparison method that is meaningful in practice. We thus tweak the model of tumor phylogeny so as to reflect the characteristics of the real cancer phylogenies outputted by most inferring methods, and consider fully-multi-labelled trees with poly-occurring labels, meaning that not only can a node of a tree be labelled by multiple mutations, but each mutation may label more than one node. With the goal of addressing the major limitations of the existing comparison methods, we then define a similarity score inspired by the triplet distance for leaf-labelled trees, and perform an extensive experimental study to show its effectiveness and advantages with respect to other tools.

A problem which is closely related to the comparison of tumor phylogenies is to reconstruct them from the binary data given, for example, by single-cell sequencing methods. As such data are often incomplete, our attention has turned to a longstanding problem in computational phylogenetics, for which an asymptotically optimal solution was hitherto unknown: to decide whether a collection of species, described by a set of characters with some missing data, can be arranged in a leaf-labelled phylogenetic tree that enjoys particular properties. Previous solutions to a specific formulation of this problem follow a graph theoretic approach, and rely on complex, pre-existing data structures used as a blackbox. In line with (almost all) the other works presented in this dissertation, the research objective in this paper is to reveal and use the combinatorial properties of the problem to obtain faster algorithms. This way of proceeding led us to describe a fairly simple data structure, ad-hoc for the problem, that implies the first asymptotically optimal algorithm for the problem. This last work is still in preparation: indeed, even if the theoretical result reported in Chapter 7 is already complete, we believe that an experimental assessment of the algorithm would strengthen the paper.

In order not to slow down the rhythm of this dissertation, and not to bore the reader from the very beginning with a massive preliminary section, I will assume that they are already familiar with the most common combinatorial objects. All such basic definitions can anyway be found in Appendix A: more specific definitions will be given in the preliminary section of each chapter.

1.3 Roadmap and Synopsis of the Publications

This dissertation is divided into two parts, which are determined by the general scope of the papers they gather. To help the reader navigate the document, each chapter begins with a preamble containing the status of the papers it consists of (whether and where they are published or submitted) and a schematic summary of the contents.

Part I contains the results on the analysis of degenerate strings: Chapter 2 (paper [58]) is on exact pattern matching on ED text, Chapter 3 (paper [60]) is about approximate pattern matching on ED text, Chapter 4 (papers [28] and [27]) is on comparison between generalized degenerate strings.

Part II gathers the results on phylogenetic trees: Chapter 5 (papers [52, 51]) is about a rearrangement distance for comparing tumor phylogenies, Chapter 6 (paper [100]) contains the only practice-oriented result of this thesis, a similarity score for tumor phylogenies, and Chapter 7 (paper [50]) presents an algorithm for solving the Incomplete Directed Perfect Phylogeny in linear time.

In the rest of this introduction I will give an overview of the results included in the main parts of the thesis, followed by a complete list of my publications. The aim of the following sections is to help the reader get the big picture of the thesis, orientate themselves on the document and find out which chapters might interest them. I will highlight the computational problems addressed in each chapter with a short synopsis and give an idea of the main algorithmic techniques they make use of, avoiding formal definitions and without diving into details.

1.3.1 Elastic-Degenerate String Matching via Matrix Multiplication

An elastic-degenerate (ED) string is a sequence of n sets of strings of total length N, which models a set of similar sequences, e.g., a set of closely related genomes. We call *deterministic segments* the sets consisting of a single string, *non-deterministic segments* the rest of the sets. Non-deterministic segments may be used to represent a position in a DNA sequence that can have multiple possible alternatives. These are used to encode the consensus of a population of sequences [108, 27, 165] in a multiple sequence alignment (MSA) in the presence of insertions or deletions.

The ED string matching (EDSM) problem is to find all occurrences of a pattern of length m in an ED text *on-line*, meaning that the ED text is read sequentially, segment by segment. The EDSM problem has received some attention in the combinatorial pattern matching community, and an $\mathcal{O}(nm^{1.5}\sqrt{\log m}+N)$-time algorithm is known [31]. The standard assumption in the prior work on this question is that N is substantially larger than both n and m (e.g., when an ED string encodes a collection of genomes, N is the total length of all the genomes) and thus we would like to have a linear dependency on the former. Under this assumption, the natural open problem is whether we can decrease the 1.5 exponent in the time complexity, similarly as in the related (but, to the best of our knowledge, not equivalent) *word break* problem [39]. A recent paper [170] provides a conditional lower bound showing that, even with arbitrary polynomial preprocessing time, an index for an ED text with n segments that can perform queries on a pattern of length m in time $\mathcal{O}(n^{\alpha}m^{\beta})$, for constants α and β

where $\alpha < 1$ or $\beta < 1$, would violate the widely believed Strong Exponential Time Hypothesis (SETH) [208, 207].

Our starting point is a different conditional lower bound for the EDSM problem, that does not allow the construction of an index using arbitrary polynomial preprocessing time. We use the popular combinatorial Boolean Matrix Multiplication (BMM) conjecture stating that there is no truly subcubic *combinatorial* algorithm for BMM [10]. By designing an appropriate reduction we show that a combinatorial algorithm solving the EDSM problem in $\mathcal{O}(nm^{1.5-\epsilon} + N)$ time, for any $\epsilon > 0$, refutes this conjecture. Our reduction should be understood as an indication that decreasing the exponent requires fast matrix multiplication.

String periodicity and fast Fourier transform are two standard tools in string algorithms. Our main technical contribution is that we successfully combine these tools with fast matrix multiplication to design a non-combinatorial $\tilde{\mathcal{O}}(nm^{\omega-1} + N)$-time algorithm for EDSM, where ω denotes the matrix multiplication exponent and the $\tilde{\mathcal{O}}(\cdot)$ notation suppresses polylog factors. To the best of our knowledge, we are the first to combine these tools in string algorithms. In particular, using the fact that $\omega < 2.373$ [242, 362], we obtain an $\mathcal{O}(nm^{1.373} + N)$-time algorithm for EDSM. An important building block in our solution, that might find applications in other problems, is a method of selecting a small set of length-ℓ substrings of the pattern, called anchors, so that any occurrence of a string from an ED text segment contains at least one but not too many such anchors inside.

1.3.2 Approximate Pattern Matching on Elastic-Degenerate Text

Since genomic sequences are endowed with polymorphisms and sequencing errors, the existence of an exact occurrence of a pattern in an ED text encoding a set of similar genomes can result into a strong assumption. The aim of this work is to generalize the EDSM problem allowing some approximation in the occurrences of the input pattern. We suggest a simple on-line $\mathcal{O}(kmG + kN)$-time and $\mathcal{O}(m)$-space algorithm, G being the total number of strings in the ED text and $k > 0$ the maximum number of allowed substitutions in a pattern's occurrence, that is nonzero *Hamming distance*.

Our main contribution is an on-line $\mathcal{O}(k^2mG + kN)$-time and $\mathcal{O}(m)$-space algorithm where the type of edit operations allowed is extended to insertions and deletions as well, that is nonzero *edit distance*. These results are *good* in the sense that for *small* values of k the algorithms incur (essentially) no increase in time complexity with respect to the $\mathcal{O}(nm^2 + N)$-time and $\mathcal{O}(m)$-space algorithm presented in [178] for the exact case. Our solution relies on an ad-hoc modification of the Landau-Vishkin algorithm [240] for computing the edit distance, properly incorporated in the algorithmic framework of [178] for exact EDSM.

1.3.3 Comparing Degenerate Strings

String comparison is the core computational task in several string-processing applications, whether they process standard or uncertain strings. For example, to extract frequent patterns from a single string, to find common substrings among several strings, or to check whether a string is palindromic, one must have a tool for comparing two

strings. In this chapter we start looking into the problem of comparing two *generalized degenerate strings* (GD strings), a restricted variant of ED strings where the ith set contains strings of the same length k_i but this length can vary between different sets. We denote by W the sum of these lengths $k_0, k_1, \ldots, k_{n-1}$. A GD string can be used to represent in a compact form a *gapless* multiple sequence alignment (MSA) of fixed width, that is, for example, a high-scoring local alignment of multiple sequences.

Our main result is an $\mathcal{O}(N + M)$-time algorithm for deciding whether two GD strings of total sizes N and M, respectively, over an integer alphabet, have a non-empty intersection. This result is based on a combinatorial result of independent interest: although the intersection of two GD strings can be exponential in the total size of the two strings, it can be represented in *linear* space. As proof of concept, we then apply our string comparison tool to devise a simple algorithm for computing all palindromes in $\hat{S}$ in $\mathcal{O}(\min\{W, n^2\}N)$-time.

We complement this upper bound by showing a non-trivial $\Omega(n^2|\Sigma|)$ lower bound under the Strong Exponential Time Hypothesis [207, 208] for computing maximal palindromes in $\hat{S}$. We also show that a result, which is essentially the same as our string comparison linear-time algorithm, can be obtained by employing an automata-based approach.

1.3.4 A Rearrangement Distance for Fully-Labelled Trees

The problem of comparing trees representing the evolutionary histories of cancerous tumors has turned out to be crucial, since there is a variety of different methods which typically infer multiple possible trees. A departure from the widely studied setting of classical phylogenetics, where trees are leaf-labelled, tumoral trees are fully labelled, i.e., every node has a label.

In this work we provide a rearrangement distance measure between two fully-labelled trees. This notion originates from two operations: one which modifies the topology of the tree, the other which permutes the labels of the nodes, hence leaving the topology unaffected. While we show that the distance between two trees in terms of each such operation alone can be decided in polynomial time, the more general notion of distance when both operations are allowed is NP-hard to decide.

For what concerns the distance between two trees in terms of permutation operations alone we show, via a two-way reduction, that calculating the permutation distance between two trees on n nodes is equivalent, up to polylogarithmic factors, to finding the largest cardinality matching in a sparse bipartite graph. Due to the recent progress in the area of fine-grained complexity we now know, for many problems that can be solved in polynomial time, what is essentially the best possible exponent in the running time, conditioned on some plausible but yet unproven hypothesis [363]. For maximum matching, this is not the case yet, although we do have some understanding of the complexity of the related problem of computing the max-flow between all pairs of nodes [9, 238, 8]. So, even though our reductions do not tell us what is the best possible exponent in the running time, they do imply that it is the same as for maximum matching in a sparse bipartite graph. In particular, by plugging in the algorithm of Liu and Sidford [254], we obtain an $\tilde{\mathcal{O}}(n^{4/3+o(1)})$ time algorithm for computing the permutation distance between two trees on n nodes. The main technical novelty in our

reduction from permutation distance is that, even though the natural approach would result in multiple instances of weighted maximum bipartite matching, we manage to keep the graphs unweighted.

As for the actual rearrangement distance, we show that it is fixed-parameter tractable, and we give a simple, linear-time 4-approximation algorithm when one of the trees is binary. Moreover, we design a general linear-time constant-factor approximation algorithm that does not assume that the trees are binary. The algorithm consists of multiple phases, each of them introducing more and more structure into the currently considered instance, while making sure that we do not pay more than the optimal distance times some constant. To connect the number of steps used in every phase with the optimal distance, we introduce a new combinatorial object that can be used to lower bound the latter, inspired by the well-known algorithm for computing the majority [76].

1.3.5 Triplet-Based Similarity Score for Tumor Phylogenies

In this chapter we address the problem of comparing tumor phylognies from a more practical point of view. Indeed, several notions of distance or similarity have recently been proposed in the literature, but none of them has emerged as the golden standard. We identified two major limitations in the existing methods: first, they are not sensitive enough to detect even major differences in the topology of the trees, as we demonstrate with ad-hoc experiments. Second, they are not able to meaningfully compare trees where the same label is assigned to more than one node, a circumstance often occurring in real cases.

To overcome these limitations we propose `MP3`, the first similarity measure for tumor phylogenies which is able to effectively manage cases where multiple mutations can occur at the same time and mutations can occur multiple times. Our measure is based on a generalization of the notion of *rooted triples similarity* for classical phylogenies to tumor phylogenies, modelled as multi-labeled trees (that is, where each node is labeled by a set of labels) with poly-occurring labels (that is, each label can be assigned to more than one node). The latter feature is needed because recent studies [239, 82] suggest widespread recurrence and loss of mutations, and more and more methods designed to infer tumor phylogenies considering such a possibility are starting to appear [135, 102, 101].

A comparison of `MP3` with other measures shows that it is able to classify correctly similar and dissimilar trees, both on simulated and on real data.

1.3.6 Incomplete Directed Perfect Phylogeny in Linear Time

Reconstructing the evolutionary history of a set of species is a central task in computational biology. In real data, it is often the case that some information is missing: the Incomplete Directed Perfect Phylogeny (IDPP) problem asks, given a collection of species described by a set of binary characters with some unknown states, to complete the missing states in such a way that the result can be explained with a perfect directed phylogeny. Pe'er et al. [293] proposed a solution that takes $\tilde{\mathcal{O}}(nm)$ time for n species and m characters. Their algorithm relies on pre-existing dynamic connectivity data structures: taking a closer look, we see that it operates on bipartite graphs and only needs to deactivate nodes on one of the sides.

While it seems plausible that some of the known dynamic connectivity structures are actually asymptotically more efficient on such instances, all of them are very complex, and this is not clear. Furthermore, recently Fernández-Baca and Liu [142] performed an experimental study of the algorithm of Pe'er et al. for IDPP [293] with the aim of assessing the impact of the underlying dynamic graph connectivity data structure on their solution. Specifically, they tested the use of the data structure of Holm et al. [195] against a simplified version of the same method, and showed that, in this context, simple data structures perform better than more sophisticated ones with better asymptotic bounds.

We are thus motivated to look for simple, ad-hoc methods that make use of the properties of the decremental connectivity as used in IDPP, so as to avoid the use of sophisticated data structures as a blackbox. We start by describing a simple data structure that dynamically maintains the connected components of a bipartite graph with N nodes on each side, whilst vertices are removed from one side of the graph. The starting point for our solution is an application of a version of the sparsification technique of Eppstein et al. [136]: we define a hierarchical decomposition of the graph, and maintain a forest representing the connected components of each subgraph in this decomposition. This allows us to obtain an extremely simple data structure with $\mathcal{O}(N^2 \log N)$ total update time, which we show to imply an $\mathcal{O}(nm \log n)$ algorithm for IDPP.

The main technical part of our paper refines this solution to shave the logarithmic factor and thus obtain an algorithm that runs in $\mathcal{O}(nm)$ time, which is asymptotically optimal under the natural assumption that the input is given as a matrix [183].

1.4 List of Publications

I have contributed to the following publications during the three years of doctoral studies. However, only the ones marked with ★ constitute the basis of this dissertation. For each of the rest of the publications, the reason for its exclusion is that its subject-area is not closely related to the theme of this dissertation.

The papers marked with ■ form the other main strand of research I followed during my PhD: combinatorial approaches to privacy-preserving data dissemination. Even if such publications were excluded from the core of the thesis because of their field of application, they are fully consistent with my training career, which revolves around combinatorial algorithms on strings and tress. For completeness, these works are included in Appendices C, D and E.

Journal Publications

1. ■ **G. Bernardini**, A. Conte, G. Gourdel, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Punzi, L. Stougie, M. Sweering. Combinatorial Algorithms for String Sanitization. In: *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2020.

2. ★ S. Ciccolella, **G. Bernardini**, L. Denti, P. Bonizzoni, M. Previtali, G. Della Vedova. Triplet-based similarity score for fully multi-labeled trees with poly-occurring labels. In *Bioinformatics*, 2020.

3. ★ M. Alzamel, L. Ayad, **G. Bernardini**, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, G. Rosone. Comparing Degenerate Strings. In *Fundamenta Informaticae*, 2020.

4. L. Ayad, **G. Bernardini**, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, G. Rosone. Longest Property-Preserved Common Factor: a New String-Processing Framework. In *Theoretical Computer Science (TCS)*, 2020.

5. ★ **G. Bernardini**, N. Pisanti, S. P. Pissis, G. Rosone. Approximate Pattern Matching on Elastic-Degenerate Text. In *Theoretical Computer Science (TCS)*, 2020.

6. L. Denti, M. Previtali, **G. Bernardini**, A. Schönhuth, P. Bonizzoni. MALVA: genotyping by Mapping-free ALlele detection of known VAriants. In *iScience*, 2019.

Conference Publications

1. ■ **G. Bernardini**, A. Conte, G. Gourdel, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Punzi, L. Stougie, M. Sweering. Hide and Mine in Strings: Hardness and Algorithms. In *20th IEEE International Conference on Data Mining (ICDM 2020)*.

2. ★ **G. Bernardini**, P. Bonizzoni, P. Gawrychowski. On Two Measures of Distance between Fully-Labelled Trees. In *31th Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*.

3. ■ **G. Bernardini**, H. Chen, G. Loukides, N. Pisanti, S. P. Pissis, L. Stougie, M. Sweering. String Sanitization under Edit Distance. In *31th Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*.

4. ■ **G. Bernardini**, H. Chen, G. Fici, G. Loukides, S. P. Pissis. Reverse-Safe Data Structures for Text Indexing. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX 2020)*.

5. ■ **G. Bernardini**, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Rosone. String Sanitization: A Combinatorial Approach. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD 2019)*.

6. ★ **G. Bernardini**, P. Gawrychowski, N. Pisanti, S. P. Pissis, G. Rosone. Even Faster Elastic-Degenerate String Matching via Fast Matrix Multiplication. In *46th International Colloquium on Automata, Languages and Programming (ICALP 2019)*.

7. ★ **G. Bernardini**, P. Bonizzoni, G. Della Vedova, M. Patterson. A rearrangement distance for fully-labelled trees. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*.

8. L. Ayad, **G. Bernardini**, R. Grossi, C. S. Iliopoulos, N. Pisanti, S.P. Pissis, G. Rosone. Longest Property-Preserved Common Factor. In *String Processing and Information Retrieval (SPIRE 2018)*.

9. ★ M. Alzamel, L. Ayad, **G. Bernardini**, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, G. Rosone. Degenerate String Comparison and Applications. In *Conference on Algorithms in Bioinformatics (WABI 2018)*.

10. ★ **G. Bernardini**, N. Pisanti, S.P. Pissis, G. Rosone. Pattern Matching on Elastic-Degenerate Text with Errors. In *String Processing and Information Retrieval (SPIRE 2017)*.

Submitted

1. ★ **G. Bernardini**, P. Gawrychowski, N. Pisanti, S. P. Pissis, G. Rosone. Elastic-Degenerate String Matching via Fast Matrix Multiplication. Submitted to *SIAM Journal on Computing (SICOMP), 2020*.

2. ■ **G. Bernardini**, H. Chen, G. Fici, G. Loukides, S. P. Pissis. Reverse-Safe Data Structures for Text Indexing. Submitted to *ACM Journal of Experimental Algorithmics (JEA), 2020*

Work in progress

1. ■ **G. Bernardini**, A. Conte, G. Gourdel, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Punzi, L. Stougie, M. Sweering. Hide and Mine in Strings: Hardness and Algorithms. To be submitted to *IEEE Transactions on Knowledge and Data Engineering (TKDE), 2020*

2. **G. Bernardini**, G. Fici, P. Gawrychowski, S. P. Pissis. Substring Complexity in Sublinear Space.

3. ★ **G. Bernardini**, P. Gawrychowski, P. Bonizzoni. Incomplete Directed Perfect Phylogeny in Linear Time.

List of Figures

List of Algorithms

Part I

Degenerate Strings

Chapter 2

Elastic-Degenerate String Matching via Matrix Multiplication

Key Points

Problem. Traditionally, a single annotated reference genome is used as representative example of the genomic sequence of a species. It serves as a reference text to which, for example, fragments of newly sequenced genomes of individuals are mapped. Although a single reference genome provides a good approximation of any individual genome, in loci with polymorphic variations mapping and sequence comparison often fail their purposes. This is where a multiple genome, i.e., a pan-genome, would be a better reference text. The computational task we consider here is to map a sequence in a collection of similar strings (e.g., a pan-genome) on-line.

Model. Elastic-degenerate (ED) strings have been introduced to represent a multiple alignment of several closely-related sequences compactly. In this representation, substrings of these sequences that match exactly are collapsed, while in positions where the sequences differ, all possible variants observed at that location are listed. We assume that an ED text is read degenerate position by degenerate position, and we search for occurrences of a deterministic pattern on-line.

Included Works

This chapter presents the paper **Elastic-Degenerate String Matching via Fast Matrix Multiplication**, which is currently submitted to the *SIAM Journal on Computing (SICOMP)*. This paper is a journal extension of **Even Faster Elastic-Degenerate String Matching via Fast Matrix Multiplication** [58], which I presented at the

46th International Colloquium on Automata, Languages and Programming (ICALP 2019).

2.1 Introduction

Boolean matrix multiplication (BMM) is one of the most fundamental computational problems. Apart from its theoretical interest, it has a wide range of applications [210, 350, 157, 150, 277]. BMM is also the core combinatorial part of integer matrix multiplication. In both problems, we are given two $\mathcal{N}\times\mathcal{N}$ matrices and we are to compute $\mathcal{N}^2$ values. Integer matrix multiplication can be performed in *truly subcubic* time, i.e., in $\mathcal{O}(\mathcal{N}^{3-\epsilon})$ operations over the field, for some $\epsilon>0$. The fastest known algorithms for this problem run in $\mathcal{O}(\mathcal{N}^{2.373})$ time [242, 362]. These algorithms are known as algebraic: they rely on the ring structure of matrices over the field.

There also exists a different family of algorithms for the BMM problem known as combinatorial. Their focus is on unveiling the combinatorial structure in the Boolean matrices to reduce redundant computations. A series of results [34, 42, 91] culminating in an $\hat{\mathcal{O}}(\mathcal{N}^3/\log^4\mathcal{N})$-time algorithm [372, 371] (the $\hat{\mathcal{O}}(\cdot)$ notation suppresses polyloglog factors) has led to the popular combinatorial BMM conjecture stating that there is no combinatorial algorithm for BMM working in time $\mathcal{O}(\mathcal{N}^{3-\epsilon})$, for any $\epsilon>0$ [10]. There has been ample work on applying this conjecture to obtain BMM hardness results: see , e.g., [243, 10, 311, 192, 241, 236, 96].

String matching is another fundamental problem, asking to find all fragments of a string text of length n that match a string pattern of length m. This problem has several linear-time solutions [113]. In many real-world applications, it is often the case that letters at some positions are either unknown or uncertain. A way of representing these positions is with a subset of the alphabet Σ. Such a representation is called *degenerate string*. A special case of a degenerate string is when at such unknown or uncertain positions the only subset of the alphabet allowed is the whole alphabet. These special degenerate strings are more commonly known as strings with wildcards. The first efficient algorithm for a text and a pattern, where both may contain wildcards, was published by Fischer and Paterson in 1974 [151]. It has undergone several improvements since then [209, 219, 107, 105]. The first efficient algorithm for a standard text and a degenerate pattern, which may contain any non-empty subset of the alphabet, was published by Abrahamson in 1987 [11], followed by several practically efficient algorithms [364, 286, 196].

Degenerate letters are used in the IUPAC notation [211] to represent a position in a DNA sequence that can have multiple possible alternatives. These are used to encode the consensus of a population of sequences [108, 27, 165] in a multiple sequence alignment (MSA). In the presence of insertions or deletions in the MSA, we may need to consider alternative representations. Consider the following MSA of three closely-related sequences:

```
GCAACGGGTA--TT
GCAACGGGTATATT
GCACCTGG----TT
```

These sequences can be compacted into a single sequence $\tilde{T}$ of sets of strings, containing some deterministic and some non-deterministic segments:

$$\tilde{T} = \{\ \texttt{GCA}\ \} \cdot \left\{ \begin{array}{c} \texttt{A} \\ \texttt{C} \end{array} \right\} \cdot \{\ \texttt{C}\ \} \cdot \left\{ \begin{array}{c} \texttt{G} \\ \texttt{T} \end{array} \right\} \cdot \{\ \texttt{GG}\ \} \cdot \left\{ \begin{array}{c} \texttt{TA} \\ \texttt{TATA} \\ \varepsilon \end{array} \right\} \cdot \{\ \texttt{TT}\ \}$$

A non-deterministic segment is a finite set of deterministic strings and may contain the empty string ε corresponding to a deletion. The total number of segments is the *length* of $\tilde{T}$ and the total number of letters is the *size* of $\tilde{T}$. We denote the length by $n = |\tilde{T}|$ and the size by $N = ||\tilde{T}||$.

This representation has been defined in [205] by Iliopoulos et al. as an *elastic-degenerate* (ED) string. Being a sequence of subsets of Σ^*, it can be seen as a generalization of a degenerate string. The natural problem that arises is finding all matches of a deterministic pattern P in an ED text $\tilde{T}$. This is the *elastic-degenerate string matching* (EDSM) problem. Since its introduction in 2017 [205], it has attracted some attention in the combinatorial pattern matching community, and a series of results have been published. The simple algorithm by Iliopoulos et al. [205] for EDSM was first improved by Grossi et al. in the same year, who showed that, for a pattern of length m, the EDSM problem can be solved *on-line* in $\mathcal{O}(nm^2 + N)$ time [178]; on-line means that it reads the text segment-by-segment and reports an occurrence as soon as this is detected. This result was improved by Aoyama et al. [31] who presented an $\mathcal{O}(nm^{1.5}\sqrt{\log m} + N)$-time algorithm. An important feature of these bounds is their *linear dependency* on N. A different branch of on-line algorithms waiving the linear-dependency restriction exists [178, 296, 104, 103]. Recent results on founder block graphs [260] can also be casted on elastic-degenerate strings.

A question with a somewhat similar flavor is the *word break* problem. We are given a dictionary $\mathcal{D}$, $m = ||\mathcal{D}||$, and a string S, $n = |S|$, and the question is whether we can split S into fragments that appear in $\mathcal{D}$ (the same element of $\mathcal{D}$ can be used multiple times). Backurs and Indyk [39] designed an $\tilde{\mathcal{O}}(nm^{1/2-1/18} + m)$-time algorithm for this problem[1]. Bringmann et al. [79] improved this to $\tilde{\mathcal{O}}(nm^{1/3} + m)$ and showed that this is optimal for combinatorial algorithms by a reduction from k-Clique. Their algorithm uses fast Fourier transform (FFT), and so it is not clear whether it should be considered combinatorial. While this problem seems similar to EDSM, there does not seem to be a direct reduction and so their lower bound does not immediately apply.

Our Results. It is known that BMM and triangle detection (TD) in graphs either both have truly subcubic combinatorial algorithms or none of them do [361]. Recall also that the currently fastest algorithm with linear dependency on N for the EDSM problem runs in $\mathcal{O}(nm^{1.5}\sqrt{\log m} + N)$ time [31]. In this chapter we prove the following two theorems.

Theorem 1. *If the EDSM problem can be solved in $\mathcal{O}(nm^{1.5-\epsilon} + N)$ time, for any $\epsilon > 0$, with a combinatorial algorithm, then there exists a truly subcubic combinatorial algorithm for TD.*

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses polylog factors.

Arguably, the notion of combinatorial algorithms is not clearly defined, and Theorem 1 should be understood as an indication that in order to achieve a better complexity one should use fast matrix multiplication. Indeed, there are examples where a lower bound conditioned on BMM was helpful in constructing efficient algorithms using fast matrix multiplication [7, 92, 78, 269, 122, 360, 377]. We successfully design such a non-combinatorial algorithm by combining three ingredients: a string periodicity argument, FFT, and fast matrix multiplication. While periodicity is the usual tool in combinatorial pattern matching [231, 120, 235] and using FFT is also not unusual (for example, it often shows up in approximate string matching [11, 30, 105, 167]), to the best of our knowledge, we are the first to combine these with fast matrix multiplication. Specifically, we show the following result for the EDSM problem, where ω denotes the matrix multiplication exponent.

Theorem 2. *The EDSM problem can be solved on-line in $\tilde{\mathcal{O}}(nm^{\omega-1}+N)$ time.*

In order to obtain a faster algorithm for the EDSM problem, we focus on the *active prefixes* (AP) problem that lies at the heart of all current solutions [178, 31]. In the AP problem, we are given a string P of length m and a set of arbitrary prefixes $P[1\,..\,i]$ of P, called *active prefixes*, stored in a bit vector U so that $U[i]=1$ if $P[1\,..\,i]$ is active. We are further given a set $\mathcal{S}$ of strings of total length N and we are asked to compute a bit vector V which stores the new set of active prefixes of P. A new active prefix of P is a concatenation of $P[1\,..\,i]$ (such that $U[i]=1$) and some element of $\mathcal{S}$.

Using the algorithmic framework introduced in [178], EDSM is addressed by solving an instance of the AP problem per each segment i of the ED text corresponding to set $\mathcal{S}$ of the AP problem. Hence, an $\mathcal{O}(f(m)+N_i)$ solution for the AP problem (with N_i being the size of a single segment of the ED text) implies an $\mathcal{O}(nf(m)+N)$ solution of EDSM, as $f(m)$ is repeated n times and $N=\sum_{i=1}^{n}N_i$. The algorithm of [31] solves the AP problem in $\mathcal{O}(m^{1.5}\sqrt{\log m}+N_i)$ time leading to $\mathcal{O}(nm^{1.5}\sqrt{\log m}+N)$ time for the EDSM problem. Our algorithm partitions the strings of each segment i of the ED text into three types according to a periodicity criterion, and then solves a restricted instance of the AP problem for each of the types. In particular, we solve the AP problem in $\tilde{\mathcal{O}}(m^{\omega-1}+N_i)$ time leading to $\tilde{\mathcal{O}}(nm^{\omega-1}+N)$ time for the EDSM problem. Given this connection between the two problems and, in particular, between their size parameter N, in the rest of the chapter we will denote with N also the parameter N_i of the AP problem.

An important building block in our solution that might find applications in other problems is a method of selecting a small set of length-ℓ substrings of the pattern, called *anchors*, so that any relevant occurrence of a string from an ED text set contains at least one but not too many such anchors inside. This is obtained by rephrasing the question in a graph-theoretical language and then generalizing the well-known fact that an instance of the hitting set problem with m sets over $[n]$, each of size at least k, has a solution of size $\mathcal{O}(n/k\cdot\log m)$. While the idea of carefully selecting some substrings of the same length is not new (for example Kociumaka et al. [235] used it to design a data structure for pattern matching queries on a string), our setting is different and hence so is the method of selecting these substrings.

In addition to the conditional lower bound for the EDSM problem (Theorem 1) we show here the following conditional lower bound for the AP problem.

Theorem 3. *If the AP problem can be solved in $\mathcal{O}(m^{1.5-\epsilon}+N)$ time, for any $\epsilon>0$, with a combinatorial algorithm, then there exists a truly subcubic combinatorial algorithm for the BMM problem.*

Roadmap. Section 2.2 provides the necessary definitions and notation as well as the algorithmic toolbox used throughout the chapter. In Section 2.3 we prove our lower bound result for the AP problem (Theorem 3). The lower bound result for the EDSM problem is proved in Section 2.4 (Theorem 1). In Section 2.5 we present our algorithm for EDSM (Theorem 2); this is the most technically involved part of the chapter.

2.2 Preliminaries

Recall that a period of a string X is any integer $p \in [1, |X|]$ such that $X[i] = X[i+p]$ for every $i = 1, 2, \ldots, |X| - p$, and *the period*, denoted by $\mathrm{per}(X)$, is the smallest such p. We call a string X *strongly periodic* if $\mathrm{per}(X) \leq |X|/4$. We start by reporting a well-known result of which we will make extensive use.

Lemma 1 ([147])**.** *If p and q are both periods of the same string X, and additionally $p+q \leq |X|+1$, then $\gcd(p,q)$ is also a period of X.*

A *heavy path decomposition* of a tree T is obtained by selecting, for every non-leaf node $u \in T$, its child v such that the subtree rooted at v is the largest. This decomposes the nodes of T into node-disjoint paths, with each such path p (called a heavy path) starting at some node, called the *head* of p, and ending at a leaf. An important property of such a decomposition is that the number of distinct heavy paths above any leaf (that is, intersecting the path from a leaf to the root) is only logarithmic in the size of T [328].

Let $\tilde{\Sigma}$ denote the set of all finite non-empty subsets of Σ^*. Previous works (cf. [205, 178, 31, 296, 60]) define $\tilde{\Sigma}$ as the set of all finite non-empty subsets of Σ^* excluding $\{\varepsilon\}$ but we waive here the latter restriction as it has no algorithmic implications. An *elastic-degenerate string* $\tilde{T} = \tilde{T}[1]\ldots\tilde{T}[n]$, or ED string, over alphabet Σ, is a string over $\tilde{\Sigma}$, i.e., an ED string is an element of $\tilde{\Sigma}^*$, and hence each $\tilde{T}[i]$ is a set of strings.

Let $\tilde{T}$ denote an ED string of length n, i.e. $|\tilde{T}| = n$. We assume that for any $1 \leq i \leq n$, the set $\tilde{T}[i] \in \tilde{\Sigma}$ is implemented as an array and can be accessed by an index, i.e., $\tilde{T}[i] = \{\tilde{T}[i][k] \mid k = 1, \ldots, |\tilde{T}[i]|\}$. For any $\tilde{\sigma} \in \tilde{\Sigma}$, $||\tilde{\sigma}||$ denotes the total length of all strings in $\tilde{\sigma}$, and for any ED string $\tilde{T}$, $||\tilde{T}||$ denotes the total length of all strings in all $\tilde{T}[i]$s. We will denote $N_i = \sum_{k=1}^{|\tilde{T}[i]|} |\tilde{T}[i][k]|$ the total length of all strings in $\tilde{T}[i]$ and $N = \sum_{i=1}^{n} ||\tilde{T}[i]||$ the *size* of $\tilde{T}$. An ED string $\tilde{T}$ can be thought of as a compact representation of the set of strings $\mathcal{A}(\tilde{T})$ which is the Cartesian product of all $\tilde{T}[i]$s; that is, $\mathcal{A}(\tilde{T}) = \tilde{T}[1] \times \ldots \times \tilde{T}[n]$ where $A \times B = \{xy \mid x \in A, y \in B\}$ for any sets of strings A and B.

For any ED string $\tilde{X}$ and a pattern P, we say that P *matches* $\tilde{X}$ if:

1. $|\tilde{X}| = 1$ and P is a substring of some string in $\tilde{X}[1]$, or,
2. $|\tilde{X}| > 1$ and $P = P_1 \ldots P_{|\tilde{X}|}$, where P_1 is a suffix of some string in $\tilde{X}[1]$, $P_{|\tilde{X}|}$ is a prefix of some string in $\tilde{X}[|\tilde{X}|]$, and $P_i \in \tilde{X}[i]$, for all $1 < i < |\tilde{X}|$.

We say that an occurrence of a string P ends at position j of an ED string $\tilde{T}$ if there exists $i \leq j$ such that P matches $\tilde{T}[i] \dots \tilde{T}[j]$. We will refer to string P as the *pattern* and to ED string $\tilde{T}$ as the *text*. We define the main problem considered in this chapter.

Elastic-Degenerate String Matching (EDSM)
INPUT: A string P of length m and an ED string $\tilde{T}$ of length n and size $N \geq m$.
OUTPUT: All positions in $\tilde{T}$ where at least one occurrence of P ends.

Example 1. *Pattern* $P = \texttt{GTAT}$ *ends at positions* 2, 6, *and* 7 *of the following text* $\tilde{T}$.

$$\tilde{T} = \{\ \texttt{ATGTA}\ \} \cdot \left\{ \begin{array}{c} \texttt{A} \\ \texttt{T} \end{array} \right\} \cdot \{\ \texttt{C}\ \} \cdot \left\{ \begin{array}{c} \texttt{G} \\ \texttt{T} \end{array} \right\} \cdot \{\ \texttt{CG}\ \} \cdot \left\{ \begin{array}{c} \texttt{TA} \\ \texttt{TATA} \\ \varepsilon \end{array} \right\} \cdot \left\{ \begin{array}{c} \texttt{TATGC} \\ \texttt{TTTTA} \end{array} \right\}$$

Aoyama et al. [31] obtained an on-line $\mathcal{O}(nm^{1.5}\sqrt{\log m} + N)$-time algorithm by designing an efficient solution for the following problem.

Active Prefixes (AP)
INPUT: A string P of length m, a bit vector U of size m, 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 $i \in [1, m], U[i] = 1$, such that $P[1\,..\,i] \cdot S = P[1\,..\,i + \lvert S\rvert]$ and $j = i + \lvert S\rvert$.

In more detail, given an ED text $\tilde{T} = \tilde{T}[1] \dots \tilde{T}[n]$, one should consider an instance of the AP problem per each $\tilde{T}[i]$. Hence, an $\mathcal{O}(f(m) + N_i)$ solution for AP (N_i being the size of $\tilde{T}[i]$) implies an $\mathcal{O}(n \cdot f(m) + N)$ solution for EDSM, as $f(m)$ is repeated n times and $N = \sum_{i=1}^{n} N_i$. We provide an example of the AP problem.

Example 2. *Let* $P = \texttt{ababbababab}$ *of length* $m = 11$, $U = 01000100000$, *and* $\mathcal{S} = \{\varepsilon, \texttt{ab}, \texttt{abb}, \texttt{ba}, \texttt{baba}\}$. *We have that* $V = 01011101010$.

For our lower bound results we rely on BMM and the following closely related problem.

Boolean Matrix Multiplication (BMM)
INPUT: Two $\mathcal{N} \times \mathcal{N}$ Boolean matrices A and B.
OUTPUT: $\mathcal{N} \times \mathcal{N}$ Boolean matrix C, where $C[i, j] = \bigvee_k (A[i, k] \wedge B[k, j])$.

Triangle Detection (TD)
INPUT: Three $\mathcal{N} \times \mathcal{N}$ Boolean matrices A, B and C.
OUTPUT: Are there i, j, k such that $A[i, j] = B[j, k] = C[k, i] = 1$?

An algorithm is called *truly subcubic* if it runs in $\mathcal{O}(\mathcal{N}^{3-\epsilon})$ time, for some $\epsilon > 0$. TD and BMM either both have truly subcubic combinatorial algorithms, or none of them do [361].

2.3 AP Conditional Lower Bound

To investigate the hardness of the EDSM problem, we first show that an $\mathcal{O}(m^{1.5-\epsilon}+N)$-time solution to the active prefixes problem, that constitutes the core of the solutions proposed in [178, 31], would imply a truly subcubic combinatorial algorithm for Boolean matrix multiplication (BMM). We recall that in the AP problem we are given a string P of length m and a set of prefixes $P[1\,..\,i]$ of P, called *active prefixes*, stored in a bit vector U ($U[i]=1$ if and only if $P[1\,..\,i]$ is active). We are further given a set $\mathcal{S}$ of strings of total length N and we are asked to compute a bit vector V storing the new set of active prefixes of P: a prefix of P that extends $P[1\,..\,i]$ (such that $U[i]=1$) with some element of $\mathcal{S}$. Of course, we can solve BMM by working over integers and using one of the fast matrix multiplication algorithms; plugging in the best known bounds results in an $\mathcal{O}(\mathcal{N}^{2.373})$-time algorithm [242, 362]. However, such an algorithm is not *combinatorial*, i.e., it uses *algebraic* methods. In comparison, the best known combinatorial algorithm for BMM works in $\mathcal{O}(\mathcal{N}^3/\log^4\mathcal{N})$ time [372]. This leads to the following popular conjecture.

Conjecture 1 ([10])**.** *There is no combinatorial algorithm for the BMM problem working in time $\mathcal{O}(\mathcal{N}^{3-\epsilon})$, for any $\epsilon>0$.*

Aoyama et al. [31] showed that the AP problem can be solved in $\mathcal{O}(m^{1.5}\sqrt{\log m}+N)$ time for constant-sized alphabets. Together with some standard string-processing techniques applied similarly as in [178], this is then used to solve the EDSM problem by creating an instance of the AP problem for every set $\tilde{T}[i]$ of $\tilde{T}$, i.e., with $\mathcal{S}=\tilde{T}[i]$.

We argue that, unless Conjecture 1 is false, the AP problem cannot be solved faster than $\mathcal{O}(m^{1.5-\epsilon}+N)$, for any $\epsilon>0$, with a combinatorial algorithm (note that the algorithm of Aoyama et al. [31] uses FFT, and so it is not completely clear whether it should be considered to be combinatorial). We show this by a reduction from combinatorial BMM. Assume that, for the AP problem, we seek combinatorial algorithms with the running time $\mathcal{O}(m^{1.5-\epsilon}+N)$, i.e., with linear dependency on the total length of the strings. We need to show that such an algorithm implies that the BMM problem can be solved in $\mathcal{O}(\mathcal{N}^{3-\epsilon'})$ time, for some $\epsilon'>0$, with a combinatorial algorithm, thus implying that Conjecture 1 is false.

Theorem 3. *If the AP problem can be solved in $\mathcal{O}(m^{1.5-\epsilon}+N)$ time, for any $\epsilon>0$, with a combinatorial algorithm, then there exists a truly subcubic combinatorial algorithm for the BMM problem.*

Proof. Recall that in the BMM problem the matrices are denoted by A and B. In order to compute $C=A\times B$, we need to find, for every $i,j=1,\ldots,\mathcal{N}$, an index k such that $A[i,k]=1$ and $B[k,j]=1$. To this purpose, we split matrix A into blocks of size $\mathcal{N}\cdot L$ and B into blocks of size $L\cdot L$. This corresponds to considering values of j and k in intervals of size L, and clearly there are $\mathcal{N}/L$ such intervals. Matrix B is thus split into $(\mathcal{N}/L)^2$ blocks, giving rise to an equal number of instances of the AP problem, each one corresponding to an interval of j and an interval of k. This creates $(\mathcal{N}/L)^2$ blocks in matrix B; we will thus create $(\mathcal{N}/L)^2$ separate instances of the AP problem corresponding to an interval of j and an interval of k. We will now describe the instance corresponding to the (K,J)-th block, where $1\leq K,J\leq\mathcal{N}/L$.

We build the string P of the AP problem, for any block, as a concatenation of $\mathcal{N}$ gadgets corresponding to $i = 1, \ldots, \mathcal{N}$, and we construct the bit vector $U^{(K,J)}$ of the AP problem as a concatenation of $\mathcal{N}$ bit vectors, one per gadget. Each gadget consists of the same string $\mathtt{a}^L\mathtt{ba}^L$; we set to 1 the k'-th bit of the i-th gadget bit vector if $A[i, (K-1)L + k'] = 1$. The solution of the AP problem $V^{(K,J)}$ will allow us to recover the solution of BMM, as we will ensure that the bit corresponding to the j'-th $\mathtt{a}$ in the second half of the gadget is set to 1 if and only if, for some $k' \in [L]$, $A[i, (K-1)L + k'] = 1$ and $B[(K-1)L + k', (J-1)L + j'] = 1$. In order to enforce this, we will include the following strings in set $\mathcal{S}^{(K,J)}$:

$$\mathtt{a}^{L-k'}\mathtt{ba}^{j'}, \text{ for every } k', j' \in [L] \text{ such that } B[(K-1)L + k', (J-1)L + j'] = 1.$$

This guarantees that after solving the AP problem we have the required property, and thus, after solving all the instances, we have obtained matrix $C = A \times B$. Indeed, consider values j, i.e., the index that runs on the columns of C, in intervals of size L. By construction and by definition of BMM, the i-th line of the J-th column interval of C is obtained by taking the disjunction of the second half of the i-th interval of each (K, J)-th bit vector for every $K = 1, 2, \ldots, \mathcal{N}/L$.

We have a total of $(\mathcal{N}/L)^2$ instances. In each of them, the total length of all strings is $\mathcal{O}(L^3)$, and the length of the input string P is $(2L+1)\mathcal{N} = \mathcal{O}(L \cdot \mathcal{N})$. Using our assumed algorithm for each instance, we obtain the following total time:

$$\mathcal{O}((\mathcal{N}/L)^2 \cdot (L^3 + (\mathcal{N} \cdot L)^{1.5-\epsilon})) = \mathcal{O}(\mathcal{N}^2 \cdot L + \mathcal{N}^{3.5-\epsilon}/L^{0.5+\epsilon}).$$

If we set $L = \mathcal{N}^{(1.5-\epsilon)/(1.5+\epsilon)}$, then the total time becomes:

$$\begin{aligned} & \mathcal{O}(\mathcal{N}^{2+(1.5-\epsilon)/(1.5+\epsilon)} + \mathcal{N}^{3.5-\epsilon-(0.5+\epsilon)(1.5-\epsilon)/(1.5+\epsilon)}) \\ = \; & \mathcal{O}(\mathcal{N}^{2+(1.5-\epsilon)/(1.5+\epsilon)} + \mathcal{N}^{2+(1.5-\epsilon)-(1.5-\epsilon)(0.5+\epsilon)/(1.5+\epsilon)}) \\ = \; & \mathcal{O}(\mathcal{N}^{2+(1.5-\epsilon)/(1.5+\epsilon)} + \mathcal{N}^{2+(1.5-\epsilon)(1.5+\epsilon-0.5-\epsilon)/(1.5+\epsilon)}) \\ = \; & \mathcal{O}(\mathcal{N}^{2+(1.5-\epsilon)/(1.5+\epsilon)}). \end{aligned}$$

Hence we obtain a combinatorial BMM algorithm with complexity $\mathcal{O}(\mathcal{N}^{3-\epsilon'})$, where $\epsilon' = 1 - (1.5-\epsilon)/(1.5+\epsilon) > 0$. □

Example 3. *Consider the following instance of the BMM problem with* $\mathcal{N} = 6$ *and* $L = 3$.

$$A = \left[\begin{array}{ccc|ccc} 0&1&0&0&1&0\\ 1&0&1&0&0&0\\ 0&0&0&0&0&1\\ \hline 1&0&0&0&1&0\\ 0&0&0&1&0&0\\ 0&1&0&0&0&0 \end{array}\right] \quad B = \left[\begin{array}{ccc|ccc} 0&0&0&0&0&1\\ 1&0&0&0&0&0\\ 0&0&1&0&1&0\\ \hline 0&1&0&0&0&0\\ 0&0&0&1&0&0\\ 1&0&0&0&1&0 \end{array}\right]$$

$$C = \left[\begin{array}{ccc|ccc} \mathbf{1}&\mathbf{0}&\mathbf{0}&1&0&0\\ 0&0&1&0&1&1\\ 1&0&0&0&1&0\\ \hline 0&0&0&1&0&1\\ 0&1&0&0&0&0\\ 1&0&0&0&0&0 \end{array}\right]$$

From matrices A and B, we now show how the resulting matrix C can be found by building and solving 4 instances of the AP problem constructed as follows. The pattern is

$$P = \texttt{aaabaaa} \cdot \texttt{aaabaaa} \cdot \texttt{aaabaaa} \cdot \texttt{aaabaaa} \cdot \texttt{aaabaaa} \cdot \texttt{aaabaaa}$$

where the six gadgets are separated by a $'\cdot'$ to be highlighted. For the AP instances, the vectors $U^{(K,J)}$ shown below are the input bit vectors, and the sets $S^{(K,J)}$ are the input set of strings.

For each instance, the bit vector $V^{(K,J)}$ shown below is the output of the AP problem.

i 1 2 3 4 5 6

$U^{(1,1)}$: [0 1 0 0 0 0 0 | 1 0 1 0 0 0 0 | 0 0 0 0 0 0 0 | 1 0 0 0 0 0 0 | 0 0 0 0 0 0 0 | 0 1 0 0 0 0 0]

$S^{(1,1)}$: {aba,baaa}

$V^{(1,1)}$: [0 0 0 0 **1 0 0** | 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 | 0 0 0 0 1 0 0]

$U^{(1,2)}$: [0 1 0 0 0 0 0 | 1 0 1 0 0 0 0 | 0 0 0 0 0 0 0 | 1 0 0 0 0 0 0 | 0 0 0 0 0 0 0 | 0 1 0 0 0 0 0]

$S^{(1,2)}$: {aabaaa,baa}

$V^{(1,2)}$: [0 0 0 0 0 0 0 | 0 0 0 0 0 1 1 | 0 0 0 0 0 0 0 | 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0]

$U^{(2,1)}$: [0 1 0 0 0 0 0 | 0 0 0 0 0 0 0 | 0 0 1 0 0 0 0 | 0 1 0 0 0 0 0 | 1 0 0 0 0 0 0 | 0 0 0 0 0 0 0]

$S^{(2,1)}$: {aabaa,ba}

$V^{(2,1)}$: [0 0 0 0 **0 0 0** | 0 0 0 0 0 0 0 | 0 0 0 0 1 0 0 | 0 0 0 0 0 0 0 | 0 0 0 0 0 1 0 | 0 0 0 0 0 0 0]

$U^{(2,2)}$: [0 1 0 0 0 0 0 | 0 0 0 0 0 0 0 | 0 0 1 0 0 0 0 | 0 1 0 0 0 0 0 | 1 0 0 0 0 0 0 | 0 0 0 0 0 0 0]

$S^{(2,2)}$: {aba,baa}

$V^{(2,2)}$: [0 0 0 0 1 0 0 | 0 0 0 0 0 0 0 | 0 0 0 0 0 1 0 | 0 0 0 0 1 0 0 | 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0]

*As an example on how to obtain matrix C, consider the bold part of C above (*i.e., *the first line of block $(1,1)$ of C). This is obtained by taking the disjunction of the bold parts of $V^{(1,1)}$ and $V^{(2,1)}$.*

2.4 EDSM Conditional Lower Bound

Since the lower bound for the AP problem does not imply *per se* a lower bound for the whole EDSM problem, in this section we show a conditional lower bound for the EDSM problem. Specifically, we perform a reduction from Triangle Detection to show that, if the EDSM problem could be solved in $\mathcal{O}(nm^{1.5-\epsilon} + N)$ time, this would imply the existence of a truly subcubic algorithm for TD. We show that TD can be reduced to the decision version of the EDSM problem: the goal is to detect whether there exists at least one occurrence of P in $\tilde{T}$. To this aim, given three matrices A, B, C, we

first decompose matrix B into blocks of size $\mathcal{N}/s \times \mathcal{N}/s$, where s is a parameter to be determined later; the pattern P is obtained by concatenating a number (namely $z = \mathcal{N}s^2$) of constituent parts P_i of length $\mathcal{O}(\mathcal{N}/s)$, each one built with five letters from disjoint subalphabets. The ED text $\tilde{T}$ is composed of three parts: the central part consists of three degenerate segments, the first one encoding the 1s of matrix A, the second one those of matrix B and the third one those of matrix C. These segments are built in such a way that the concatenation of strings of subsequent segments is of the same form as the pattern's building blocks. This central part is then padded to the left and to the right with sets containing appropriately chosen concatenations of substrings P_i of P, so that an occurrence of the pattern in the text implies that one of its building blocks matches the central part of the text, thus corresponding to a triangle. Formally:

Theorem 1. *If the EDSM problem can be solved in $\mathcal{O}(nm^{1.5-\epsilon} + N)$ time, for any $\epsilon > 0$, with a combinatorial algorithm, then there exists a truly subcubic combinatorial algorithm for TD.*

Proof. Consider an instance of TD, where we are given three $\mathcal{N} \times \mathcal{N}$ Boolean matrices A, B, C, and the question is to check if there exist i, j, k such that $A[i, j] = B[j, k] = C[k, i] = 1$. Let s be a parameter, to be determined later, that corresponds to decomposing B into blocks of size $(\mathcal{N}/s) \times (\mathcal{N}/s)$. We reduce to an instance of EDSM over an alphabet Σ of size $\mathcal{O}(\mathcal{N})$.

Pattern P. We construct P by concatenating, in some fixed order, the following strings:

$$P(i, x, y) = v(i)xa^{\mathcal{N}/s}x\$\$ya^{\mathcal{N}/s}yv(i)$$

for every $i = 1, 2, \ldots, \mathcal{N}$ and $x, y = 1, 2, \ldots, s$, where $a \in \Sigma_1$, $\$ \in \Sigma_2$, $x \in \Sigma_3$, $y \in \Sigma_4$, $v(i) \in \Sigma_5$, and $\Sigma_1, \Sigma_2, \ldots, \Sigma_5$ are disjoint subsets of Σ.

ED text $\tilde{T}$. The text $\tilde{T}$ consists of three parts. Its middle part encodes all the entries equal to 1 in matrices A, B and C, and consists of three string sets $\mathcal{X} = \mathcal{X}_1 \cdot \mathcal{X}_2 \cdot \mathcal{X}_3$, where:

1. $\mathcal{X}_1$ contains all strings of the form $v(i)xa^j$, for some $i \in [\mathcal{N}]$, $x \in [s]$ and $j \in [\mathcal{N}/s]$ such that $A[i, (x-1) \cdot (\mathcal{N}/s) + j] = 1$;
2. $\mathcal{X}_2$ contains all strings of the form $a^{\mathcal{N}/s-j}\ x\$\$ya^{\mathcal{N}/s-k}$, for some $x, y \in [s]$ and $j, k = \in [\mathcal{N}/s]$ such that $B[(x-1) \cdot (\mathcal{N}/s) + j, (y-1) \cdot (\mathcal{N}/s) + k] = 1$, i.e., if the corresponding entry of B is 1;
3. $\mathcal{X}_3$ contains all strings of the form $a^k yv(i)$, for some $i \in [\mathcal{N}]$, $y \in [s]$ and $k \in [\mathcal{N}/s]$ such that $C[(y-1) \cdot (\mathcal{N}/s) + k, i] = 1$.

It is easy to see that $|P(i, x, y)| = \mathcal{O}(\mathcal{N}/s)$. This implies the following:

1. The length of the pattern is $m = \mathcal{O}(\mathcal{N} \cdot s^2 \cdot \mathcal{N}/s) = \mathcal{O}(\mathcal{N}^2 \cdot s)$;
2. The total length of $\mathcal{X}$ is

$$||\mathcal{X}|| = \mathcal{O}(\mathcal{N} \cdot s \cdot \mathcal{N}/s \cdot \mathcal{N}/s + s^2 \cdot (\mathcal{N}/s)^2 \cdot \mathcal{N}/s + \mathcal{N} \cdot s \cdot \mathcal{N}/s \cdot \mathcal{N}/s) = \mathcal{O}(\mathcal{N}^3/s).$$

By the above construction, we obtain the following fact.

Fact 4. *$P(i,x,y)$ matches $\mathcal{X}$ if and only if the following holds for some $j,k = 1,2,\ldots,\mathcal{N}/s$:*

$$A[i,(x-1)\cdot(\mathcal{N}/s)+j] = B[(x-1)\cdot(\mathcal{N}/s)+j,(y-1)\cdot(\mathcal{N}/s)+k] = C[(y-1)\cdot(\mathcal{N}/s)+k,i] = 1$$

Solving the TD problem thus reduces to taking the disjunction of all such conditions. Let us write down all strings $P(i,x,y)$ in some arbitrary but fixed order to obtain $P = P_1P_2\ldots P_z$ with $z = \mathcal{N}s^2$, where every $P_t = P(i,x,y)$, for some i,x,y. We aim to construct a small number of sets of strings that, when considered as an ED text, match any prefix $P_1P_2\ldots P_t$ of the pattern, $1 \le t \le z-1$; a similar construction can be carried on to obtain sets of strings that match any suffix $P_k\ldots P_{z-1}P_z$, $2 \le k \le z$. These sets will then be added to the left and to the right of $\mathcal{X}$, respectively, to obtain the ED text $\tilde{T}$.

ED Prefix. We construct $\log z$ sets of strings as follows. The first one contains the empty string $P_1P_2\ldots P_{z/2}$ and ε. The second one contains $P_{z/2+1}\ldots P_{z/2+z/4}$, $P_1P_2\ldots P_{z/4}$ and ε. The third one contains $P_{z/2+z/4+1}\cdots P_{z/2+z/4+z/8}$, $P_{z/2+1}\cdots P_{z/2+z/8}$, $P_{z/4+1}\ldots P_{z/4+z/8}$, $P_1P_2\ldots P_{z/8}$ and ε. Formally, for every $i = 1,2,\ldots,\log z$, the i-th of such sets is:

$$\tilde{T}_i^p = \varepsilon \cup \{P_{j\frac{z}{2^{i-1}}+1}\ldots P_{j\frac{z}{2^{i-1}}+\frac{z}{2^i}} \mid j = 0,1,\ldots,2^{i-1}-1\}.$$

ED Suffix. We similarly construct $\log z$ sets to be appended to $\mathcal{X}$:

$$\tilde{T}_i^s = \varepsilon \cup \{P_{z-j\frac{z}{2^{i-1}}-\frac{z}{2^i}+1}\ldots P_{z-j\frac{z}{2^{i-1}}} \mid j = 0,1,\ldots,2^{i-1}-1\}.$$

The total length of all the ED prefix and ED suffix strings is $\mathcal{O}(\log z\cdot\mathcal{N}^2\cdot s) = \mathcal{O}(\mathcal{N}^2\cdot s\cdot\log\mathcal{N})$. The whole ED text $\tilde{T}$ is thus: $\tilde{T} = \tilde{T}_1^p\cdot\cdots\cdot\tilde{T}_{\log z}^p\cdot\mathcal{X}\cdot\tilde{T}_{\log z}^s\cdot\cdots\cdot\tilde{T}_1^s$. We next show how a solution of such instance of EDSM corresponds to the solution of TD.

Lemma 2. *The pattern P occurs in the ED text $\tilde{T}$ if and only if there exist i,j,k such that $A[i,j] = B[j,k] = C[k,i] = 1$.*

Proof. By Fact 4, if such i,j,k exist then P_t matches $\mathcal{X}$, for some $t \in \{1,\ldots,z\}$. Then, by construction of the sets $\tilde{T}_i^p$ and $\tilde{T}_i^s$, the prefix $P_1\ldots P_{t-1}$ matches the ED prefix (this can be proved by induction), and similarly the suffix $P_{t+1}\ldots P_z$ matches the ED suffix, so the whole P matches $\tilde{T}$, and so P occurs therein. Because of the letters \$ appearing only in the center of P_is and strings from $\mathcal{X}_2$, every P_is and a concatenation of $X_1 \in \mathcal{X}_1$, $X_2 \in \mathcal{X}_2$, $X_3 \in \mathcal{X}_3$ having the same length, and the P_is being distinct, there is an occurrence of the pattern P in $\tilde{T}$ if and only if $X_1X_2X_3 = P_t$ for some t and $X_1 \in \mathcal{X}_1$, $X_2 \in \mathcal{X}_2$, $X_3 \in \mathcal{X}_3$. But then, by Fact 4 there exists a triangle. □

Note that for the EDSM problem we have $m = \mathcal{N}^2\cdot s$, $n = 1+2\log z$ and $N = ||\mathcal{X}|| + \mathcal{O}(\mathcal{N}^2\cdot s\cdot\log\mathcal{N})$. Thus if we had a solution running in $\mathcal{O}(\log z\cdot m^{1.5-\epsilon} + ||\mathcal{X}|| + \mathcal{N}^2\cdot s\cdot\log\mathcal{N}) = \mathcal{O}(\log\mathcal{N}\cdot(\mathcal{N}^2\cdot s)^{1.5-\epsilon} + \mathcal{N}^3/s)$ time, for some $\epsilon > 0$, by choosing a sufficiently small $\alpha > 0$ and setting $s = \mathcal{N}^\alpha$ we would obtain, for some $\delta > 0$, an $\mathcal{O}(\mathcal{N}^{3-\delta})$-time algorithm for TD. □

2.5 An $\tilde{\mathcal{O}}(nm^{\omega-1}+N)$-time Algorithm for EDSM

Our goal is to design a non-combinatorial $\tilde{\mathcal{O}}(nm^{\omega-1}+N)$-time algorithm for EDSM, which in turn can be achieved with a non-combinatorial $\tilde{\mathcal{O}}(m^{\omega-1}+N)$-time algorithm for the AP problem, that is the bottleneck of EDSM (cf. [178]).

We reduce AP to a logarithmic number of restricted instances of the same problem, based on the length of the strings in $\mathcal{S}$. We start by giving a lemma that we will use to process naïvely the strings of length up to a constant c, to be determined later, in $\mathcal{O}(m+N)$ time.

Lemma 3. *For any integer t, all strings in $\mathcal{S}$ of length at most t can be processed in $\mathcal{O}(m\log m+mt+N)$ time.*

Proof. We first construct the suffix tree ST of P and store, for every node, the first letters on its outgoing edges in a static dictionary with constant access time. This can be done in $\mathcal{O}(m\log m)$ time [313]. For every $S\in\mathcal{S}$, find and mark its corresponding (implicit or explicit) node of ST. This takes $\mathcal{O}(N)$ time overall. For every possible length $t'\leq t$, scan P with a window of length t' while maintaining its corresponding node of ST. This takes $\mathcal{O}(m)$ time overall. If the current window $P[i\,..\,(i+t'-1)]$ corresponds to a marked node of ST and additionally $U[i-1]=1$, we set $V[i+t'-1]=1$. □

We build the rest of the restricted instances of the AP problem by restricting on strings in $\mathcal{S}_k\subseteq\mathcal{S}$ of length in $[(19/18)^k,(19/18)^{k+1})$ for each integer k ranging from $\left\lceil\frac{\log c}{\log(19/18)}\right\rceil$ to $\left\lfloor\frac{\log m}{\log(19/18)}\right\rfloor$. These intervals are a partition of the set of all strings in $\mathcal{S}$ of length up to m; longer strings are not addressed in EDSM by solving AP.

For each integer k from $\left\lceil\frac{\log c}{\log(19/18)}\right\rceil$ to $\left\lfloor\frac{\log m}{\log(19/18)}\right\rfloor$, let ℓ be an integer such that the length of every string in $\mathcal{S}_k$ belongs to $[9/8\cdot\ell,5/4\cdot\ell)$. Note that such an integer always exists for an appropriate choice of the integer constant c. In fact, it must hold that

$$\frac{9}{8}\cdot\ell\ \leq\ \left(\frac{19}{18}\right)^k<\ \left(\frac{19}{18}\right)^{k+1}\ \leq\ \frac{5}{4}\cdot\ell\iff\frac{4}{5}\cdot\left(\frac{19}{18}\right)^{k+1}\ \leq\ \ell\ \leq\ \frac{8}{9}\cdot\left(\frac{19}{18}\right)^k.$$

To ensure that there exists an *integer* ℓ satisfying such conditions, it must actually hold that

$$\frac{4}{5}\cdot\left(\frac{19}{18}\right)^{k+1}+1\ \leq\ \frac{8}{9}\cdot\left(\frac{19}{18}\right)^k\iff\frac{45}{2}\ \leq\ \left(\frac{19}{18}\right)^k.$$

The last equation holds for $k\geq 58$, implying that we will process naïvely the strings of length up to $c=23$, and each $\mathcal{S}_k$, for k ranging from 58 to $\left\lfloor\frac{\log m}{\log(19/18)}\right\rfloor$, will be processed separately as described in the next paragraph.

Denoting by N_k the total size of strings in $\mathcal{S}_k$, we have that, if we solve every such instance of AP in $\mathcal{O}(N_k+f(m))$ time, then we can solve the original instance of AP in $\mathcal{O}(N+f(m)\log m)$ time by taking the results disjunction. Switching to $\tilde{\mathcal{O}}$ notation that disregards polylog factors, it thus suffices to solve each such instance of the AP problem in $\tilde{\mathcal{O}}(N+m^{\omega-1})$ time.

We further partition the strings in $\mathcal{S}_k$ into three types, compute the corresponding bit vector V for each type separately and, finally, take the disjunction of the resulting bit vectors V to obtain the answer for each restricted instance.

Partitioning $\mathcal{S}_k$. Keeping in mind that from now on (until Section 2.5.4) we address the AP problem assuming that $\mathcal{S}$ only contains strings of length in $[9/8 \cdot \ell, 5/4 \cdot \ell)$, and thus is in fact $\mathcal{S}_k$, to lighten the notation we now switch back to denote it simply with $\mathcal{S}$. The three types of strings are as follows:

Type 1: Strings $S \in \mathcal{S}$ such that every length-ℓ substring of S is not strongly periodic.

Type 2: Strings $S \in \mathcal{S}$ containing at least one length-ℓ substring that is not strongly periodic and at least one length-ℓ substring that is strongly periodic.

Type 3: Strings $S \in \mathcal{S}$ such that every length-ℓ substring of S is strongly periodic (in Lemma 4 we show that in this case $\mathrm{per}(S) \leq \ell/4$).

These three types are evidently a partition of $\mathcal{S}$. We start with showing that, in fact, strings of type 3 are exactly strings with period at most $\ell/4$.

Lemma 4. *Let S be a string. If $\mathrm{per}(S[j\,..\,j+\ell-1]) \leq \ell/4$ for every j then $\mathrm{per}(S) \leq \ell/4$.*

Proof. We first show that, for any string W and letters a, b, if $\mathrm{per}(aW) \leq |aW|/4$ and $\mathrm{per}(Wb) \leq |Wb|/4$ then $\mathrm{per}(aW) = \mathrm{per}(Wb)$. This follows from Lemma 1: since $\mathrm{per}(aW)$ and $\mathrm{per}(Wb)$ are both periods of W and $(1 + |W|)/4 \leq |W|/2$, then we have that $d = \gcd(\mathrm{per}(aW), \mathrm{per}(Wb))$ is a period of W. Assuming by contradiction that $\mathrm{per}(aW) \neq \mathrm{per}(Wb)$, then it must be that either $d < \mathrm{per}(aW)$ or $d < \mathrm{per}(Wb)$; by symmetry it is enough to consider the former possibility, and we claim that then d is a period of aW. Indeed, $a = W[\mathrm{per}(aW) - 1]$ (observe that $\mathrm{per}(aW) - 1 \leq |W|$) and $W[i] = W[i + d]$ for any $i = 1, 2, \ldots, |W| - d$, so by $\mathrm{per}(aW)$ being a multiple of d we obtain that $a = W[\mathrm{per}(aW) - 1] = W[d - 1]$, which is a contradiction because by definition of $\mathrm{per}(aW)$ we have that $d < \mathrm{per}(aW)$ cannot be a period of aW.

If $\mathrm{per}(S[j\,..\,j + \ell - 1]) \leq \ell/4$ for every j then by the above reasoning the periods of all substrings $S[j\,..\,j+\ell-1]$ is the same and in fact equal to p. But then $S[i] = S[i+p]$ for every i, so $\mathrm{per}(S) \leq \ell/4$. □

Before proceeding with the algorithm, we show that, for each string $S \in \mathcal{S}$, we can determine its type in $\mathcal{O}(|S|)$ time.

Lemma 5. *Given a string S we can determine its type in $\mathcal{O}(|S|)$ time.*

Proof. It is well-known that $\mathrm{per}(T)$ can be computed in $\mathcal{O}(|T|)$ time for any string T (cf. [113]). We partition S into blocks $T_\alpha = S[\alpha\lfloor \ell/2 \rfloor\,..\,(\alpha + 1)\lfloor \ell/2 \rfloor - 1]$ of size $\lfloor \ell/2 \rfloor$, and compute $\mathrm{per}(T_\alpha)$ for every α in $\mathcal{O}(|S|)$ total time. Observe that every substring $S[i\,..\,i + \ell - 1]$ contains at least one whole block inside.

If $\mathrm{per}(T_\alpha) > \ell/4$ then the period of any substring $S[i\,..\,i + \ell - 1]$ that contains T_α is also larger than $\ell/4$. Consequently, if $\mathrm{per}(T_\alpha) > \ell/4$ for every α, then we declare S to be of type 1.

Consider any α such that $p = \mathrm{per}(T_\alpha) \leq \ell/4$. If the period p' of a substring $S' = S[i\,..\,i + \ell - 1]$ that contains T_α is at most $\ell/4$, then in fact it must be equal to p, because $p' \geq p$ and so, by Lemma 1 applied on T_α, p' must be a multiple of p and, by repeatedly applying $S'[j] = S'[j + p']$ and $T_\alpha[j] = T_\alpha[j + p]$ and using the fact that T_α occurs inside S', we conclude that in fact $S'[j] = S'[j + p]$ for any j, and thus $p' = p$. This allows us to check whether there exists a substring $S' = S[i\,..\,i+\ell-1]$ that

contains T_α such that $\mathrm{per}(S') \leq \ell/4$ by computing, in $\mathcal{O}(\ell)$ time, how far the period p extends to the left and to the right of T_α in $T_{\alpha-1}T_\alpha T_{\alpha+1}$ (if either $T_{\alpha-1}$ or $T_{\alpha+1}$ do not exist, then we do not extend the period in the corresponding direction). There exists such a substring S' if and only if the length of the extended substring with period p is at least ℓ. Therefore, for every α we can check in $\mathcal{O}(\ell)$ time if there exists a length-ℓ substring S' containing T_α with $\mathrm{per}(S') \leq \ell/4$. By repeating this procedure for every α, we can distinguish between S of type 2 and S of type 3 in $\mathcal{O}(|S|)$ total time. □

Since we have shown how to efficiently partition the strings of S into the three types, in what follows we present our solution of the AP problem for each type of strings separately.

Remark 1. *The length of every string in $\mathcal{S}$ belonging to $[9/8 \cdot \ell, 5/4 \cdot \ell)$ implies that every string in $\mathcal{S}$ contains at most $\ell/4$ length-ℓ substrings (and at least $1 + \ell/8$ of them).*

2.5.1 Type 1 Strings

In this section we show how to solve a restricted instance of the AP problem where every string $S \in \mathcal{S}$ is of type 1, that is, each of its length-ℓ substrings is not strongly periodic, and furthermore $|S| \in [9/8 \cdot \ell, 5/4 \cdot \ell)$ for some $\ell \leq m$. Observe that all (and hence at most $\ell/4$ by Remark 1) length-ℓ substrings of any $S \in \mathcal{S}$ must be distinct, as otherwise we would be able to find two occurrences of a length-ℓ substring at distance at most $\ell/4$ in S, making the period of the substring at most $\ell/4$ and contradicting the assumption that S is of type 1.

We start with constructing the suffix tree ST of P (our pattern in the EDSM problem) in $\mathcal{O}(m \log m)$ time [357]. Let us remark that we are spending $\mathcal{O}(m \log m)$ time and not just $\mathcal{O}(m)$ so as to avoid any assumptions on the size of the alphabet. For every explicit node $u \in ST$, we construct a perfect hash function mapping the first letter on every edge outgoing from u to the corresponding edge. This takes $\mathcal{O}(m \log m)$ time [313] and allows us to navigate in ST in constant time per letter. Then, for every $S \in \mathcal{S}$, we check in $\mathcal{O}(|S|)$ time using ST if it occurs in P and, if not, we disregard it from further consideration. Therefore, from now on we assume that all strings S, and thus all their length-ℓ substrings, occur in P. We will select a set of length-ℓ substrings of P, called the *anchors*, each represented by one of its occurrences in P, such that:

1. The total number of occurrences of all anchors in P is $\mathcal{O}(m/\ell \cdot \log m)$.
2. For every $S \in \mathcal{S}$, at least one of its length-ℓ substrings is an anchor.
3. The total number of occurrences of all anchors in strings $S \in \mathcal{S}$ is $\mathcal{O}(|\mathcal{S}| \cdot \log m)$.

We formalize this using the following auxiliary problem, which is a strengthening of a well-known *Hitting Set* problem, which given a collection of m sets over $[n]$, each of size at least k, asks to choose a subset of $[n]$ of size $\mathcal{O}(n/k \cdot \log m)$ that nontrivially intersects every set.

NODE SELECTION (NS)
INPUT: A bipartite graph $G = (U, V, E)$ with $\deg(u) \in (d, 2d]$ for every $u \in U$ and weight $w(v)$ for every $v \in V$, where $W = \sum_{v \in V} w(v)$.
OUTPUT: A set $V' \subseteq V$ of total weight $\mathcal{O}(W/d \cdot \log |U|)$ such that $N[U] \cap V' \neq \emptyset$ for every node in U, and $\sum_{u \in U} |N[u] \cap V'| = \mathcal{O}(|U| \log |U|)$.

We reduce the problem of finding anchors to an instance of the NS problem, by building a bipartite graph G in which the nodes in U correspond to strings $S \in \mathcal{S}$, the nodes in V correspond to distinct length-ℓ substrings of P, and there is an edge (u, v) if the length-ℓ string corresponding to v occurs in the string S corresponding to u. Using suffix links, we can find the node of the suffix tree corresponding to every length-ℓ substring of S in $\mathcal{O}(|S|)$ total time, so the whole construction takes $\mathcal{O}(m \log m + \sum_{S \in \mathcal{S}} |S|) = \mathcal{O}(m \log m + N)$ time. The size of G is $\mathcal{O}(m + N)$, and the degree of every node in U belongs to $(\ell/8, \ell/4]$. We set the weight of a node $v \in V$ to be its number of occurrences in P, and solve the obtained instance of the NS problem to obtain the set of anchors. It is not immediately clear that an instance of the NS problem always has a solution. We show that indeed it does, and that it can be found in linear time.

Lemma 6. *A solution to an instance of the NS problem always exists and can be found in linear time in the size of G.*

Proof. We first show a solution that uses the probabilistic method and leads us to an efficient Las Vegas algorithm; we will then derandomize the solution using the method of conditional expectations.

We independently choose each node of V with probability p to obtain the set V' of selected nodes. The expected total weight of V' is $\sum_{v \in V} p \cdot w(v) = p \cdot W$, so by Markov's inequality it exceeds $4p \cdot W$ with probability at most $1/4$. For every node $u \in U$, the probability that $N[u]$ intersects V' is at most $(1-p)^d \leq e^{-pd}$. Finally, $\mathbb{E}[\sum_{u \in U} |N[u] \cap V'|] \leq |U| \cdot 2pd$, so by Markov's inequality $\sum_{u \in U} |N[u] \cap V'|$ exceeds $|U| \cdot 8pd$ with probability at most $1/4$. We set $p = \ln(4|U|)/d$ (observe that if $p > 1$ then we can select all nodes in V). By union bound, the probability that V' is not a valid solution is at most $3/4$, so indeed a valid solution exists. Furthermore, this reasoning gives us an efficient Las Vegas algorithm that chooses V' randomly as described above and then verifies if it constitutes a valid solution. Each iteration takes linear time in the size of G, and the expected number of required iterations is constant.

To derandomize the above procedure we apply the method of conditional expectations. Let $X_1, X_2, \ldots$ be the binary random variables corresponding to the nodes of V. Recall that in the above proof we set $X_i = 1$ with probability p. Now we will choose the values of $X_1, X_2, \ldots$ one-by-one. Define a function $f(X_1, X_2, \ldots)$ that that bounds the probability that $X_1, X_2, \ldots$ corresponds to a valid solution as follows:

$$f(X_1, X_2, \ldots) = \frac{\sum_v X_v \cdot w(v)}{4W/d \cdot \ln(4|U|)} + \sum_{u \in U} \prod_{v \in N[u]} (1 - X_v) + \frac{\sum_{u \in U} \sum_{v \in N[u]} X_v}{8|U| \ln(4|U|)}.$$

As explained above, we have $\mathbb{E}[f(X_1, X_2, \ldots)] = 3/4$. Assume that we have already fixed the values $X_1 = x_1, \ldots, X_i = x_i$. Then there must be a choice of $X_{i+1} = x_{i+1}$

that does not increase the expected value of $f(X_1, X_2, \ldots)$ conditioned on the already chosen values. We want to compare the following two quantities:

$$\mathbb{E}[f(X_1, X_2, \ldots) \,|\, X_1 = x_1, \ldots, X_i = x_i, X_{i+1} = 0]$$
$$\mathbb{E}[f(X_1, X_2, \ldots) \,|\, X_1 = x_1, \ldots, X_i = x_i, X_{i+1} = 1]$$

and choose x_{i+1} corresponding to the smaller one. Cancelling out the shared terms, we need to compare the expected values of:

$$0 \quad + \sum_{u \in N[i+1]} \prod_{v \in N[u]} (1 - X_v) \; + \quad 0 \quad \text{and}$$
$$\frac{w(i+1)}{4W/d \cdot \ln(4|U|)} \; + \quad 0 \quad + \; \frac{\deg(i+1)}{8|U| \ln(4|U|)}.$$

The second quantity can be computed in constant time. We claim that (ignoring the issue of numerical precision) the first quantity can be computed in time $\mathcal{O}(\deg(i+1))$ after a linear-time preprocessing as follows. In the preprocessing we compute and store $E[i] = \mathbb{E}[\prod_{j=1}^{i}(1 - Y_j)]$, where the Y_j's are independent indicator variables with $\Pr[Y_j = 1] = p$, for every $i = 0, 1, \ldots, |V|$. It is straightforward to compute $E[i+1]$ from $E[i]$ in constant time. Then, during the computation we maintain, for every $u \in U$, the number $c[u]$ of $v \in N[u]$ for which we still need to choose the value X_e, and a single bit $b[u]$ denoting whether for some $v \in N[u] \cap \{1, \ldots, i\}$ we already have $x_v = 1$. This information can be updated in $\mathcal{O}(\deg(i+1))$ time after selecting x_{i+1}. Now to compute the first quantity, we iterate over $u \in N[i+1]$ and, if $b[u] = 0$ then we add $E[c[u]]$ to the result. Finally, we claim that it is enough to implement all calculations with precision $\Theta(\log |V|)$ bits. This is because such precision allows us to calculate both quantities with relative accuracy $1/(8|V|)$, so the expected value of $f(X_1, X_2, \ldots)$ might increase by a factor of $(1 + 1/(4|V|))$ in every step, which is at most $(1 + 1/(4|V|))^{|V|} \leq e^{1/4}$ overall. This still guarantees that the final value is at most $3/4 \cdot e^{1/4} < 1$, so we obtain a valid solution. □

In the rest of this section we explain how to compute the bit vector V from the bit vector U, and thus solve the AP problem, after having obtained a set $\mathcal{A}$ of anchors. For any $S \in \mathcal{S}$, since S contains an occurrence of at least one anchor $H \in \mathcal{A}$, say $S[j\,..(j+|H|-1)] = H$, so any occurrence of S in P can be generated by choosing some occurrence of H in P, say $P[i\,..(i+|H|-1)] = H$, and then checking that $S[1\,..(j-1)] = P[(i-j+1)\,..(i-1)]$ and $S[(j+|H|)\,..|S|] = P[(i+|H|)\,..(i+|S|-j)]$. In other words, $S[1\,..(j-1)]$ should be a suffix of $P[1\,..(i-1)]$ and $S[(j+|H|)\,..|S|]$ should be a prefix of $P[(i+|H|)\,..|P|]$. In such case, we say that the occurrence of S in P is generated by H. By the properties of $\mathcal{A}$, any occurrence of $S \in \mathcal{S}$ is generated by $occ_S \geq 1$ occurrences of anchors, where $\sum_{S \in \mathcal{S}} occ_S = \mathcal{O}(|\mathcal{S}| \log m)$. For every $H \in \mathcal{A}$ we create a separate data structure $D(H)$ responsible for setting $V[i+|S|-1] = 1$, when $U[i-1] = 1$ and $P[i\,..(i+|S|-1)] = S$ is generated by H. We now first describe what information is used to initialize each $D(H)$, and how this is later processed to update V.

Initialization. $D(H)$ consists of two compact tries $T(H)$ and $T^r(H)$. For every occurrence of H in P, denoted by $P[i\,..(i+|H|-1)] = H$, $T(H)$ should contain a leaf

corresponding to $P[(i+|H|)\,..\,|P|]\$$ and $T^r(H)$ should contain a leaf corresponding to $(P[1\,..\,(i-1)])^r\$$, both decorated with position i. Additionally, $D(H)$ stores a list $L(H)$ of pairs of nodes (u,v), where $u \in T^r(H)$ and $v \in T(H)$. Each such pair corresponds to an occurrence of H in a string $S \in \mathcal{S}_h$, $S[j\,..\,(j+|H|-1)] = H$, where u is the node of $T^r(H)$ corresponding to $(S[1\,..\,(j-1)])^r\$$ and v is the node of $T(H)$ corresponding to $S[(j+|H|+1)\,..\,|S|]\$$. We claim that $D(H)$, for all H, can be constructed in $\mathcal{O}(m \log m + N)$ total time.

We first construct the suffix tree ST of $P\$$ and the suffix tree ST^r of $P^r\$$ (again in $\mathcal{O}(m \log m)$ time not to make assumptions on the alphabet). We augment both trees with data for answering both *weighted ancestor* (WA) and *lowest common ancestor* (LCA) queries, that are defined as follows. For a rooted tree T on n nodes with an integer weight $\mathcal{D}(v)$ assigned to every node u, such that the weight of the root is zero and $\mathcal{D}(u) < \mathcal{D}(v)$ if u is the parent of v, we say that a node v is a weighted ancestor of a node v at depth ℓ, denoted by $\mathrm{WA}_T(u,\ell)$, if v is the highest ancestor of u with weight of at least ℓ. Such queries can be answered in $\mathcal{O}(\log n)$ time after an $\mathcal{O}(n)$ preprocessing [140]. For a rooted tree T, $\mathrm{LCA}_T(u,v)$ is the lowest node that is an ancestor of both u and v. Such queries can be answered in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$ preprocessing [47]. Recall that every anchor H is represented by one of its occurrences in P. Using WA queries, we can access in $\mathcal{O}(\log m)$ time the nodes corresponding to H and H^r, respectively, and extract a lexicographically sorted list of suffixes following an occurrence of H in $P\$$ and a lexicographically sorted list of reversed prefixes preceding an occurrence of H in $P^r\$$ in time proportional to the number of such occurrences. Then, by iterating over the lexicographically sorted list of suffixes and using LCA queries on ST we can build $T(H)$ in time proportional to the length of the list, and similarly we can build $T^r(H)$. To construct $L(H)$ we start by computing, for every $S \in \mathcal{S}$ and $j = 1, \ldots, |S|$, the node of ST^r corresponding to $(S[1\,..\,j])^r$ and the node of ST corresponding to $S([(j+1)\,..\,|S|]$ (the nodes might possibly be implicit). This takes only $\mathcal{O}(|S|)$ time, by using suffix links. We also find, for every length-ℓ substring $S[j\,..\,(j+\ell-1)]$ of S, an anchor $H \in \mathcal{A}$ such that $S[j\,..\,(j+\ell-1)] = H$, if any exists. This can be done by finding the nodes (implicit or explicit) of ST that correspond to the anchors, and then scanning over all length-ℓ substrings while maintaining the node of ST corresponding to the current substring using suffix links in $\mathcal{O}(|S|)$ total time. After having determined that $S[j\,..\,(j+\ell-1)] = H$ we add (u,v) to $L(H)$, where u and v are the previously found nodes of ST^r and ST corresponding to $(S[1\,..\,(j-1)])^r$ and $S[(j+\ell)\,..\,|S|]$, respectively. By construction, we have the following property, also illustrated in Figure 2.1.

Fact 5. *A string $S \in \mathcal{S}$ starts at position $i-j+1$ in P if and only if, for some anchor $H \in \mathcal{A}$, $L(H)$ contains a pair (u,v) corresponding to $S[j\,..\,(j+|H|-1)] = H$, such that the subtree of $T^r(H)$ rooted at u and that of $T(H)$ rooted at v contain a leaf decorated with i.*

Note that the overall size of all lists $L(H)$, when summed up over all $H \in \mathcal{A}$, is $\sum_{S\in\mathcal{S}} occ_S = \mathcal{O}(|\mathcal{S}| \log m)$, and since each S is of length at least ℓ this is $\mathcal{O}(N/\ell \cdot \log m)$.

Processing. The goal of processing $D(H)$ is to efficiently process all occurrences generated by H. As a preliminary step, we decompose $T^r(H)$ and $T(H)$ into heavy paths. Then, for every pair of leaves $u \in T^r(H)$ and $v \in T(H)$ decorated by the same

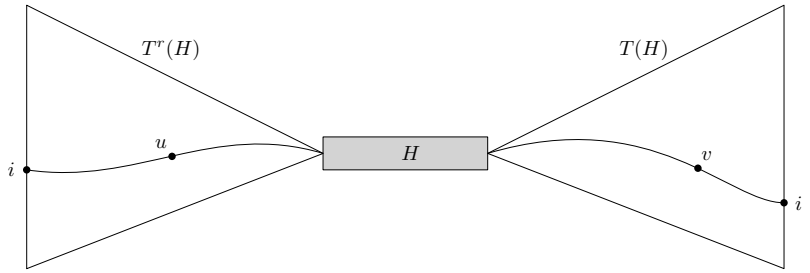


Figure 2.1: An occurrence of S starting at position i in P is generated by H: (u, v) corresponds to $S[j\,..\,(j + |H| - 1)] = H$ and i appears in the subtree of $T^r(H)$ rooted at u, as well as in the subtree of $T(H)$ rooted at v.

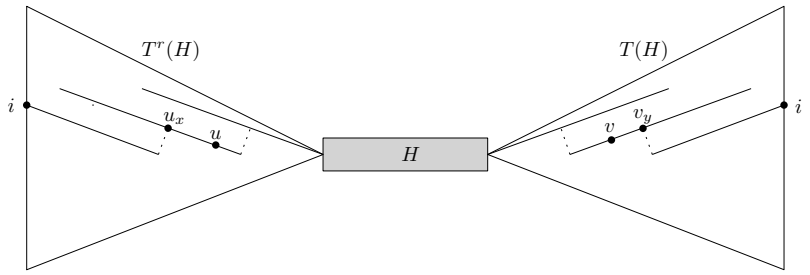


Figure 2.2: An occurrence of S starting at position i in P corresponds to a triple $(i, \mathcal{L}(u_x), \mathcal{L}(v_y))$ on some auxiliary list.

i, we consider all heavy paths above u and v. Let $p = u_1 - u_2 - \ldots$ be a heavy path above u in $T^r(H)$ and $q = v_1 - v_2 - \ldots$ be a heavy path above v in $T(H)$, where u_1 is the head of p and v_1 is the head of q, respectively. Further, choose the largest x such that u is in the subtree rooted at u_x, and the largest y such that v is in the subtree rooted at v_y (this is well-defined by the choice of p and q, as u is in the subtree rooted at u_1 and v is in the subtree rooted at v_1). We add $(i, |\mathcal{L}(u_x)|, |\mathcal{L}(v_y)|)$ to an auxiliary list associated with the pair of heavy paths (p, q). In the rest of the processing we work with each such lists separately. Notice that the overall size of all auxiliary lists, when summed up over all $H \in \mathcal{A}$, is $\mathcal{O}(m/\ell \cdot \log^3 m)$, because there are at most $\log^2 m$ pairs of heavy paths above u and v decorated by the same i, and the total number of leaves in all trees $T^r(H)$ and $T(H)$ is bounded by the total number of occurrences of all anchors in P, which is $\mathcal{O}(m/\ell \cdot \log m)$. By Fact 5, there is an occurrence of a string $\mathcal{S}$ generated by H and starting at position $i - j + 1$ in P if and only if $L(H)$ contains a pair (u, v) corresponding to $S[j\,..\,(j + |H| - 1)] = H$ such that, denoting by p the heavy path containing u in $T^r(H)$ and by q the heavy path containing v in $T(H)$, the auxiliary list associated with (p, q) contains a triple (i, x, y) such that $x \geq |\mathcal{L}(u)|$ and $y \geq |\mathcal{L}(v)|$. This is illustrated in Figure 2.2. Henceforth, we focus onthe problem of processing a single auxiliary list associated with (p, q), together with a list of pairs (u, v), such that u belongs to p and v belongs to q.

An auxiliary list can be interpreted geometrically as follows: for every (i, x, y) we

create a red point (x, y), and for every (u, v) we create a blue point $(|\mathcal{L}(u)|, |\mathcal{L}(v)|)$. Then, each occurrence of $S \in \mathcal{S}$ generated by H corresponds to a pair of points (p_1, p_2) such that p_1 is red, p_2 is blue, and p_1 dominates p_2. We further reduce this to a collection of simpler instances in which all red points already dominate all blue points. This can be done with a divide-and-conquer procedure which is essentially equivalent to constructing a 2D range tree [49]. The total number of points in all obtained instances increases by a factor of $\mathcal{O}(\log^2 m)$, making the total number of red points in all instances $\mathcal{O}(m/\ell \cdot \log^5 m)$, while the total number of blue points is $\mathcal{O}(N/\ell \cdot \log^3 m)$. There is an occurrence of a string $S \in \mathcal{S}$ generated by H and starting at position $i - j + 1$ in P if and only if some simpler instance contains a red point created for some (i, x, y) and a blue point created for some (u, v) corresponding to $S[j\,..\,(j + |H| - 1)] = H$. In the following we focus on processing a single simpler instance.

To process a simpler instance we need to check if $U[i - j] = 1$, for a red point created for some (i, x, y) and a blue point created for some (u, v) corresponding to $S[j\,..\,(j + |H| - 1)] = H$, and if so set $V[i - j + |S|] = 1$. This has a natural interpretation as an instance of BMM: we create a $\lceil 5/4 \cdot \ell \rceil \times \lceil 5/4 \cdot \ell \rceil$ matrix M such that $M[|S| - j, \lceil 5/4 \cdot \ell \rceil + 1 - j] = 1$ if and only if there is a blue point created for some (u, v) corresponding to $S[j\,..\,(j + |H| - 1)] = H$; then for every red point created for some (i, x, y) we construct a bit vector $U_i = U[(i - \lceil 5/4 \cdot \ell \rceil)\,..\,(i - 1)]$ (if $i < \lceil 5/4 \cdot \ell \rceil$, we pad U_i with 0s to make its length always equal to $\lceil 5/4 \cdot \ell \rceil$); calculate $V_i = M \times U_i$; and finally set $V[i + j] = 1$ whenever $V_i[j] = 1$ (and $i + j \leq m$).

Lemma 7. *$V_i[k] = 1$ if and only if there is a blue point created for some (u, v) corresponding to $S[j\,..\,(j + |H| - 1)] = H$ such that $U[i - j] = 1$ and $k = |S| - j$.*

Proof. By definition of $V_i = M \times U_i$, we have that $V_i[k] = 1$ if and only if $M[k, t] = 1$ for some t such that $U_i[t] = 1$. By definition of U_i, we have that $U_i[t] = 1$ if and only if $U[i - \lceil 5/4 \cdot \ell \rceil + t - 1] = 1$, and hence the previous condition can be rewritten as $M[k, t] = 1$ and $U[i - \lceil 5/4 \cdot \ell \rceil + t - 1] = 1$, or equivalently, by substituting $j = \lceil 5/4 \cdot \ell \rceil + 1 - t$, $M[k, \lceil 5/4 \cdot \ell \rceil + 1 - j] = 1$ and $U[i - j] = 1$. By definition of M, we have that $M[k, \lceil 5/4 \cdot \ell \rceil + 1 - j] = 1$ if and only if there is a blue point created for some (u, v) corresponding to $S[j\,..\,(j + |H| - 1)] = H$ with $k = |S| - j$, which proves the lemma. □

The total length of all vectors U_i and V_i is $\mathcal{O}(m \log^5 m)$, so we can afford to extract the appropriate fragment of U and then update the corresponding fragment of V. The bottleneck is computing the matrix-vector product $V_i = M \times U_i$. Since the total number of 1s in all matrices M is bounded by the total number of blue points, a naïve method would take $\mathcal{O}(N/\ell \cdot \log^3 m)$ time; we overcome this by processing together all multiplications concerning the same matrix M, thus amortizing the costs. Let $U_{i_1}, U_{i_2}, \ldots, U_{i_s}$ be all bit vectors that need to be multiplied with M, and let z a parameter to be determined later. We distinguish between two cases: (i) if $s < z$, then we compute the products naïvely by iterating over all 1s in M, and the total computation time, when summed up over all such matrices M, is $\mathcal{O}(N/\ell \cdot \log^3 m \cdot z)$; (ii) if $s \geq z$, then we partition the bit vectors into $\lceil s/z \rceil \leq s/z + 1$ groups of z (padding the last group with bit vectors containing all 0s) and, for every group, we create a single matrix whose columns contain all the bit vectors belonging to the group. Thus, we reduce the problem of computing all matrix-vector products $M \times U_i$ to that of computing $\mathcal{O}(s/z)$

matrix-matrix products of the form $M \times M'$, where M' is an $\lceil 5/4 \cdot \ell \rceil \times z$ matrix. Even if M' is not necessarily a square matrix, we can still apply the fast matrix multiplication algorithm to compute $M \times M'$ using the standard trick of decomposing the matrices into square blocks.

Lemma 8. *If two $\mathcal{N} \times \mathcal{N}$ matrices can be multiplied in $\mathcal{O}(\mathcal{N}^\omega)$ time, then, for any $\mathcal{N} \geq \mathcal{N}'$, an $\mathcal{N} \times \mathcal{N}$ and an $\mathcal{N} \times \mathcal{N}'$ matrix can be multiplied in $\mathcal{O}((\mathcal{N}/\mathcal{N}')^2 \mathcal{N}'^\omega)$ time.*

Proof. We partition both matrices into blocks of size $\mathcal{N}' \times \mathcal{N}'$. There are $(\mathcal{N}/\mathcal{N}')^2$ such blocks in the first matrix and $\mathcal{N}/\mathcal{N}'$ in the second matrix. Then, to compute the product we multiply each block from the first matrix by the appropriate block in the second matrix in $\mathcal{O}(\mathcal{N}'^\omega)$ time, resulting in the claimed complexity. □

By applying Lemma 8, we can compute $M \times M'$ in $\mathcal{O}(\ell^2 z^{\omega-2})$ time (as long as we later verify that $5/4 \cdot \ell \geq z$), so all products $M \times U_i$ can be computed in $\mathcal{O}(\ell^2 z^{\omega-2} \cdot (s/z+1))$ time. Note that this case can occur only $\mathcal{O}(m/(\ell \cdot z) \cdot \log^5 m)$ times, because all values of s sum up to $\mathcal{O}(m/\ell \cdot \log^5 m)$. This makes the total computation time, when summed up over all such matrices M, $\mathcal{O}(\ell^2 z^{\omega-2} \cdot m/(\ell \cdot z) \cdot \log^5 m) = \mathcal{O}(\ell z^{\omega-3} \cdot m \log^5 m)$. We can now prove our final result for strings of type 1.

Theorem 6. *An instance of the AP problem where all strings are of type 1 can be solved in $\tilde{\mathcal{O}}(m^{\omega-1} + N)$ time.*

Proof. The total time complexity is first $\mathcal{O}(m + N)$ to construct the graph G, then $\mathcal{O}(m \log m + N)$ to solve its corresponding instances of the NodeSelection problem and obtain the set of anchors H. The time to initialize all structures $D(H)$ is $\mathcal{O}(m \log m + N)$. For every $D(H)$, we obtain in $\mathcal{O}(m/\ell \cdot \log^5 m + N/\ell \cdot \log^3 m)$ time a number of simpler instances, and then construct the corresponding Boolean matrices M and bit vectors U_i in additional $\mathcal{O}(m \log^5 m)$ time. Note that some M might be sparse, so we need to represent them as a list of 1s. Then, summing up over all matrices M and both cases, we spend $\mathcal{O}(N/\ell \cdot \log^3 m \cdot z + \ell z^{\omega-3} \cdot m \log^5 m)$ time. We would like to assume that $\ell \geq \log^3 m$ so that we can set $z = \ell / \log^3 m$. This is indeed possible, because for any t we can switch to a more naïve approach to process all strings of length at most t in $\mathcal{O}(mt^2 + N)$ time as described in 3. After applying it with $t = \log^3 m$ in $\mathcal{O}(m \log^6 m + N)$ time, we can set $z = \ell / \log^3 m$ (so that indeed $5/4 \cdot \ell \geq z$ as required in case $s \geq z$) and the overall time complexity for all matrices M and both cases becomes $\mathcal{O}(N + \ell^{\omega-2} \cdot m \log^{5+3(3-\omega)} m)$. Summing up over all values of ℓ and taking the initialization into account we obtain $\mathcal{O}(m \log^7 m + m^{\omega-1} \log^{5+3(3-\omega)} m + N) = \tilde{\mathcal{O}}(m^{\omega-1} + N)$ total time. □

2.5.2 Type 2 Strings

In this section we show how to solve a restricted instance of the AP problem where every string $S \in \mathcal{S}$ is of type 2, that is, S contains a length-ℓ substring that is not strongly periodic as well as a length-ℓ substring that is strongly periodic, and furthermore $|S| \in [9/8 \cdot \ell, 5/4 \cdot \ell)$ for some $\ell \leq m$.

Similarly as in Section 2.5.1, we select a set of anchors. In this case, instead of the NodeSelection problem we need to exploit periodicity. We call a string T *ℓ-periodic* if $|T| \geq \ell$ and $\mathrm{per}(T) \leq \ell/4$. We consider all maximal ℓ-periodic substrings of S, that

is, ℓ-periodic substrings $S[i\,..\,j]$ such that either $i = 1$ or $\mathrm{per}(S[(i-1)\,..\,j]) > \ell/4$, and $j = |S|$ or $\mathrm{per}(S[i\,..\,(j+1)]) > \ell/4$. We know that S contains at least one such substring (because there exists a length-ℓ substring that is strongly periodic), and that the whole S is not such a substring (because otherwise S would be of type 3). Further, two maximal ℓ-periodic substrings cannot overlap too much, as formalized in the following lemma.

Lemma 9. *Any two distinct maximal ℓ-periodic substrings of the same string S overlap by less than $\ell/2$ letters.*

Proof. Assume (by contradiction) the opposite; then we have two distinct ℓ-periodic substrings $S[i\,..\,j]$ and $S[i'\,..\,j']$ such that $i < i' \leq j < j'$ and $j - i' + 1 \geq \ell/2$. Then, both $p = \mathrm{per}(S[i\,..\,j])$ and $p' = \mathrm{per}(S[i'\,..\,j'])$ are periods of $S[i'\,..\,j]$, and hence by Lemma 1 we have that $\gcd(p, p')$ is a period of $S[i'\,..\,j]$. If $p \neq p'$ then, because $S[i'\,..\,j]$ contains an occurrence of both $S[i\,..\,(i+p-1)]$ and $S[i'\,..\,(i'+p'-1)]$, we obtain that one of these two substrings is a power of a shorter string, thus contradicting the definition of p or p'. So $p = p'$, but then $p \leq \ell/4$ is actually a period of the whole $S[i\,..\,j']$, meaning that $S[i\,..\,j]$ and $S[i'\,..\,j']$ are not maximal, a contradiction. □

By Lemma 9, every $S \in \mathcal{S}$ contains exactly one maximal ℓ-periodic substring, and by the same argument P contains $\mathcal{O}(m/\ell)$ such substrings. The set of anchors will be generated by considering the unique maximal ℓ-periodic substring of every $S \in \mathcal{S}$, so we first need to show how to efficiently generate such substrings.

Lemma 10. *Given a string S of length at most $5/4 \cdot \ell$, we can generate its (unique) maximal ℓ-periodic substring in $\mathcal{O}(|S|)$ time.*

Proof. We start with observing that any length-ℓ substring of S must contain $S[(\lfloor \ell/2 \rfloor + 1)\,..\,\ell]$ inside. Consequently, we can proceed similarly as in the proof of Lemma 5. We compute $p = \mathrm{per}(S[(\lfloor \ell/2 \rfloor + 1)\,..\,\ell])$ in $\mathcal{O}(|S|)$ time. If $p > \ell/4$ then S does not contain any ℓ-periodic substrings. Otherwise, we compute in $\mathcal{O}(|S|)$ time how far the period p extends to the left and to the right; that is, we compute the smallest $i \leq \lfloor \ell/2 \rfloor + 1$ such that $S[k] = S[k+p]$ for every $k = i, i+1, \ldots, \lfloor \ell/2 \rfloor$ and the largest $j \geq \ell$ such that $S[k] = S[k-p]$ for every $k = \ell+1, \ell+2, \ldots, j$. If $j - i + 1 \geq \ell$ then $S[i\,..\,j]$ is a maximal ℓ-periodic substring of S, and, as shown earlier by Lemma 9, S cannot contain any other maximal ℓ-periodic substrings. We return $S[i\,..\,j]$ as the (unique) maximal ℓ-periodic substring of S. □

For every $S \in \mathcal{S}$, we apply Lemma 10 on S to find its (unique) maximal ℓ-periodic substring $S[i\,..\,j]$ in $\mathcal{O}(|S|)$ time. If $i > 1$ then we designate $S[(i-1)\,..\,(i-1+\ell)]$ as an anchor, and similarly if $j < |S|$ we designate $S[(j+1-\ell)\,..\,(j+1)]$ as an anchor. Observe that because S is of type 2 (and not of type 3) either $i > 1$ or $j < |S|$, so for every $S \in \mathcal{S}$ we designate at least one if its length-$(\ell+1)$ substrings as an anchor. As in Section 2.5.1, we represent each anchor by one of its occurrences in P, and so need to find its corresponding node in the suffix tree of P (if any). This can be done in $\mathcal{O}(|S|)$ time, so $\mathcal{O}(N)$ overall. During this process we might designate the same string as an anchor multiple times, but we can easily remove the possible duplicates to obtain the set $\mathcal{A}$ of anchors in the end. Then, we generate the occurrences of all anchors in

P by accessing their corresponding nodes in the suffix tree of P and iterating over all leaves in their subtrees. We claim that the total number of all these occurrences is only $\mathcal{O}(m/\ell)$. This follows from the following characterization.

Lemma 11. *If $P[x\,..\,(x+\ell)]$ is an occurrence of an anchor then either $P[(x+1)\,..\,y]$ is a maximal ℓ-periodic substring of P, for some $y \geq x+\ell$, or $P[x'\,..\,(x+\ell-1)]$ is a maximal ℓ-periodic substring of P, for some $x' \leq x$.*

Proof. By symmetry, it is enough to consider an anchor H created because of a maximal ℓ-periodic substring $S[i\,..\,j]$ such that $i > 1$, when we add $S[(i-1)\,..\,(i-1+\ell)]$ to $\mathcal{A}$. Thus, $\mathrm{per}(H[2\,..\,|H|]) \leq \ell/4$ and if $P[x\,..\,(x+\ell)] = H$ then $\mathrm{per}(P[(x+1)\,..\,(x+\ell)]) \leq \ell/4$, making $P[(x+1)\,..\,(x+\ell)]$ a substring of some maximal ℓ-periodic substring of $P[(x'+1)\,..\,y]$, where $x' \leq x$ and $y \geq x+\ell$. If $x' < x$ then $\mathrm{per}(H) \leq \ell/4$. But then $H = S[(i-1)\,..\,(i-1+\ell)]$ can be extended to some maximal ℓ-periodic substring $S[i'\,..\,j']$ such that $i' \leq i-1$ and $j' \geq i-1+\ell$. The overlap between $S[i\,..\,j]$ and $S[i'\,..\,j']$ is at least ℓ, so by Lemma 9 $i = i'$ and $j = j'$, which is a contradiction. Consequently, $x' = x$ and we obtain the lemma. □

By Lemma 11, the number of occurrences of all anchors in P is at most two per each maximal ℓ-periodic substrings, so $\mathcal{O}(m/\ell)$ in total. We thus obtain a set of length-$(\ell+1)$ anchors with the following properties:

1. The total number of occurrences of all anchors in P is $\mathcal{O}(m/\ell)$.
2. For every $S \in \mathcal{S}$, at least one of its length-$(\ell+1)$ substrings is an anchor.
3. For every $S \in \mathcal{S}$, at most two of its length-$(\ell+1)$ substrings are anchors.

These properties are even stronger than what we had used in Section 2.5.1 (except that now we are working with length-$(\ell+1)$ substrings, which is irrelevant), we can now prove our final result also for strings of type 2.

Theorem 7. *An instance of the AP problem where all strings are of type 2 can be solved in $\tilde{\mathcal{O}}(m^{\omega-1}+N)$ time.*

2.5.3 Type 3 Strings

In this section we show how to solve a restricted instance of the AP problem where every string $S \in \mathcal{S}$ is of type 3, that is, $\mathrm{per}(S) \leq \ell/4$. Furthermore $|S| \in [9/8 \cdot \ell, 5/4 \cdot \ell)$ for some $\ell \leq m$. Recall that strings $S \in \mathcal{S}$ are such that every length-ℓ substring of S is strongly periodic and, by Lemma 31, in this case, $\mathrm{per}(S) \leq \ell/4$. An occurrence of such S in P must be contained in a maximal ℓ-periodic substring of P. Recall that a string T is called ℓ-periodic if $|T| \geq \ell$ and $\mathrm{per}(T) \leq \ell/4$. For an ℓ-periodic string T, let its *root*, denoted by $\mathrm{root}(T)$, be the lexicographically smallest cyclic shift of $T[1\,..\,\mathrm{per}(T)]$. Because $\mathrm{per}(T) \leq \ell/4$ and $|T| \geq \ell$ by definition, there are at least four repetitions of the period in T, so we can write $T = R[i\,..\,|R|]R^{\alpha}R[1\,..\,j]$, where $R = \mathrm{root}(T)$, for some $i, j \in [1, |R|]$ and $\alpha \geq 2$. It is well known that $\mathrm{root}(T)$ can be computed in $\mathcal{O}(|T|)$ time [132].

Example 4. *Let* $T = \texttt{bababababab}$ *and* $\ell = 8$*. We have* $|T| = 9 \geq \ell = 8$ *and* $\mathrm{per}(T) = 2 \leq \ell/4 = 2$*, so* T *is* ℓ*-periodic. We have* $\mathrm{root}(T) = R = \texttt{ab}$*, and* T *can be written as* $T = \texttt{b} \cdot (\texttt{ab})^3 \cdot \texttt{ab}$*, for* $i = 2$ *and* $j = 2$*.*

We will now make a partition of type 3 strings based on their root. We start with extracting all maximal ℓ-periodic substrings of P using Lemma 10 and compute the root of every such substring. This can be done in $\mathcal{O}(m)$ total time because two maximal ℓ-periodic substrings cannot overlap by more than $\ell/2$ letters, and hence their total length is at most $3/2 \cdot \ell$. We also extract the root of every $S \in \mathcal{S}$ in $\mathcal{O}(N)$ total time. We then partition maximal ℓ-periodic substrings of P and strings $S \in \mathcal{S}$ into groups that have the same root. In the remaining part we describe how to process each such group corresponding to root R in which all maximal ℓ-periodic substrings of P have total length m', and the strings $S \in \mathcal{S}$ have total length N'.

Recall that the bit vector U stores the active prefixes input to the AP problem, and the bit vector V encodes the new active prefixes we aim to compute. For every maximal ℓ-periodic substring of P with root R we extract the corresponding fragment of the bit vector U and need to update the corresponding fragment of the bit vector V. To make the description less cluttered, we assume that each such substring of P is a power of R, that is, R^α for some $\alpha \geq 4$. This can be assumed without loss of generality as it can be ensured by appropriately padding the extracted fragment of U and then truncating the results, while increasing the total length of all considered substrings of P by at most half of their length. In the description below, for simplicity of presentation, U and V denote these padded fragments of the original U and V. When computing V from U we use two different methods for processing the elements $S = R[i\,..\,|R|]R^\beta R[1\,..\,j]$ of $\mathcal{S}$ depending on their length: either $\beta > \alpha/|R|$ (large β) or $\beta \leq \alpha/|R|$ (small β).

Large β. We proceed in phases corresponding to $\beta = \alpha/|R| + 1, \ldots, \alpha$. In each single phase, we consider all strings $S \in \mathcal{S}$ with $S = R[i\,..\,|R|]R^\beta R[1\,..\,j]$, for some i and j. Let $C(\beta)$ be the set of the corresponding pairs (i, j), and observe that $\sum_\beta |C(\beta)| \cdot |R^\beta| \leq N'$. This is because the length of R^β is not greater than that of $S = R[i\,..\,|R|]R^\beta R[1\,..\,j]$, there are $|C(\beta)|$ distinct strings of the latter form, and the total length of all $S \in \mathcal{S}$ is N'. The total number of occurrences of a string $S = R[i\,..\,|R|]R^\beta R[1\,..\,j]$ in R^α is bounded by $\mathcal{O}(\alpha)$, and all such occurrences can be generated in time proportional to their number. Thus, for every $(i, j) \in C(\beta)$, we can generate all occurrences of the corresponding string and appropriately update V in $\mathcal{O}(\alpha \cdot |C(\beta)|)$ total time.

Small β. We start by giving a technical lemma on the complexity of multiplying two $r \times r$ matrices whose cells are polynomials of degree up to d.

Lemma 12. *If two* $r \times r$ *matrices over* $\mathbb{Z}$ *can be multiplied in* $\mathcal{O}(r^\omega)$ *time, then two* $r \times r$ *matrices over* $\mathbb{Z}[X]$ *with degrees up to* d *can be multiplied in* $\tilde{\mathcal{O}}(dr^\omega)$ *time.*

Proof. Let A and B be two $r \times r$ matrices over $\mathbb{Z}[X]$ with degrees up to d. We reduce the product $A \cdot B = C$ to $2d$ products of $r \times r$ matrices over $\mathbb{Z}$ as follows. We evaluate the polynomials of each matrix in the complex $(2d)$th roots of unity: let A_i and B_i be the matrices obtained by evaluating the polynomials of A and B in the i-th such root, respectively. We then perform the $2d$ products $A_1 \cdot B_1, \ldots, A_{2d} \cdot B_{2d}$ to obtain matrices $C_1, \ldots, C_{2d}$: the $2d$ values $C_1[i, j], \ldots, C_{2d}[i, j]$ are finally interpolated to obtain the

coefficient representation of $C[i,j]$, for each $i,j=1,\ldots,r$, in $\mathcal{O}(d\log d)$ time for each polynomial [110]. Since we perform $2d$ products of matrices in $\mathbb{Z}^{r\times r}$, and we evaluate and interpolate r^2 polynomials of degree up to $2d$, the overall time complexity is $2d\mathcal{O}(r^\omega)+r^2\mathcal{O}(d\log d)=\tilde{\mathcal{O}}(dr^\omega)$. □

Unlike in the large β case, we process $\beta=2,\ldots,\alpha/|R|$ simultaneously as follows. For each β we construct an $|R|\times|R|$ matrix M_β, with $M_\beta[i,j]=1$ if and only if $(i,j)\in C(\beta)$ (and $M_\beta[i,j]=0$ otherwise), and collect them in a single 3D matrix $M\in\{0,1\}^{|R|\times|R|\times(\alpha/|R|-1)}$ with the third dimension corresponding to the value of β. We then create another $\alpha\times|R|$ matrix, denoted by M', by setting $M'[\gamma,i]=1$ if and only if $U[\gamma\cdot|R|+i-1]=1$ (and $M_\beta[i,j]=0$ otherwise). Observe that M' can be interpreted as a vector of length $|R|$ over $\mathbb{Z}[X]$ with degrees up to α, and M as an $|R|\times|R|$ matrix over $\mathbb{Z}[X]$ with degrees up to $\alpha/|R|$: in this way, x^γ appears with non-zero coefficient in the polynomial at $M'[i]$ if and only if $U[\gamma\cdot|R|+i-1]=1$, and x^β appears with non-zero coefficient in the polynomial at $M[i,j]$ if and only if $(i,j)\in C(\beta)$. M can be constructed in total $\mathcal{O}(N')$ time by first iterating over all $S\in\mathcal{S}$ and adding x^β to the polynomial at $M[i,j]$, where $S=R[i\,..\,|R|]R^\beta R[1\,..\,j]$, and then extracting a prefix of each polynomial consisting of monomials of degree less than $\alpha/|R|$.

The product $M'\cdot M=M''$ allows us to recover the updates to V by observing that $V[(q+1)\cdot|R|+j]=1$ if and only if x^q appears with non-zero coefficient in the polynomial at $M''[j]$. We aim at reducing this product to a matrix-matrix product over $\mathbb{Z}[X]$ with degrees up to $\alpha/|R|$, so as to compute it efficiently by applying Lemma 12.

The idea now is to decompose the columns of M' into $|R|$ chunks of size $\alpha/|R|$ in order to transform it into another 3D matrix. To this end, we transform M' into an $|R|\times|R|\times(\alpha/|R|)$ matrix A by setting

$$A[k,i,\gamma]=1\Leftrightarrow M'[(k-1)\alpha/|R|+\gamma,i]=1\Leftrightarrow U[(k-1)\alpha/|R|+\gamma+i-1]=1.$$

By interpreting A as an $|R|\times|R|$ matrix over $\mathbb{Z}[X]$ with degrees up to $\alpha/|R|$, and interpreting M' as a vector of length $|R|$ over $\mathbb{Z}[X]$ with degrees up to α, we have that the first row of A consists of the coefficients of $x^1,\ldots,x^{\alpha/|R|}$ of each of the $|R|$ polynomials of M', the second row consists of the coefficients of $x^{\alpha/|R|+1},\ldots,x^{2\alpha/|R|}$ of each of the $|R|$ polynomials of M', and so on. In general, $A[k,i]$ consists of the coefficients of $x^{(k-1)\alpha/|R|+1},\ldots,x^{k\alpha/|R|}$ of polynomial $M'[i]$.

The product $A\cdot M=C$ still allows us to recover the updates of V, by observing that $V[((k-1)\alpha/|R|+1+q+1)|R|+j]=1$ if x^q appears with non-zero coefficient in the polynomial at $C[k,j]$. This is because at row $A[k,\cdot]$ there are the coefficients that correspond to $U[\gamma\cdot|R|+i-1]$ for $\gamma=(k-1)\alpha/|R|+1,\ldots,k\alpha/|R|$ and $i=1,\ldots,|R|$, and hence a x^q appearing at $C[k,j]$ is equivalent to a $x^{q+(k-1)\alpha/|R|+1}$ at $M''[j]$.

We are now in a position to prove the following result for type 3 strings.

Theorem 8. *An instance of the AP problem where all strings are of type 3 can be solved in $\tilde{\mathcal{O}}(m^{\omega-1}+N)$ time.*

Proof. Recall that we consider strings S of type 3 with root R and substrings of P with root R together. We first analyze the time to process a single group containing a number of substrings of P of total length m' and a number of strings $S\in\mathcal{S}$ of total

length N'. Let us denote by R^{α_i} the i-th considered substring of P and further define $\alpha = \sum_i \alpha_i = m'/|R|$.

If $\beta > \alpha/|R|$ we use the first method and spend $\mathcal{O}(\alpha_i \cdot |C(\beta)|)$ time, where $C(\beta)$ is the set of (i,j) for this specific β. The overall time used for all applications of the first method is:

$$\begin{aligned}\sum_i \mathcal{O}(\alpha_i \cdot \sum_{\beta > \alpha/|R|} |C(\beta)|) &= \mathcal{O}(\alpha/|R^{\alpha/|R|}| \cdot \sum_{\beta > \alpha/|R|} |C(\beta)| \cdot |R^{\alpha/|R|}|) \\ &= \mathcal{O}(\sum_{\beta > \alpha/|R|} |C(\beta)| \cdot |R^{\beta}|) = \mathcal{O}(N'),\end{aligned}$$

using the fact that $\sum_\beta |C(\beta)| \cdot |R^\beta| \leq N'$ and $\alpha/|R^{\alpha/|R|}| = \alpha/(\alpha/|R|)|R| = \mathcal{O}(1)$.

For each α_i, we process together all $\beta \leq \alpha_i/|R|$ using the second method, and we need to multiply two $|R| \times |R|$ matrices of polynomials of degree up to $\alpha_i/|R|$, that we can build in time $\mathcal{O}(N')$ and multiply in time $\mathcal{O}(|R|^\omega \cdot \alpha_i/|R| + |R|^2(\alpha_i/|R|)\log(\alpha_i/|R|))$ by Lemma 12. The overall time used for all applications of the second method is:

$$\mathcal{O}(N') + \sum_i \mathcal{O}(|R|^\omega \cdot \alpha_i/|R| + |R|^2(\alpha_i/|R|)\log(\alpha_i/|R|)) = \mathcal{O}(|R|^{\omega-2}m' + m'\log m' + N'),$$

using the fact that $\alpha = m'/|R|$. Since $|R| \leq m'$, this is in fact $\mathcal{O}((m')^{\omega-1} + m'\log m' + N')$.

Because all values of N' sum up to N and all values of m' sum up to $\mathcal{O}(m)$, by convexity of $x^{\omega-1}$ we obtain that the overall time complexity is $\tilde{\mathcal{O}}(m^{\omega-1} + N)$. □

2.5.4 Wrapping Up

In Sections 2.5.1, 2.5.2 and 2.5.3 we design three $\tilde{\mathcal{O}}(m^{\omega-1} + N)$-time algorithms for an instance of the AP problem where all strings are of type 1, 2 and 3, respectively. We thus obtain Theorem 2, and using the fact that $\omega < 2.373$ [242, 362] we obtain the following corollary.

Corollary 9. *The EDSM problem can be solved on-line in $\mathcal{O}(nm^{1.373} + N)$ time.*

Note that the polylog factors are shaved from $\tilde{\mathcal{O}}(nm^{\omega-1} + N)$ by using the fact that the inequality of $\omega < 2.373$ is strict.

2.6 Final Remarks

Our contribution in this chapter is twofold. First, we designed an appropriate reduction showing that a combinatorial algorithm solving the EDSM problem in $\mathcal{O}(nm^{1.5-\epsilon} + N)$ time, for any $\epsilon > 0$, refutes the well-known BMM conjecture. Second, we designed a non-combinatorial $\tilde{\mathcal{O}}(nm^{\omega-1} + N)$ -time algorithm to attack the same problem. By using the fact that $\omega < 2.373$, our algorithm runs in $\mathcal{O}(nm^{1.373} + N)$ time thus breaking the combinatorial conditional lower bound for the EDSM problem. Let us point out that if $\omega = 2$ then our algorithm for the AP problem is time-optimal up to polylog factors.

Chapter 3

Approximate Pattern Matching on Elastic-Degenerate Text

Key Points

Problem. The computational task we consider here is to search on-line for a pattern in a collection of similar strings, typically closely related genomes, allowing a fixed amount of errors in the matches. A certain degree of approximation in the matching is needed to account for possible polymorphisms and sequencing errors, often occurring in real data.

Model. We study the problem of matching a pattern against an ED string (EDSM) on-line under the Hamming and the edit distance models, limiting the maximum number of allowed errors in both cases. Our solution relies on an ad-hoc modification of the Landau-Vishkin algorithm [240] for computing the edit distance.

Included Works

This chapter contains the results of the paper **Approximate Pattern Matching on Elastic-Degenerate Text** [60], published in the journal *Theoretical Computer Science*. This paper is the journal extension of **Pattern matching on Elastic-Degenerate Text with Errors** [59], which I presented at the *24th International Symposium on String Processing and Information Retrieval (SPIRE 2017)*.

3.1 Introduction

There is a growing interest in the notion of *pan-genome* [345]. In the last ten years, with faster and cheaper sequencing technologies, re-sequencing (that is, sequencing the

genome of yet another individual of a species) became more and more a common task in modern genome analysis workflows. By now, a huge amount of genomic variations within the same population has been detected (e.g., in humans for medical applications, but not only), and this is only the beginning. With this, new challenges of functional annotation and comparative analysis have been raised. Traditionally, a single annotated *reference genome* is used as a control sequence. The reference genome is a representative example of the genomic sequence of a species. It serves as a reference text to which, for example, fragments of newly sequenced genomes of individuals are mapped. Although a single reference genome provides a good approximation of any individual genome, in loci with polymorphic variations, mapping and sequence comparison often fail their purposes. This is where a multiple genome, i.e., a pan-genome, would be a better reference text [198].

In the literature, many different (compressed) representations and thus algorithms have been considered for pattern matching on a set of similar texts [65, 158, 287, 353, 159, 45, 234]. A natural representation of pan-genomes, or fragments of them, that we consider here are elastic-degenerate texts [205]. An *elastic-degenerate text* is a sequence which compactly represents a multiple alignment of several closely-related sequences. In this representation, substrings that match exactly are collapsed, while in positions where the sequences differ (by means of substitutions, insertions, and deletions of substrings), all possible variants observed at that location are listed in so-called *degenerate segments*. In the literature, other similar types of uncertain sequences have also been considered, namely *degenerate strings*, where each degenerate segment can contain only single letters (see [11, 116, 206, 294, 331] and references therein) and *generalised degenerate strings* [27], where each degenerate segment contains strings of the same length. Elastic-degenerate texts are more general than both the previous objects, and they correspond to the Variant Call Format (VCF), that is, the *standard* for storing gene sequence variations [344]. A tool for creating elastic-degenerate texts from VCF files was also made available in [299].

The natural problem that arises is to find all matches of a deterministic pattern P in an ED text $\tilde{T}$. We call this the Elastic-Degenerate String Matching (*EDSM*) problem. The simplest version of this problem assumes that a degenerate (sometimes called indeterminate) segment can contain only single letters [196].

Due to the application of cataloguing human genetic variation [344], there has been ample work in the literature on the *off-line* (indexing) version of the pattern matching problem [198, 304, 258, 327, 281]. The *on-line*, more fundamental, version of the *EDSM* problem has not been studied as much as indexing approaches. The motivation for considering the on-line version of the problem is to remove the hardship of building disk-based indexes or rebuilding them with every update in the sequences. Indexes are often unwieldy, take a lot of time and space to build, and require lots of disk space to be stored. Their usage is therefore only convenient when the data is static or changes very infrequently. Solutions to the on-line version can thus be beneficial for a number of reasons: **(a)** efficient on-line solutions can be used in combination with partial indexes as practical trade-offs; **(b)** efficient on-line solutions for exact pattern matching can be applied for fast average-case approximate pattern matching similar to standard strings [40]; **(c)** on-line solutions can be useful when one wants to search for a few patterns in many degenerate texts similar to standard strings

such as protein or DNA sequences [26]. A few results exist on the exact *EDSM* problem [205, 178, 31, 299, 104, 307], culminating in the result presented in Chapter 2.

Our Contribution. Since genomic sequences are endowed with polymorphisms and sequencing errors, the existence of an exact occurrence can result into a strong assumption. The aim of this work is to generalize the studies of [205] and [178] for the exact case, allowing some approximation in the occurrences of the input pattern. We suggest a simple on-line $\mathcal{O}(kmG + kN)$-time and $\mathcal{O}(m)$-space algorithm, G being the total number of strings in $\tilde{T}$ and $k > 0$ the maximum number of allowed substitutions in a pattern's occurrence, that is nonzero *Hamming distance*. Our main contribution is an on-line $\mathcal{O}(k^2mG + kN)$-time and $\mathcal{O}(m)$-space algorithm where the type of edit operations allowed is extended to insertions and deletions as well, that is nonzero *edit distance*. These results are *good* in the sense that for *small* values of k the algorithms incur (essentially) no increase in time complexity with respect to the $\mathcal{O}(nm^2 + N)$-time and $\mathcal{O}(m)$-space algorithm presented in [178] for the exact case.

Structure of the Chapter. Section 3.2 provides some preliminary definitions and facts as well as the formal statements of the problems we address. Section 3.3 describes our solution for constant-sized alphabets under the edit distance model, while Section 3.4 describes the algorithm under the Hamming distance model for constant-sized alphabets. Section 3.5 extends these algorithms to work for general integer alphabets. We conclude in Section 3.6.

3.2 Preliminaries

For convenience, we start by recalling the definition of elastic-degenerate string and the notation we adopt. An *elastic-degenerate (ED) string* of *length* n on alphabet Σ, $\tilde{T} = \tilde{T}[0]\tilde{T}[1]\dots\tilde{T}[n-1]$, is a finite sequence of n degenerate letters. Every *degenerate letter* $\tilde{T}[i]$ is a finite non-empty set of strings $\tilde{T}[i][j] \in \Sigma^*$, with $0 \leq j < |\tilde{T}[i]|$. The *size* N of $\tilde{T}$ is defined as

$$N = \sum_{i=0}^{n-1} \sum_{j=0}^{|\tilde{T}[i]|-1} |\tilde{T}[i][j]|$$

assuming (for representation purposes only) that $|\varepsilon|$=1. The *total number of strings* in $\tilde{T}$ is defined as $G = \sum_{i=0}^{n-1} |\tilde{T}[i]|$. Notice that $n \leq G \leq N$.

In Chapter 2 we gave a definition of an exact match between a deterministic string P and an ED string $\tilde{T}$; here we extend such definition to deal with errors.

Definition 1. *Given an integer $k > 0$, we say that a string $P \in \Sigma^m$ k_H-matches (resp. k_E-) an ED string $\tilde{T} = \tilde{T}[0]\tilde{T}[1]\dots\tilde{T}[n-1]$ of length $n > 1$ if all of the following hold:*

- *there exists a non-empty suffix X of some string $S \in \tilde{T}[0]$;*
- *if $n > 2$, there exist strings $Y_1 \in \tilde{T}[1], \dots, Y_t \in \tilde{T}[t]$, for $1 \leq t \leq n-2$;*
- *there exists a non-empty prefix Z of some string $S \in \tilde{T}[n-1]$;*
- *the Hamming (resp. edit) distance between P and $XY_1\dots Y_tZ$ (note that $Y_1\dots Y_t$ can be ε) is no more than k.*

We say that P has a *k_H-occurrence* (resp. *k_E-*) ending at position j in $\tilde{T}$ if either there exists a k_H-match (resp. k_E-) between P and $\tilde{T}[i]\dots\tilde{T}[j]$ for some $0 \le i < j \le n-1$ or P is at Hamming (resp. edit) distance of at most k from a substring of some string $S \in \tilde{T}[j]$. We say that P has a *partial* k_H-occurrence (resp. k_E-) $P[0\dots\ell]$, for some $\ell < m-1$, ending at position j of $\tilde{T}$ if $P[0\dots\ell]$ k_H-matches (resp. k_E-) $\tilde{T}[i]\dots\tilde{T}[j]$, for some $0 \le i \le j$.

Example 5 (Running example)**.** *Consider $P = \texttt{GAACAA}$ of length $m = 6$. The following ED string has $n = 7$, $N = 20$, and $G = 12$. An 1_H-occurrence of P is underlined, and an 1_E-occurrences of P is overlined.*

$$\tilde{T} = \{\overline{\underline{\texttt{G}}}\} \cdot \left\{\begin{matrix}\overline{\underline{\texttt{AA}}}\\ \texttt{AG}\\ \overline{\underline{\varepsilon}}\end{matrix}\right\} \cdot \{\overline{\underline{\texttt{A}}}\} \cdot \left\{\begin{matrix}\overline{\underline{\texttt{CAA}}}\\ \texttt{GTG}\\ \underline{\texttt{AC}}\end{matrix}\right\} \cdot \{\underline{\texttt{A}}\} \cdot \left\{\begin{matrix}\texttt{A}\\ \underline{\varepsilon}\end{matrix}\right\} \cdot \{\underline{\texttt{C}}\texttt{A}\}$$

Given two strings X and Y and a pair (i, j), with $0 \le i \le |X|-1$ and $0 \le j \le |Y|-1$, the *longest common extension* at (i, j), denoted by $lce_{X,Y}(i, j)$, is the length of the longest substring of X starting at position i that matches a substring of Y starting at position j. For instance, when $X = \texttt{CGCGT}$ and $Y = \texttt{ACG}$, $lce_{X,Y}(2, 1) = 2$, corresponding to the substring $\texttt{CG}$. We define $lce_{X,Y}(i, j) = 0$ when either $i \notin \{0, 1, \dots, |X| - 1\}$ or $j \notin \{0, 1, \dots, |Y| - 1\}$.

Fact 10 ([184])**.** *Given a string X, its suffix tree ST_X, and a set of strings $W = \{Y_1, \dots, Y_l\}$ over a constant-sized alphabet, it is possible to build the generalized suffix tree $ST_{X,W}$ extending ST_X, in time $\mathcal{O}(\sum_{h=1}^{l} |Y_h|)$. Moreover, given two strings X and Y of total length q, for each index pair (i, j), $lce_{X,Y}(i,j)$ queries can be computed in constant time per query, after a pre-processing of $ST_{X,Y}$ that takes time and space $\mathcal{O}(q)$.*

We will denote by $ST^*_{X,Y}$ such a pre-processed tree for answering *lce* queries. The time is ripe now to formally introduce the two problems considered here.

[k_E-**EDSM**] Elastic-Degenerate String Matching under the edit distance model:

Input: A deterministic pattern P of length m, an ED text $\tilde{T}$ of length n and size $N \ge m$, an integer $0 < k < m$.

Output: Pairs (i, d), i being a position in $\tilde{T}$ where at least one k_E-occurrence of P ends and $d \le k$ being the minimal number of errors (substitutions, insertions and deletions) for occurrence i.

[k_H-**EDSM**] Elastic-Degenerate String Matching under the Hamming distance model:

Input: A deterministic pattern P of length m, an ED text $\tilde{T}$ of length n and size $N \ge m$, an integer $0 < k < m$.

Output: Pairs (i, d), i being a position in $\tilde{T}$ where at least one k_H-occurrence of P ends and $d \le k$ being the minimal number of substitutions for occurrence i.

3.3 An Algorithm for k_E-EDSM

In Chapter 2 we showed that the exact *EDSM* problem (that is, for $k = 0$) cannot be solved $\mathcal{O}(nm^{1.5-\epsilon} + N)$ time with a combinatorial algorithm unless there exists a truly subcubic combinatorial algorithm for Triangle Detection, and showed a non-combinatorial algorithm requiring $\tilde{\mathcal{O}}(nm^{\omega-1} + N)$ time, where ω is the matrix multiplication exponent. Allowing up to k substitutions, insertions, and deletions in the occurrences clearly entails a time-cost increase. The solution proposed here manages to keep the time-cost growth limited, solving the k_E-EDSM problem in time $\mathcal{O}(k^2mG+kN)$, G being the total number of strings in the ED text. We assume a constant-sized alphabet. At a high level, the k_E-EDSM algorithm (pseudocode shown below) works as follows.

Pre-processing phase: build the suffix tree for the pattern P.

Searching phase: in an on-line manner, scan the text $\tilde{T}$ from left to right and, for each $\tilde{T}[i]$:

(1) Find the prefixes of P that are at edit distance at most k from any suffix of some $S \in \tilde{T}[i]$; if there exists an $S \in \tilde{T}[i]$ that is long enough, also search for k_E-occurrences of P that start and end at position i (lines 3 and 13 of the pseudocode)

(2) Try to extend at $\tilde{T}[i]$ each partial k_E-occurrence of P which has started earlier in $\tilde{T}$ (line 19)

(3) In both previous cases, if a full k_E-occurrence of P also ends at $\tilde{T}[i]$, then output position i; otherwise store the prefixes of P extended at $\tilde{T}[i]$ (lines 4-7, 14-17, 20-28)

Step (1) of algorithm k_E-EDSM is implemented by algorithm k_E-BORD described in Section 3.3.1. Step (2) is implemented by algorithm k_E-EXT described in Section 3.3.2.

The following lemma follows directly from Fact 10.

Lemma 13. *Given P of length m and $\tilde{T}$ of length n and size N, to build $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, requires total time $\mathcal{O}(N)$.*

Besides ST_P (built once as a pre-processing step) and $ST^*_{P,\tilde{T}[i]}$ (built for all $\tilde{T}[i]$'s), the algorithm uses the following data structures:

L' - a list that temporarily stores the output of functions k_E-BORD and k_E-EXT. It is re-initialized to $\emptyset$ (lines 3, 13 and 19) for each $S \in \tilde{T}[i]$ before executing either of the functions.

V_c - a vector of size $|P|$ such that $V_c[j]$ contains the lowest number of errors for a partial k_E-occurrence of $P[0 \dots j]$ ending at $\tilde{T}[i]$. For each pair position - edit operations (j, d) in L', if $V_c[j] < d$ then $V_c[j]$ is updated with d by the function INSERT. V_c (c stands for *current*) is re-initialized to $V_c[j] = \infty$ (line 11) for all j's each time a new degenerate segment $\tilde{T}[i]$ is read: $V_c[j] = \infty$ denotes that a partial k_E-occurrence of $P[0 \dots j]$ ending at $\tilde{T}[i]$ has not yet been found.

k_E-EDSM(P,ST_P,$\tilde{T}$,n,k)

$V_c[0 \ldots |P|-1] \leftarrow \infty$; Build $ST^*_{P,\tilde{T}[0]}$;
forall $S \in \tilde{T}[0]$ **do**
  $L' \leftarrow \emptyset$; $L' \leftarrow k_E$-BORD$(P, S, ST^*_{P,\tilde{T}[0]}, k)$;
  forall $(j, d) \in L'$ **do**
    if $j = |P| - 1$ **then**
      if $d < V_c[|P|-1]$ **then** $V_c[|P|-1] \leftarrow d$;
    else INSERT(L_c,(j, d),V_c);
if $V_c[|P|-1] \neq \infty$ **then report** $(0, V_c[|P|-1])$;
for $i = 1$ **to** $n-1$ **do**
  $L_p \leftarrow L_c$; $L_c \leftarrow \emptyset$; $V_p \leftarrow V_c$;
  $V_c[0 \ldots |P|-1] \leftarrow \infty$; Build $ST^*_{P,\tilde{T}[i]}$;
  forall $S \in \tilde{T}[i]$ **do**
    $L' \leftarrow \emptyset$; $L' \leftarrow k_E$-BORD$(P, S, ST^*_{P,\tilde{T}[i]}, k)$;
    forall $(j, d) \in L'$ **do**
      if $j = |P| - 1$ **then**
        if $d < V_c[|P|-1]$ **then** $V_c[|P|-1] \leftarrow d$;
      else INSERT(L_c,(j, d),V_c);
    forall $p \in L_p$ **do**
      $L' \leftarrow \emptyset$; $L' \leftarrow k_E$-EXT$(p+1$,P,S,$ST^*_{P,\tilde{T}[i]}$,$k - V_p[p])$;
      forall $(j, d) \in L'$ **do**
        if $j = |P| - 1$ **then**
          if $d + V_p[p] < V_c[|P|-1]$ **then** $V_c[|P|-1] \leftarrow d + V_p[p]$;
        else INSERT(L_c,$(j, d + V_p[p])$, V_c);
  if $V_c[|P|-1] \neq \infty \vee V_p[|P|-1] < k$ **then**
    report $(i, \min\{V_c[|P|-1], V_p[|P|-1]+1\})$
  if $V_p[|P|-1] + \min_{S \in \tilde{T}[i]} |S| < k$ **then**
    if $V_p[|P|-1] + \min_{S \in \tilde{T}[i]} |S| < V_c[|P|-1]$ **then**
      $V_c[|P|-1] \leftarrow V_p[|P|-1] + \min_{S \in \tilde{T}[i]} |S|$

L_c - a list that contains the rightmost position of the prefixes of P found in L'. It is filled in by function INSERT for each prefix $P[0 \dots j]$ where $V_c[j]$ turns into a value $\neq \infty$. Before reading a new degenerate segment $\tilde{T}[i]$, it is copied into L_p and re-initialized to $\emptyset$ (line 10).

L_p - a list where the L_c list, filled in at iteration $i-1$, is copied at the beginning of each iteration i (line 10). L_p thus stores prefixes of P found in L' during the previous iteration (p stands for *previous*).

V_p - similarly, V_p stores a copy of the vector V_c of the previous position.

Algorithm k_E-EDSM needs to report each position i in $\tilde{T}$ where some k_E-occurrence of P ends with edit distance $d \leq k$, d being the minimal such value for position i. To this aim, the last position of V_c is updated with the following criterion: each time we find an occurrence of P ending at $\tilde{T}[i]$, corresponding to pair $(m-1, d)$, if $V_c[m-1] > d$ then we set $V_c[m-1] = d$ (lines 6, 16 and 22). After all $S \in \tilde{T}[i]$ have been examined, if either $V_c[m-1] \neq \infty$ or $V_p[m-1] < k$ (i.e., an occurrence of P at the previous position implies an occurrence at the current one by deleting a letter in any $S \in \tilde{T}[i]$: see Example 6) the algorithm outputs position i together with the minimum between $V_c[m-1]$ and $V_p[m-1]+1$ (lines 24-25). If an occurrence of P at $i-1$ can lead to an occurrence at $i+1$ by deleting a whole string $S \in \tilde{T}[i]$ and a letter of any string in $\tilde{T}[i+1]$, i.e., if $V_p[m-1] + \min_{S \in \tilde{T}[i]} |S| < k$, and if this value is smaller than $V_c[m-1]$, it eventually updates $V_c[m-1]$ (lines 26-28).

Example 6 (Running example)**.** *Consider text $\tilde{T}$ and pattern $P =$ `GAACAA` of Example 5. The k_E-occurrence of P beginning at position 0 and ending at position 5 of $\tilde{T}$ with edit distance 0 implies an occurrence of P ending at position 6 with 1 deletion (namely, letter `C`).*

3.3.1 Algorithm k_E-BORD

For each i and for each $S \in \tilde{T}[i]$, Step (1) of the algorithm finds the prefixes of P that are at distance at most k from any suffix of S, as well as k_E-occurrences of P that start and end at position i if S is long enough. To this end, we use and modify the Landau-Vishkin algorithm [240]. We first recall some relevant definitions concerning the dynamic programming table [184].

Given an $m \times q$ dynamic programming table (m rows, q columns), the *main diagonal* consists of cells (h, h) for $0 \leq h \leq \min\{m-1, q-1\}$. The diagonals above the main diagonal are numbered 1 through $(q-1)$; the diagonal starting in cell $(0, h)$ is diagonal h. The diagonals below the main one are numbered -1 through $-(m-1)$; the diagonal starting in cell $(h, 0)$ is numbered $-h$. A *d-path* in the dynamic programming table is a path that starts in row zero and specifies a total of exactly d edit operations (substitutions, insertions, and deletions). A d-path is *farthest reaching in diagonal h* if it is a d-path that ends in diagonal h and the index of its ending column c is $\geq$ to the ending column of any other d-path ending in diagonal h.

Algorithm k_E-BORD takes as input a pattern P, a string $S \in \tilde{T}[i]$, $ST^*_{P,\tilde{T}[i]}$ and the upper bound k for edit distance; it outputs pairs (j, d), where j is the rightmost position

of the prefix of P that is at distance $d \leq k$ from a suffix of S, with the minimal value of d reported for each j. In order to fulfill this task, at a high level, the algorithm executes the following steps on a table having P at the rows and S at the columns:

(1a) For each diagonal $0 \leq h \leq |S| - 1$ it finds $lce_{P,S}(0, h)$. This specifies the end column of the farthest reaching 0-path on each diagonal from 0 to $|S| - 1$.

(1b) For each $1 \leq d \leq k$, it finds the farthest reaching d-path on diagonal h, for each $-d \leq h \leq |S| - 1$. This path is derived from the farthest reaching $(d-1)$-paths on diagonals $(h-1)$, h and $(h+1)$.

(1c) Any d-path that reaches the last row of the dynamic programming table indicates a k_E-occurrence of P with edit distance d that starts and ends at position i, thus the algorithm reports $(|P| - 1, d)$; any d-path that reaches the end of S in row r denotes that the prefix of P ending at $P[r]$ is at distance d from a suffix of S, and the algorithm reports (r, d).

In Step (1b), the farthest reaching d-path on diagonal h is found by computing and comparing the following three particular paths that end on diagonal h:

R_i - consists of the farthest reaching $(d-1)$-path on diagonal $h+1$, followed by a vertical edge to diagonal h, and then by the maximal extension along diagonal h that corresponds to identical substrings. Function R_i takes as input the length $|X|$ of a string X, whose letters spell the rows of the dynamic programming table, the length $|Y|$ of a string Y, whose letters spell the columns, $ST^*_{X,Y}$ and the pair row-column (r, c) where the farthest reaching $(d-1)$-path on diagonal $h+1$ ends. It outputs pair (r_i, c_i) where path R_i ends. This path represents a letter insertion in X.

R_d - consists of the dual case of R_i with a horizontal edge representing a letter deletion in X.

R_s - consists of the farthest reaching $(d-1)$-path on diagonal h followed by a diagonal edge, and then by the maximal extension along diagonal h that corresponds to identical substrings. Function R_s takes as input the length $|X|$ of a string X, whose letters spell the rows of the dynamic programming table, the length $|Y|$ of a string Y, whose letters spell the columns, $ST^*_{X,Y}$ and the pair row-column (r, c) where the farthest reaching $(d-1)$-path on diagonal h ends. It outputs pair (r_s, c_s) where path R_s ends. This path represents a letter substitution.

All such functions output $(-\infty, -\infty)$ if it is not possible to derive a path from the given parameters (e.g., if r or c exceed the input dimension).

Fact 11 ([184]). *The farthest reaching path on diagonal h is the path among R_i, R_d or R_s that extends the farthest along h.*

In each one of the iterations in k_E-BORD, a diagonal h is associated with two variables $F_p(h)$ and $F_c(h)$, storing the column reached by the farthest reaching path (FRP) on h in the previous and in the current iteration, respectively. We define $F_p(h) = F_c(h) = -\infty$ when $h \notin \{-(|P| - 1), \ldots, |S| - 1\}$. Notice that at most $k + |S|$ diagonals will be

INSERT(L,(j,d),V)

```
1 if V[j] > d then
2     if V[j] = ∞ then Insert j in L;
3     V[j] ← d;
```

$R_i(|X|, |Y|, ST^*_{X,Y}, r, c)$

```
1 if −1 ≤ r ≤ |X| − 2 ∧ −1 ≤ c ≤ |Y| − 1 then
2     ℓ ← lce_{X,Y}(r + 2, c + 1);
3     r_i ← r + 1 + ℓ;
4     c_i ← c + ℓ;
5     return (r_i, c_i)
6 else return (−∞, −∞);
```

$R_d(|X|, |Y|, ST^*_{X,Y}, r, c)$

```
1 if −1 ≤ r ≤ |X| − 1 ∧ −1 ≤ c ≤ |Y| − 2 then
2     ℓ ← lce_{X,Y}(r + 1, c + 2);
3     r_d ← r + ℓ;
4     c_d ← c + 1 + ℓ;
5     return (r_d, c_d)
6 else return (−∞, −∞);
```

$R_s(|X|, |Y|, ST^*_{X,Y}, r, c)$

```
1 if −1 ≤ r ≤ |X| − 2 ∧ −1 ≤ c ≤ |Y| − 2 then
2     ℓ ← lce_{X,Y}(r + 2, c + 2);
3     r_s ← r + 1 + ℓ;
4     c_s ← c + 1 + ℓ;
5     return (r_s, c_s)
6 else return (−∞, −∞);
```

taken into account: the algorithm first finds the *lce*'s between $P[0]$ and $S[j]$, for all $0 \le j \le |S| - 1$, and hence it initializes $|S|$ diagonals; after this, for each successive step (there are at most k of them), it widens to the left one diagonal at a time because an initial deletion can be added; therefore, it will consider at most $k + |S|$ diagonals. The only difference between algorithm k_E-BORD and the algorithm by Landau and Vishkin [240] is that k_E-BORD outputs pairs (ℓ, d) corresponding to FRPs that reach the last *column* of the DP table, in addition to the ones corresponding to FRPs that reach the last row. By construction, these additional pairs correspond to k_E-matches between prefixes of P and suffixes of S. The correctness of the Landau-Vishkin algorithm thus directly implies the following lemma:

Lemma 14. *Algorithm k_E-BORD is correct.*

The next lemma provides the time complexity of applying k_E-BORD to every $S \in \tilde{T}[i]$, for all $i = 0, \ldots, n-1$.

Lemma 15. *Given P of length m, $\tilde{T}$ of length n and size N, $ST^*_{P,\tilde{T}[i]}$ for all $i \in [0, n-1]$, and an integer $0 < k < m$, k_E-BORD finds the minimal edit distance $\le k$ between the prefixes of P and any suffix of $S \in \tilde{T}[i]$, as well as the k_E-occurrences of P that start and end at position i, in time $\mathcal{O}(k^2 G + kN)$, G being the total number of strings in $\tilde{T}$.*

Proof. For a string $S \in \tilde{T}[i]$, for each $0 \le d \le k$ and each diagonal $-k \le h \le |S| - 1$, the k_E-BORD algorithm retrieves the end of three $(d-1)$-paths (constant-time operations) and computes the path extension along the diagonal via a constant-time *lce* query (Fact 10). It thus takes time $\mathcal{O}(k^2 + k|S|)$ to find the prefixes of P that are at distance at most k from any suffix of S; the k_E-occurrences of P that start and end at position i are computed within the same complexity. The total time is $\mathcal{O}(k^2|\tilde{T}[i]| + k\sum_{j=0}^{|\tilde{T}[i]|-1} |S|)$,

k_E-BORD$(P, S, ST^*_{P,\tilde{T}[i]}, k)$

for $h = -(k+1)$ **to** -1 **do** $F_c(h) \leftarrow h - 1$;
for $h = 0$ **to** $|S| - 1$ **do**
  $\ell \leftarrow lce_{P,S}(0, h)$;
  $F_c(h) \leftarrow \ell - 1 + h$;
  if $\ell + h = |S|$ **then report** $(\ell - 1, 0)$;
  else
    if $\ell = |P|$ **then report** $(|P| - 1, 0)$;
for $d = 1$ **to** k **do**
  for $h = -(k+1)$ **to** $|S| - 1$ **do** $F_p(h) \leftarrow F_c(h)$;
  for $h = -d$ **to** $|S| - 1$ **do**
    $(r_i, c_i) \leftarrow R_i(|P|, |S|, ST^*_{P,\tilde{T}[i]}, F_p(h+1) - (h+1), F_p(h+1))$;
    $(r_d, c_d) \leftarrow R_d(|P|, |S|, ST^*_{P,\tilde{T}[i]}, F_p(h-1) - (h-1), F_p(h-1))$;
    $(r_s, c_s) \leftarrow R_s(|P|, |S|, ST^*_{P,\tilde{T}[i]}, F_p(h) - h, F_p(h))$;
    if $\max\{c_i, c_d, c_s\} > -\infty$ **then** $F_c(h) \leftarrow \max\{c_i, c_d, c_s\}$;
    else $F_c(h) \leftarrow F_p(h)$;
    if $\max\{r_i, r_d, r_s\} = |P| - 1$ **then report** $(|P| - 1, d)$;
    if $\max\{c_i, c_d, c_s\} = |S| - 1$ **then report** $(|S| - 1 - h, d)$;

for all $S \in \tilde{T}[i]$. Since the size of $\tilde{T}$ is N and the total number of strings in $\tilde{T}$ is G, the result follows. □

Example 7 (Running example)**.** *Let us consider again text $\tilde{T}$ and pattern $P =$ `GAACAA` of Example 5, and let $k = 1$. Suppose we already executed iteration 0, and we move to position $i = 1$, where we need to find the suffixes of all $S \in \tilde{T}[1]$ that are at edit distance at most 1 from some prefix of P. Consider then $S =$ `AA` $\in \tilde{T}[1]$. The borders at edit distance 1 are the following:*

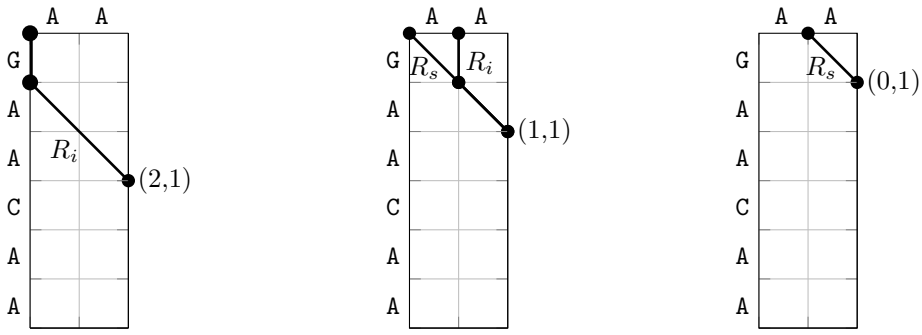
P:	GAACAA	GAACAA	GAACAA
S:	-AA	AA	AA
Output:	$(2, 1)$	$(1, 1)$	$(0, 1)$

To find them, Algorithm k_E-BORD$(P, S, ST^*_{P,\tilde{T}[1]}, 1)$ executes the following steps:

$d = 0$**:** find 0-paths on diagonals 0, 1 via *lce* queries. $lce_{P,S}(0, 0) = lce_{P,S}(0, 1) = 0$, thus $F_c(-2) = -3$, $F_c(-1) = -2$, $F_c(0) = -1$, $F_c(1) = 0$.

$d = 1$**:** compute farthest reaching 1-paths for diagonals -1, 0, 1 with $F_p(-2) = -3$, $F_p(-1) = -2$, $F_p(0) = -1$ and $F_p(1) = 0$ (Figure 3.1).

This results in $L_c = \{0, 1, 2\}$ and $V_c = [1, 1, 1, \infty, \infty, \infty]$.



(a) **(b)** **(c)**

Figure 3.1: (3.1a) diagonal $h = -1$: $R_i(|P|, |S|, ST^*_{P,S}, -1, -1)$ returns $(r_i, c_i) = (2, 1)$ (as $lce_{P,S}(1, 0) = 2$), hence $F_c(-1) = 1$, the path exhausts S and k_E-BORD returns pair $(2, 1)$. (3.1b) diagonal 0: both $R_i(|P|, |S|, ST^*_{P,S}, -1, 0)$ and $R_s(|P|, |S|, ST^*_{P,S}, -1, -1)$ return $(1, 1)$ (as $lce_{P,S}(1, 1) = 1$), thus they reach the end of S and k_E-BORD returns pair $(1, 1)$. (3.1c) diagonal 1: $R_s(|P|, |S|, ST^*_{P,S}, -1, 0)$ returns $(0, 1)$ (as $lce_{P,S}(1, 2) = 0$), the path consumes the whole S and k_E-BORD returns pair $(0, 1)$.

3.3.2 Algorithm k_E-EXT

In Step (2), algorithm k_E-EDSM tries to extend each partial k_E-occurrence that has started earlier in $\tilde{T}$. That is, at position i, for each $p \in L_p$ and for each string $S \in \tilde{T}[i]$, we try to extend $P[0 \dots p]$ with S. Once again, we modify the Landau-Vishkin algorithm [240] to our purpose: it suffices to look for the FRPs starting at the desired position only.

k_E-EXT takes as input a pattern P, a string $S \in \tilde{T}[i]$, the $ST^*_{P,\tilde{T}[i]}$, the upper bound k for edit distance and the position j in P where the extension should start; it outputs a list of distinct pairs (h, d), where h is the index of P where the extension ends, and d is the minimum additional number of edit operations introduced by the extension. Algorithm k_E-EXT performs a task similar to that of k_E-BORD: (i) it builds a $|S| \times |P|$ DP table (rather than a $|P| \times |S|$ table) and (ii) instead of searching for occurrences of P starting anywhere within S, k_E-EXT checks whether the whole of S can extend the prefix $P[0 \dots j-1]$ detected at the previous text position or whether a prefix of S matches the suffix of P starting at $P[j]$ (and hence a k_E-occurrence of P has been found). In order to fulfill this task, at a high level, the algorithm executes the following steps on a table having S at the rows and P at the columns:

(2a) It finds $lce_{S,P}(0, j)$ specifying the end column of the farthest reaching 0-path on diagonal j. The value of the end column of the farthest reaching 0-path for the rest of the diagonals from $j - (k + 1)$ to $j + k + 1$ is set to $-\infty$ by default. This initialization ensures that any FRP on the other diagonals will originate from diagonal j.

(2b) For each $1 \leq d \leq k$, it finds the farthest reaching d-path on diagonal h, for each $j - d \leq h \leq j + d$. This path is found from the farthest reaching $(d-1)$-paths on diagonals $(h-1)$, h and $(h+1)$.

k_E-EXT$(j, P, S, ST^*_{P,\tilde{T}[i]}, k)$

1 **if** $S = \varepsilon$ **then**
2 **for** $d = 0$ **to** k **do report** $(j + d, d)$;
3 **else**
4 **for** $h = j - (k + 1)$ **to** $j + k + 1$ **do** $F_c(h) \leftarrow -\infty$;
5 $\ell \leftarrow lce_{S,P}(0, j)$;
6 $F_c(j) \leftarrow \ell - 1 + j$;
7 **if** $\ell = |S|$ **then report** $(\ell + j - 1, 0)$;
8 **for** $d = 1$ **to** k **do**
9 **for** $h = j - (k + 1)$ **to** $j + k + 1$ **do** $F_p(h) \leftarrow F_c(h)$;
10 **for** $h = j - d$ **to** $j + d$ **do**
11 $(r_i, c_i) \leftarrow R_i(|S|, |P|, ST^*_{P,\tilde{T}[i]}, F_p(h + 1) - (h + 1), F_p(h + 1))$;
12 $(r_d, c_d) \leftarrow R_d(|S|, |P|, ST^*_{P,\tilde{T}[i]}, F_p(h - 1) - (h - 1), F_p(h - 1))$;
13 $(r_s, c_s) \leftarrow R_s(|S|, |P|, ST^*_{P,\tilde{T}[i]}, F_p(h) - h, F_p(h))$;
14 **if** $\max\{c_i, c_d, c_s\} > -\infty$ **then** $F_c(h) \leftarrow \max\{c_i, c_d, c_s\}$;
15 **else** $F_c(h) \leftarrow F_p(h)$;
16 **if** $\max\{r_i, r_d, r_s\} = |S| - 1$ **then report** $(F_c(h), d)$;
17 **if** $\max\{c_i, c_d, c_s\} = |P| - 1$ **then report** $(|P| - 1, d)$;

(2c) Any d-path that reaches the last row of the dynamic programming table in column c denotes an occurrence of the whole S with edit distance d, and the algorithm reports (c, d), c being the position in P where this extension ends; any d-path that reaches the end of P denotes that a prefix of S is at distance d from a suffix of P starting at position j, and the algorithm reports $(|P| - 1, d)$.

Example 8 (Running example)**.** *Let us continue our running example with pattern* P=`GAACAA` *and text* $\tilde{T}$ *of Example 5; let again* $k = 1$*, and let us consider* $i = 1$*. After computing the borders as hinted in Example 7, we need to extend previous partial* k_E*-occurrences of* P *with the strings in* $\tilde{T}[1]$*. Consider thus* S =`AA` $\in \tilde{T}[1]$*,* $L_p = \{0, 1\}$*,* $V_p = [0, 1, \infty, \infty, \infty, \infty]$*. We try to extend* $P[0]$ *with* S *and up to* $k - V_p[0] = 1$ *extra errors.* k_E-EXT$(1, P, S, ST^*_{P,\tilde{T}[1]}, 1)$ *performs the following steps:*

$d = 0$: *find a 0-path on diagonal* $j = 1$*: Since* $lce_{S,P}(0, 1) = 2$*, the value* $F_c(1) = 2$ *is updated and the algorithm reports pair* $(2, 0)$ *(see Figure 3.2a). The value of the rest of* $F_c(h)$ *for* h *from* $j - (k + 1) = -1$ *to* $j + k + 1 = 3$ *is set to* $-\infty$ *by default.*

$d = 1$: *compute FRPs for diagonals* $j - d = 0, j = 1, j + d = 2$*. Since the 0-FRP on diagonal* 1 *reaches the last row of the DP table already, it is not possible to extend it to 1-paths on lower diagonals: indeed,* R_i*,* R_d *and* R_s *all return* $(-\infty, -\infty)$ *for both diagonals 0 and 1. It is possible to extend it to a 1-path on diagonal 2 though, as shown in Figure 3.2b.*

This results in $L_c = \{0, 1, 2, 3\}$ *and* $V_c = [1, 1, 0, 1, \infty, \infty]$.

It is easy to see that the correctness of the Landau-Vishkin algorithm directly implies the correctness of k_E-EXT, providing the following lemma.

Lemma 16. *Algorithm k_E-EXT is correct.*

(a) **(b)**

Figure 3.2: (3.2a) diagonal $j = 1$: $lce_{S,P}(0, 1) = 2$, thus the 0-FRP reaches the last row of the table and k_E-EXT correctly returns pair $(2, 0)$.
(3.2b) diagonal $j + d = 2$: $R_d(|S|, |P|, ST^*_{P,S}, 1, 2)$ returns $(r_d, c_d) = (1, 3)$ (as $lce_{S,P}(2, 4) = 0$): since $r_d = |S| - 1$, this path reaches the last row of the DP table, and k_E-EXT correctly returns pair $(3, 1)$.

Lemma 17. *Given a prefix of P, a string $S \in \tilde{T}[i]$, $ST^*_{P,\tilde{T}[i]}$, and an integer $0 < k < m$, k_E-EXT extends the prefix of P with S in time $\mathcal{O}(k^2)$.*

Proof. The k_E-EXT algorithm does k iterations: at iteration d, for each diagonal $-d \leq h \leq d$, the end of three paths must be retrieved (constant-time operations) and the path extension along diagonal h must be computed via a constant-time *lce* query (Fact 10). The overall time for the extension is then bounded by $\mathcal{O}(1+3+\cdots+(2k+1))=\mathcal{O}(k^2)$. □

Correctness As for the correctness of algorithm k_E-EDSM, Lemmas 14 and 16 ensure that borders and extensions are correctly computed; we further observe that, by storing just the minimum edit distance for every partial k_E-occurrence of P at a certain position $\tilde{T}[i]$, we do not miss any occurrence of P nor report spurious occurrences. It is easy to find examples where, should we store a single value different from the minimum, we would either fail to report an occurrence (in case we stored a greater value), or report a spurious one (if we stored a lower value). On the other hand, any additional distance value beyond the minimum would be redundant according to the following observation: assume $P[0 \dots \ell]$ has two partial k_E-occurrences at $\tilde{T}[i]$ with distances, respectively, d and $d' > d$. If $P[\ell + 1 \dots m - 1]$ matches a prefix of some string in $\tilde{T}[i + 1]$ with e errors and $e + d' \leq k$, then also $e + d \leq k$. Therefore, it suffices to store distance d associated to $P[0 \dots \ell]$ to output an occurrence of P (or an extended partial one) at $\tilde{T}[i + 1]$.

The following lemma summarizes the time complexity of k_E-EDSM.

Lemma 18. *Given P of length m, $\tilde{T}$ of length n and total size N, and an integer $0 < k < m$, algorithm k_E-EDSM solves the k_E-EDSM problem on-line in time $\mathcal{O}(k^2mG+kN)$, G being the total number of strings in $\tilde{T}$.*

Proof. At the i-th iteration, algorithm k_E-EDSM tries to extend each $p \in L_p$ with each string $S \in \tilde{T}[i]$. By Lemma 13, to build $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n - 1]$, requires time $\mathcal{O}(N)$. By Lemma 17, to extend a single prefix with a string S can be done in time $\mathcal{O}(k^2)$. Since in L_p there are at most $|P| = m$ prefixes, to extend them all with a single

string S requires time $\mathcal{O}(mk^2)$. In $\tilde{T}[i]$ there are $|\tilde{T}[i]|$ strings, so the time cost rises to $\mathcal{O}(|\tilde{T}[i]|mk^2)$ for each $\tilde{T}[i]$, leading to an overall time cost of $\mathcal{O}(k^2mG)$ to perform extensions. By Lemma 15, the prefixes of P that are at distance at most k from any suffix of S as well as the k_E-occurrences of P that start and end at position i can be found in time $\mathcal{O}(k^2G + kN)$; the overall time complexity for the whole k_E-EDSM algorithm is then $\mathcal{O}(N + k^2mG + k^2G + kN) = \mathcal{O}(k^2mG + kN)$. The algorithm is on-line in the sense that any occurrence of the pattern ending at position i is reported before reading $\tilde{T}[i+1]$. □

We thus have the following result.

Theorem 12. *The k_E-EDSM problem can be solved on-line in time $\mathcal{O}(k^2mG + kN)$ and space $\mathcal{O}(m)$ for constant-sized alphabets.*

Proof. To obtain the space bound $\mathcal{O}(m)$, we need to slightly modify Algorithm k_E-EDSM in the following way: each string $S \in \tilde{T}[i]$ is (conceptually) divided into windows of size $2m$ (except for the last one, whose length is $\leq m$) overlapping by m. Let W_j be the j-th window in S, $1 \leq j \leq \lceil \frac{|S|}{m} \rceil$. Instead of building $ST^*_{P,\tilde{T}[i]}$ for each degenerate letter $\tilde{T}[i]$, the algorithm now builds ST^*_{P,W_j} for each $1 \leq j \leq \lceil \frac{|S|}{m} \rceil$ and for each $S \in \tilde{T}[i]$: since the windows are of size $2m$, this can be done in both time and space $\mathcal{O}(m)$. Both algorithms k_E-BORD and k_E-EXT require space linear in the size of the string that spell the columns of the dynamic programming table, that is either P (in extensions) or a window of size $2m$ (in borders). Each list (L_c, L_p, L') and each vector (V_c, V_p) requires space $\mathcal{O}(m)$, so the overall required space is actually $\mathcal{O}(m)$.

The time bound is not affected by these modifications of the algorithm: the maximum number of windows in $\tilde{T}[i]$, in fact, is $\max\{|\tilde{T}[i]|, \lceil \frac{N_i}{m} \rceil\}$, where $N_i = \sum_{j=0}^{|\tilde{T}[i]|-1} |\tilde{T}[i][j]|$. This means that it takes time $\mathcal{O}(m|\tilde{T}[i]|)$ or $\mathcal{O}(m\frac{N_i}{m}) = O(N_i)$ to build and pre-process every suffix tree for $\tilde{T}[i]$. Algorithm k_E-BORD requires time $\mathcal{O}(k^2 + km) = \mathcal{O}(km)$ (because $k < m$) for each window: again, this must be multiplied by the number of windows in $\tilde{T}[i]$, so the time is $\max\{\mathcal{O}(km|\tilde{T}[i]|), \mathcal{O}(kN_i))\}$ for $\tilde{T}[i]$. Coming to algorithm k_E-EXT, nothing changes, as prefixes of P can only be extended by prefixes of S, so it suffices to consider one window for each S: it still requires time $\mathcal{O}(k^2mG)$ over the whole ED text. Summing up all these considerations, the overall time is

$$\mathcal{O}(\sum_{i=0}^{n-1}[\max\{m|\tilde{T}[i]|, N_i\} + \max\{km|\tilde{T}[i]|, kN_i\}] + k^2mG) =$$
$$= \mathcal{O}(\sum_{i=0}^{n-1}[\max\{km|\tilde{T}[i]|, kN_i\}] + k^2mG)$$

which is clearly bounded by $\mathcal{O}(k^2mG + kN)$. □

To sum up, the following example shows a full iteration of k_E-EDSM.

Example 9 (Running example)**.** *Consider the usual pattern P=`GAACAA` and text $\tilde{T}$ of Example 5, $k = 1$, $i = 1$. Examples 7 and 8 considered string S = `AA` $\in \tilde{T}[1]$ to compute borders and extensions respectively, so that $L_c = \{0, 1, 2, 3\}$ and $V_c = [1, 1, 0, 1, \infty, \infty]$*

so far: consider now $S = \texttt{AG} \in \tilde{T}[1]$.
k_E-BORD$(P, S, ST'_{P,\tilde{T}[1]}1)$ *returns pair* $(0,0)$*: since 0 already belongs to* L_c *and* $V_c[0] = 1 > 0$*, we set* $V_c[0] = 0$*, so that* $L_c = \{0,1,2,3\}$ *and* $V_c = [0,1,0,1,\infty,\infty]$*. Now* k_E-EXT$(1, P, S, ST'_{P,\tilde{T}[1]}1)$ *returns* $(2,1)$*: 2 is already in the list, but since* $V_c[2] = 0 < 1$ *we leave it as it is.*
k_E-EXT$(2, P, S, ST'_{P,\tilde{T}[1]}0)$ *does not provide any additional extensions (as it is not possible to extend* $P[0,1] = \texttt{GA}$ *with* $S = \texttt{AG}$ *and no additional edit operations), so we move to* $S = \varepsilon \in \tilde{T}[1]$*. Of course the empty string can only be used to extend the prefixes already matched at* $\tilde{T}[0]$*: in this case,* k_E-EXT$(1, P, S, ST'_{P,\tilde{T}[1]}1)$ *outputs pairs* $(1,0)$ *and* $(1,1)$*,* k_E-EXT$(2, P, S, ST'_{P,\tilde{T}[1]}0)$ *reports pair* $(2,1)$*, which are all stored in* L_c *and* V_c *already. The whole iteration thus ends with* $L_c = \{0,1,2,3\}$ *and* $V_c = [0,1,0,1,\infty,\infty]$.

3.4 An Algorithm for k_H-EDSM

The overall structure of algorithm k_H-EDSM (pseudocode not shown) is the same as k_E-EDSM. We assume a constant-sized alphabet. The two algorithms differ in the functions used to perform Step (1) (k_H-BORD rather than k_E-BORD) and Step (2) (k_H-EXT rather than k_E-EXT). The new functions take as input the same parameters as the old ones and, like them, they both return lists of pairs (j, d) (pseudocode shown below). Unlike k_E-BORD and k_E-EXT, with k_H-BORD and k_H-EXT such pairs now represent partial k_H-occurrences of P in $\tilde{T}$.

k_H-BORD$(P, S, ST^*_{P,\tilde{T}[i]}, k)$

```
1  for h = 0 to |S| − 1 do
2    d ← 0; j ← 0; h' ← h;
3    while d ≤ k do
4      ℓ ← lce_{P,S}(j, h');
5      if h' + ℓ = |S| then report (|S| − h − 1, d) ;
6      else
7        if h' + ℓ + 1 = |S| ∧ d + 1 ≤ k then report (|S| − h, d + 1) ;
8        else
9          if j + ℓ = |P| then report (|P| − 1, d) ;
10         else
11           if j + ℓ + 1 = |P| ∧ d + 1 ≤ k then report (|P| − 1, d + 1) ;
12           else d ← d + 1; j ← j + ℓ + 1; h' ← h' + ℓ + 1 ;
```

At the i-th iteration, for each $S \in \tilde{T}[i]$ and any position h in S, k_H-BORD determines whether a prefix of P is at distance at most k from the suffix of S starting at position h via executing up to $k+1$ *lce* queries in the following manner: computing $\ell = lce_{P,S}(0, h)$, it finds out that $P[0 \ldots \ell - 1]$ and $S[h \ldots h + \ell - 1]$ match exactly and $P[\ell] \neq S[h + \ell]$. It can then skip one position in both strings (the mismatch $P[\ell] \neq S[h + \ell]$), increasing the error-counter d by 1, and compute the $lce_{P,S}(\ell + 1, h + \ell + 1)$. This process is performed up to $k+1$ times, until either (i) the end of S is reached, and then a prefix of

k_H-EXT(j,P,S,$ST^*_{P,\tilde{T}[i]}$,k)

```
1  if S = ε then report (j, 0);
2  else
3    d ← 0; h ← 0; j' ← j;
4    while d ≤ k do
5      ℓ ← lce_{S,P}(h, j');
6      if h + ℓ = |S| then report (j' + ℓ − 1, d) ;
7      else
8        if h + ℓ + 1 = |S| ∧ d + 1 ≤ k then report (j' + ℓ, d + 1) ;
9        else
10         if j' + ℓ = |P| then report (|P| − 1, d) ;
11         else
12           if j' + ℓ + 1 = |P| ∧ d + 1 ≤ k then report (|P| − 1, d + 1) ;
13           else d ← d + 1; h ← h + ℓ + 1; j' ← j' + ℓ + 1 ;
```

P is at distance at most k from the suffix of S starting at h (lines 7-12 in pseudocode); or (ii) the end of P is reached, then a k_H-occurrence of P has been found (lines 13-17 in pseudocode). If the end of S nor the end of P are reached, then more than k substitutions are required, and the algorithm continues with the next position (that is, $h+1$) in S.

The following lemma gives the total cost of all the calls of algorithm k_H-BORD in k_H-EDSM.

Lemma 19. *Given P of length m, $\tilde{T}$ of length n and size N, the $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, and an integer $0 < k < m$, k_H-BORD finds the minimal Hamming distance $\leq k$ between the prefixes of P and any suffix of $S \in \tilde{T}[i]$, as well as the k_H-occurrences of P that start and end at position i, in time $\mathcal{O}(kN)$.*

Proof. For any position h in S, the k_H-BORD algorithm finds the prefix of P that is at distance at most k from the suffix of S starting at position h in time $\mathcal{O}(k)$ by performing up to $k+1$ *lce* queries (Fact 10). Over all positions of S, the method therefore requires time $\mathcal{O}(k|S|)$. Doing this for all $S \in \tilde{T}[i]$ and for all $i \in [0, n-1]$ leads to the result. □

At the i-th iteration, for each partial k_H-occurrence of P started earlier (represented by $p \in L_p$ similar to algorithm k_E-EDSM) k_H-EXT tries to extend it with a string from the current text position. To this end, for each string $S \in \tilde{T}[i]$, it checks whether some partial k_H-occurrence can be extended with the whole S starting from position $j = p+1$ of P, or whether a full k_H-occurrence can be obtained by considering only a prefix of S for the extension. The algorithm therefore executes up to $k+1$ *lce* queries with the same possible outcomes and consequences mentioned for k_H-BORD.

Lemma 20. *Given P of length m, $\tilde{T}$ of length n and size N, the $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, and an integer $0 < k < m$, k_H-EXT finds all the extensions of prefixes of P required by k_H-EDSM in time $\mathcal{O}(kmG)$, G being the total number of strings in $\tilde{T}$.*

Proof. Algorithm k_H-EXT determines in time $\mathcal{O}(k)$ whether a partial k_H-occurrence of P can be extended by S by performing up to $k+1$ constant-time *lce* queries (Fact 10); checking whether a full k_H-occurrence is obtained by considering only a prefix of S for the extension can be performed within the same complexity. Since P has m different prefixes, extending all of them costs $\mathcal{O}(km)$ per each string S. Since there are G such strings, the overall time is $\mathcal{O}(kmG)$. □

Lemma 21. *Given P of length m, $\tilde{T}$ of length n and total size N, and an integer $0 < k < m$, algorithm k_H-EDSM solves the k_H-EDSM problem on-line in time $\mathcal{O}(kmG + kN)$, G being the total number of strings in $\tilde{T}$.*

Proof. At the i-th iteration, algorithm k_H-EDSM tries to extend each $p \in L_p$ with each string $S \in \tilde{T}[i]$. By Lemma 13, building $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, requires time $\mathcal{O}(N)$. By Lemma 20, extending prefixes of P stored in L_p with each string $S \in \tilde{T}[i]$ has an overall time cost of $\mathcal{O}(kmG)$. By Lemma 19, the prefixes of P that are at distance at most k from any suffix of S as well as the k_H-occurrences of P that start and end at position i can be found in time $\mathcal{O}(kN)$ in total. Summing up, the overall time complexity for the whole k_H-EDSM algorithm is then $\mathcal{O}(N + kmG + kN) = \mathcal{O}(kmG + kN)$, as $G \geq n$. The algorithm is on-line in the sense that any occurrence of the pattern ending at position i is reported before reading $\tilde{T}[i+1]$. □

The proof of Theorem 12 suggests a way in which algorithm k_E-EDSM can be run on-line in space $\mathcal{O}(m)$; it should be straightforward to see that a similar modification of algorithm k_H-EDSM leads to the following result.

Theorem 13. *The k_H-EDSM problem can be solved on-line in time $\mathcal{O}(kmG + kN)$ and space $\mathcal{O}(m)$ for constant-sized alphabets.*

3.5 Extension to General Integer Alphabets

The algorithms presented in the previous sections are designed for constant-sized alphabets only: a straightforward switch to the general integer alphabets case would entail an increase in the time required to build the suffix trees, and hence in the complexity of the algorithm. In this section we show how to extend our results to the case of general integer alphabets, while maintaining the same time and space complexity. We obtain this by using perfect hashing [154] to build the suffix tree of a window of length (at most) $2m$ in $\mathcal{O}(m)$ time for general integer alphabets. The procedure consists of a preprocessing phase followed by the proper construction of the suffix tree.

Preprocessing: We hash the letters of pattern P using perfect hashing. For each key, we assign a rank value from $\{1, \ldots, m\}$. This takes $\mathcal{O}(m)$ (expected) time and space [154].

Construction: When reading a window W of length (at most) $2m$ of the text we look up its letters using the hash table constructed during the preprocessing phase. If a letter is in the hash table we replace it in W by its rank value; otherwise we replace it by rank $m+1$. This operation takes $\mathcal{O}(1)$ time [154]. We can now

construct the suffix tree of W in $\mathcal{O}(m)$ time and $\mathcal{O}(m)$ space using Farach's suffix tree construction algorithm [139]. This is because string W is over $\{1, \ldots, m+1\}$.

We thus have the following lemma.

Lemma 22. *Given P of length m and $\tilde{T}$ of length n and size N, to build ST^*_{P,W_j} for each window W_j of length $2m$, $1 \leq j \leq \lceil \frac{|S|}{m} \rceil$, and for each $S \in \tilde{T}[i]$, for all $i \in [0, n-1]$, requires total time $\mathcal{O}(N)$ for general integer alphabets.*

By plugging this lemma into the algorithms of, respectively, Section 3.3 and Section 3.4, we obtain the following results.

Theorem 14. *The k_E-EDSM problem can be solved on-line in time $\mathcal{O}(k^2mG + kN)$ and space $\mathcal{O}(m)$ for general integer alphabets.*

Theorem 15. *The k_H-EDSM problem can be solved on-line in time $\mathcal{O}(kmG + kN)$ and space $\mathcal{O}(m)$ for general integer alphabets.*

3.6 Final Remarks

In this chapter we introduced two algorithms for finding all approximate matches of a pattern P of length m in an ED text $\tilde{T}$ of length n and size N: an $\mathcal{O}(kmG + kN)$-time algorithm for Hamming distance; and an $\mathcal{O}(k^2mG + kN)$-time algorithm for edit distance, where G is the total number of strings in $\tilde{T}$ and k is the maximum distance allowed. Both algorithms are on-line, their working space is $\mathcal{O}(m)$, and they work for general integer alphabets.

There are at least two directions for future work. The first one is to improve the time complexity for these problems by perhaps removing the dependency on parameter G. The second direction is to develop algorithms for searching multiple patterns simultaneously under the approximate setting.

Chapter 4

Comparing Degenerate Strings

Key Points

Problem. String comparison is the core computational task in several string-processing applications, whether they process standard or uncertain strings. For example, to extract frequent patterns from a single string, to find common substrings among several strings, or to check whether a string is palindromic, one must have a tool for comparing two strings. Our goal here is to find an efficient method to compare two sets of similar strings, represented in a string-like, compact form. More precisely, we consider *gapless* multiple sequence alignments (MSA) of fixed width, that is, for example, high-scoring local alignments of multiple sequences.

Model. We encode the gapless multiple sequence alignments as *generalized degenerate strings* (GD strings), a restricted variant of ED strings where the ith set contains strings of the same length k_i but this length can vary between different sets. Our solution to the problem of comparing two GD strings is based on a combinatorial result of independent interest: although the intersection of two GD strings can be exponential in the total size of the two strings, it can be represented in *linear* space. We also show that a result, which is essentially the same as our string comparison algorithm, can be obtained by employing an automata-based approach when the alphabet has constant size.

Included Works

This chapter presents the results of the paper **Comparing Degenerate Strings** [28], published in *Fundamenta Informaticae*, which is the journal extension of **Degenerate String Comparison and Applications** [27], presented at the *18th Workshop on Algorithms in Bioinformatics (WABI 2018)*.

4.1 Introduction

Uncertain sequences are useful for representing sets of similar strings in a compact form. They highlight common segments by collapsing them, and explicitly represent varying segments by listing all possible options. In Chapters 2 and 3 we considered elastic-degenerate (ED) strings, that are sequence of subsets of Σ^* (see also network expressions [278]), and solve both the exact and the approximate version of pattern matching on ED strings.

In this chapter we introduce another special type of uncertain sequence called a generalized degenerate string; this can be viewed as a restricted variant of ED strings. Formally, a *generalized degenerate string* (GD string) $\hat{S}$ over Σ is a sequence of n sets of strings of total size N over Σ, where the ith set contains strings of the same length $k_i > 0$ but this length can vary between different sets. We denote the sum of these lengths $k_0, k_1, \ldots, k_{n-1}$ by W. A GD string can be used to represent in a compact form a *gapless* multiple sequence alignment (MSA) of fixed width, that is, for example, a high-scoring local alignment of multiple sequences (see Figure 4.1 and Figure 4.2).

```
CA--AGCTCTATCTCGTA--TT        AGCTCTATCTCG
C---AGCCGAAGCTCGTATATT        AGCCGAAGCTCG
CATCAAGTCAACGCAG----TT        AAGTCAACGCAG
```

Figure 4.1: Multiple sequence alignment (left) and Local Gapless Alignment (right).

In this chapter we solve the problem of comparing two GD strings, that is, deciding whether two GD strings have a non-empty intersection. String comparison is the core computational task in several string-processing applications, whether they process standard or uncertain strings. For example, to extract frequent patterns from a single string, to find common substrings among several strings, or to check whether a string is palindromic, one must have a tool for comparing two strings. Therefore, we first develop an efficient algorithm to compare two GD strings, and then, as proof of concept, we apply this algorithm to compute all palindromes in a GD string.

In a standard string, a palindrome is a sequence that reads the same from left to right and from right to left. Detection of palindromic factors in texts is a classical and well-studied problem in algorithms on strings and combinatorics on words with a lot of variants arising out of different practical scenarios. A string $X = X[0]X[1]\ldots X[n-1]$ is said to have an initial palindrome of length k if its prefix of length k is a palindrome. Manacher first discovered an on-line algorithm that finds all initial palindromes in a

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

Figure 4.2: GD string obtained from the local gapless alignment of Figure 4.1.

string [264]. Later Apostolico et al. observed that the algorithm given by Manacher is actually able to find all maximal palindromic factors in the string in $\mathcal{O}(n)$ time [32]. Gusfield gave an off-line linear-time algorithm to find all maximal palindromes in a string and also discussed the relation between biological sequences and gapped palindromes [184].

With uncertain sequences, we first need to have an algorithm for efficient *string comparison*, where automata provide the following baseline. Let $\hat{X}$ and $\hat{Y}$ be two GD (or two ED) strings of total sizes N and M, respectively. We first construct the non-deterministic finite automaton (NFA) A of $\hat{X}$ and the NFA B of $\hat{Y}$ in time $\mathcal{O}(N+M)$. We then construct the product NFA C such that $L(C) = L(A) \cap L(B)$ in time $\mathcal{O}(NM)$. The non-emptiness decision problem, namely, checking if $L(C) \neq \emptyset$, is decidable in time linear in the size of C, using breadth-first search (BFS). Hence the comparison of $\hat{X}$ and $\hat{Y}$ can be done in time $\mathcal{O}(NM)$. It is known that if there existed faster methods for obtaining the automata intersection, then significant improvements would be implied to many long standing open problems [251]. Hence an immediate reduction to the problem of NFA intersection does not particularly help. For GD strings, specifically, we show that we can construct an ad-hoc deterministic finite automaton (DFA) for $\hat{X}$ and $\hat{Y}$, so that the intersection can be performed efficiently, but this simple solution cannot achieve $\mathcal{O}(N+M)$ time for integer alphabets as its cost is alphabet-dependent.

Our Contribution. Our first result is an $\mathcal{O}(N+M)$-time algorithm for deciding whether the intersection of two GD strings of sizes N and M, respectively, over a constant-sized alphabet is non-empty. We show this by means of DFAs (see Section 4.3). This result is based on a combinatorial result of independent interest: although the intersection of two GD strings can be exponential in the total size of the two strings, it can be represented in linear space. We next present an efficient implementation of this result in the standard word RAM model with word size $w = \Omega(\log(N+M))$ that works also for integer alphabets. Specifically, we show an $\mathcal{O}(N+M)$-time algorithm for deciding whether the intersection of two GD strings of sizes N and M, respectively, over an integer alphabet is non-empty (see Section 4.4). We then apply our string comparison tool to compute palindromes in GD strings. We present a simple $\mathcal{O}(\min\{W, n^2\}N)$-time algorithm for computing all palindromes in $\hat{S}$; a proof-of-concept experiment is also presented (see Section 4.5). Notably, we complement this upper bound with a non-trivial $\Omega(n^2|\Sigma|)$ lower bound under the Strong Exponential Time Hypothesis [207, 208] for computing all maximal palindromes in $\hat{S}$ (see Section 4.6). Let us remark that there exists an infinite family of GD strings over an integer alphabet of size $|\Sigma| = \Theta(N)$ on which our algorithm requires time $\mathcal{O}(n^2 N)$, thus matching the conditional lower bound. We conclude this chapter with some open problems (see Section 4.7).

4.2 Preliminaries

A string P is a *palindrome* if and only if $P = P^R$. If factor $X[i\,..\,j]$, $0 \leq i \leq j \leq n-1$, of string X of length n is a palindrome, then $\frac{i+j}{2}$ is the *center* of $X[i\,..\,j]$ in X and $\frac{j-i+1}{2}$ is the *radius* of $X[i\,..\,j]$. In other words, a palindrome is a string that reads the same forward and backward, i.e., a string P is a palindrome if $P = YaY^R$ where Y is a string, Y^R is the reversal of Y and a is either a single letter (when the center is an integer) or the empty string (when it is not). Moreover, $X[i\,..\,j]$ is called a *palindromic factor* of

X. It is said to be a *maximal palindrome* if there is no other palindrome in X with center $\frac{i+j}{2}$ and larger radius. Hence X has exactly $2n-1$ maximal palindromes. A maximal palindrome P of X can be encoded as a pair (c, r), where c is the center of P in X and r is the radius of P.

Definition 2. *A* generalized degenerate string (GD string) $\hat{S} = \hat{S}[0]\hat{S}[1]\dots\hat{S}[n-1]$ *of* length n *over an alphabet* Σ *is a finite sequence of* n *degenerate letters. Every degenerate letter* $\hat{S}[i]$ *of* width $k_i > 0$*, denoted also by* $w(\hat{S}[i])$*, is a finite non-empty set of strings such that* $\hat{S}[i][0], \dots, \hat{S}[i][|\hat{S}[i]|-1] \in \Sigma^{k_i}$*. For any GD string* $\hat{S}$*, we denote by* $\hat{S}[i]\dots\hat{S}[j]$ *the* GD substring *of* $\hat{S}$ *that starts at position* i *and ends at position* j*.*

In this work, we generally consider GD strings over an *integer alphabet* of size $\sigma = N^{\mathcal{O}(1)}$. We now define some parameters that describe the structure of a GD string.

Definition 3. *The* total size N *and* total width W*, denoted also by* $w(\hat{S})$*, of a GD string* $\hat{S}$ *are respectively defined as* $N = \sum_{i=0}^{n-1} |\hat{S}[i]| \cdot k_i$ *and* $W = \sum_{i=0}^{n-1} k_i$*.*

Example 10. *The GD string* $\hat{S}$ *of Figure 4.2 has length* $n=6$*, size* $N=28$*, and* $W=12$*.*

Definition 4. *Given two degenerate letters* $\hat{X}$ *and* $\hat{Y}$*, their* Cartesian concatenation *is*

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

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

Definition 5. *Consider a GD string* $\hat{S}$ *of length* n*. The* language of $\hat{S}$ *is*

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

Our goal for GD string comparison is to establish, given two GD strings $\hat{R}$ and $\hat{S}$, whether the intersection of their language is non-empty. To this purpose, we define the notions of *chop* and *active suffixes*.

Definition 6. *Let* $\hat{X} = \{x_i \in \Sigma^k\}$ *and* $\hat{Y} = \{y_j \in \Sigma^h\}$ *be two degenerate letters on alphabet* Σ*. Further let us assume without loss of generality that* $w(\hat{Y}) < w(\hat{X})$ *(i.e.,* $h < k$*). We define the set* chop of $\hat{X}$ and $\hat{Y}$ *and the set* active suffixes of $\hat{X}$ and $\hat{Y}$ *as follows:*

- $chop_{\hat{X},\hat{Y}} = \{y_j \in \hat{Y} \mid y_j$ *matches a prefix of some* $x_i \in \hat{X}\}$
- $active_{\hat{X},\hat{Y}} = \{x_i[h\,..\,k-1] \mid x_i[0\,..\,h-1] \in chop_{\hat{X},\hat{Y}}\}$

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

Example 11 (for Definition 6)**.** *Consider the following degenerate letters* $\hat{X}$ *and* $\hat{Y}$ *where* $w(\hat{Y}) < w(\hat{X})$*. The underlined strings in letter* $\hat{Y}$ *are prefixes of strings in letter* $\hat{X}$*, hence they are in* $chop_{\hat{X},\hat{Y}}$*. The suffixes of such strings in* $\hat{X}$ *are the active suffixes in* $active_{\hat{X},\hat{Y}}$*.*

$$\hat{X} = \begin{Bmatrix} \underline{\texttt{TCC}}\texttt{TA} \\ \texttt{ATCGA} \\ \underline{\texttt{TCC}}\texttt{AC} \\ \underline{\texttt{CAT}}\texttt{TA} \end{Bmatrix} \quad \hat{Y} = \begin{Bmatrix} \texttt{GCA} \\ \underline{\texttt{CAT}} \\ \underline{\texttt{TCC}} \end{Bmatrix} \quad chop_{\hat{X},\hat{Y}} = \{\texttt{CAT}, \texttt{TCC}\} \quad active_{\hat{X},\hat{Y}} = \{\texttt{TA}, \texttt{AC}\}$$

Definition 7. *Let $\hat{R}$ and $\hat{S}$ be two GD strings of length r and s, respectively. $\hat{R}[0]\dots\hat{R}[i]$ is the* prefix *of $\hat{R}$ that ends at position i. It is called* proper *if $i \neq r-1$. We say that $\hat{R}[0]\dots\hat{R}[i]$ is* synchronized *with $\hat{S}[0]\dots\hat{S}[j]$ if $w(\hat{R}[0]\dots\hat{R}[i]) = w(\hat{S}[0]\dots\hat{S}[j])$. We call these the* shortest synchronized prefixes *of $\hat{R}$ and $\hat{S}$, respectively, when for all i', j' such that $i' < i$ and $j' < j$, we have that $w(\hat{R}[0]\dots\hat{R}[i']) \neq w(\hat{S}[0]\dots\hat{S}[j'])$.*

Example 12 (for Definition 7)**.** *The GD string $\hat{S}$ of Figure 4.2 and the GD string $\hat{R}$ below have no synchronized proper prefixes, while $\hat{S}$ and the GD string $\hat{T}$ below have synchronized prefix of length 1 and 6 (because $w(\hat{S}[0]) = w(\hat{T}[0]) = 1$ and $w(\hat{S}[0]\hat{S}[1]\hat{S}[2]) = w(\hat{T}[0]\hat{T}[1]) = 6$) , and thus their shortest synchronized prefix has length 1.*

$$\hat{R} = \begin{Bmatrix} \texttt{CGCAC} \\ \texttt{AGCCG} \\ \texttt{AGCCG} \end{Bmatrix} \cdot \begin{Bmatrix} \texttt{AAT} \\ \texttt{TAG} \end{Bmatrix} \cdot \begin{Bmatrix} \texttt{CTCG} \\ \texttt{GCAG} \\ \texttt{CTCA} \end{Bmatrix} \qquad \hat{T} = \begin{Bmatrix} \texttt{C} \\ \texttt{A} \end{Bmatrix} \cdot \begin{Bmatrix} \texttt{CCACT} \\ \texttt{GCCCA} \\ \texttt{TCCTT} \end{Bmatrix} \cdot \begin{Bmatrix} \texttt{AT} \\ \texttt{AG} \end{Bmatrix} \cdot \begin{Bmatrix} \texttt{ATCG} \\ \texttt{AGCT} \\ \texttt{GGCA} \end{Bmatrix}$$

4.3 GD String Comparison for Small Alphabets Using Automata

In this section we describe a simple algorithm for GD string comparison that is based on DFAs. It works in linear time for constant-sized alphabets, and it is generalized in the subsequent section to work for integer alphabets.

Given $\hat{R}$ and $\hat{S}$ of total size N and M, respectively, each degenerate letter of $\hat{R}$ and $\hat{S}$ can be represented by a trie, where its leaves are collapsed to a single one. For every two consecutive degenerate letters, the collapsed leaves of the former trie coincide with the root of the latter trie. An acyclic DFA is obtained in this way, as illustrated in Example 13 below. We can perform the comparison of $\hat{R}$ and $\hat{S}$ by intersecting their corresponding DFAs using BFS on their product DFA.

The trivial upper bound on the number of reachable states is $\mathcal{O}(NM)$, and the traditional method for constructing the DFA for $L(\hat{R}) \cap L(\hat{S})$ is non-linear, but this can be improved to $\mathcal{O}(N+M)$ by exploiting the structure of the two input DFAs. Each state in such a DFA has a unique level: the common length of paths from the initial state. This structure is *inherited* by the product DFA. In other words, a level-i state in the product DFA corresponds to a pair of level-i states in the input DFAs. Observe that a level-i state in one DFA is uniquely represented by the label of the path from the root of its trie, and for a fixed DFA and level, these labels have uniform lengths. Considering the two states composing a reachable state in the product DFA, it is easy to see that the shorter label must be a suffix of the longer label. Hence, the state in the DFA with longer labels at level i uniquely determines the state in the DFA with

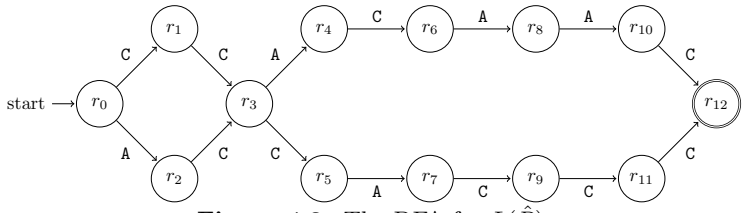


Figure 4.3: The DFA for $L(\hat{R})$

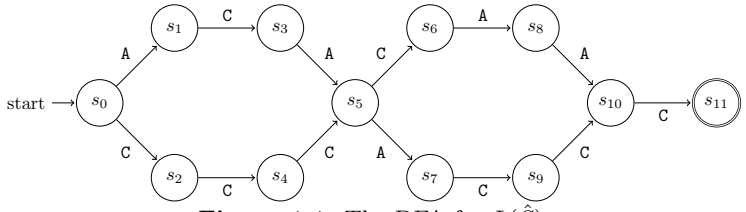


Figure 4.4: The DFA for $L(\hat{S})$

shorter labels at level i. Consequently, the number of reachable level-i states in the product DFA is bounded by the number of level-i states in the input DFAs, and the size is $\mathcal{O}(N + M)$.

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

Example 13. *Say we want to compare the following two GD strings:*

$$\hat{R} = \begin{Bmatrix} \texttt{A C} \\ \texttt{C C} \end{Bmatrix} \cdot \begin{Bmatrix} \texttt{A C A A C} \\ \texttt{C A C C C} \end{Bmatrix} \qquad \hat{S} = \begin{Bmatrix} \texttt{A C A} \\ \texttt{C C C} \end{Bmatrix} \cdot \begin{Bmatrix} \texttt{A C C} \\ \texttt{C A A} \end{Bmatrix} \cdot \{\texttt{C}\}$$

We start by constructing the DFA for $L(\hat{R})$ and the DFA for $L(\hat{S})$, shown in Figure 4.3 and Figure 4.4, respectively. We then construct their product DFA, shown in Figure 4.5, which gives the intersection of $L(\hat{R})$ and $L(\hat{S})$: `ACACAAC` *and* `CCCACCC`*.*

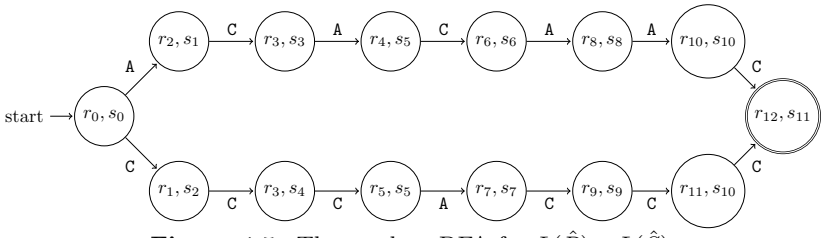


Figure 4.5: The product DFA for $L(\hat{R}) \cap L(\hat{S})$

4.4 GD String Comparison for Integer Alphabets

In this section we describe a more general algorithm for the problem of GD string comparison addressed also in Section 4.3. Let $\hat{R}$ and $\hat{S}$ be of total size N and M,

respectively. We provide an $\mathcal{O}(N+M)$-time algorithm in the standard word RAM model with word size $w = \Omega(\log(N+M))$ to decide whether the intersection of $\hat{R}$ and $\hat{S}$ is non-empty, which also works for integer alphabets.

We show that, even if the size of $L(\hat{R}) \cap L(\hat{S})$ can be exponential in the total sizes of $\hat{R}$ and $\hat{S}$ (Fact 16), the problem of GD string comparison, i.e., deciding whether $L(\hat{R}) \cap L(\hat{S})$ is non-empty, can be solved in time linear with respect to the sum of the total sizes of the two GD strings (Theorem 18) and is thus of independent interest.

Fact 16. *Given two GD strings $\hat{R}$ and $\hat{S}$, $L(\hat{S}) \cap L(\hat{R})$ can have size exponential in the total sizes of $\hat{R}$ and $\hat{S}$. A trivial example of this is given when $\hat{R} = \hat{S}$ and they are a sequence of n degenerate letters of, for example, 2 strings of 2 letters each: the total size of both $\hat{R}$ and $\hat{S}$ is $4n$, while $|L(\hat{S}) \cap L(\hat{R})| = |L(\hat{R})| = 2^n$.*

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

Lemma 23. *Consider two GD strings $\hat{S} = \hat{S}'\hat{S}''$ and $\hat{R} = \hat{R}'\hat{R}''$ such that $w(\hat{S}) = w(\hat{R})$. If $\hat{S}'$ is synchronized with $\hat{R}'$, then $L(\hat{R}) \cap L(\hat{S}) = (L(\hat{R}') \cap L(\hat{S}')) \otimes (L(\hat{R}'') \cap L(\hat{S}''))$.*

By applying Lemma 23 wherever $\hat{R}$ and $\hat{S}$ have synchronized prefixes, we are then left with the problem of intersecting GD strings with no synchronized proper prefixes. We now define an alternative decomposition within such strings (see also Example 14).

Definition 8. *Let $\hat{R}$ and $\hat{S}$ be two GD strings of length r and s, respectively, with no synchronized proper prefixes. We define*

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

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

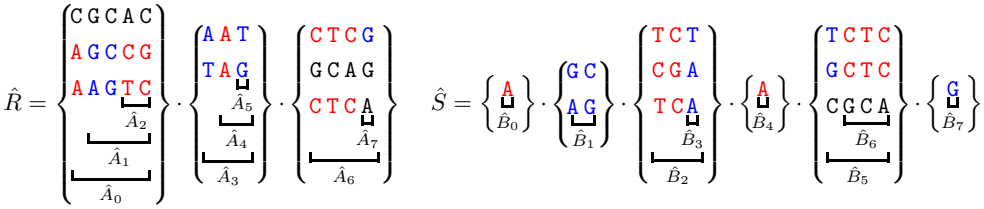
$$\hat{A}_0 = \hat{R}[0], \hat{B}_0 = \hat{S}[0], \text{ and } \textit{pos}(\hat{A}_0) = \textit{pos}(\hat{B}_0) = 0. \text{ If } \textit{chop}_{i-1} \neq \emptyset, \text{ for } 0 < i \le r+s-2,$$

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

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

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

Example 14 (for Definition 8)**.** *Consider the following GD strings $\hat{R}$ and $\hat{S}$ with no synchronized proper prefixes: $\textit{chop}_0$ is the first red set from the left, $\textit{chop}_1$ is the first blue one, $\textit{chop}_2$ is the second red one, etc. The c-chain$(\hat{R}, \hat{S})$ terminates when $q = 7$.*



Definition 9. *Let $\hat{R}$ and $\hat{S}$ be two GD strings of length r and s, respectively, with $w(\hat{R}) = w(\hat{S})$ and no synchronized proper prefixes. We define $G_{\hat{R},\hat{S}}$ as a directed acyclic graph with a structure of up to $r + s - 1$ levels, each node being a set of strings, as follows, where we assume without loss of generality that $w(\hat{R}[0]) > w(\hat{S}[0])$:*

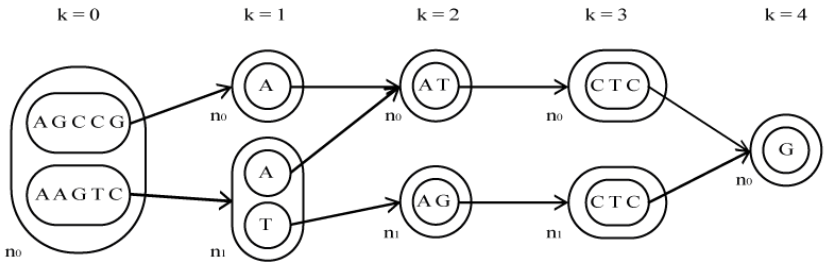
Level $k = 0$ *consists of a single node:*
$n_0 = \{x \in \hat{R}[0] \,|\, x = y_0 \dots y_{q_0}$ *with* $y_j \in chop_j \; \forall j : 0 \le j \le q_0\}$, *where q_0 is the index of the rightmost chop containing suffixes of $\hat{R}[0]$.*

Level $k > 0$ *consists of $\ell = |chop_{q_{k-1}}|$ nodes. Assuming without loss of generality that level $k-1$ has been built with suffixes of $\hat{R}[pos(\hat{A}_{q_{k-1}})]$, level k contains suffixes of a position of $\hat{S}$. Let $c_0, \dots, c_{\ell-1}$ denote the elements of $chop_{q_{k-1}}$. Then, for $0 \le i \le \ell-1$, the i-th node of level k is:*
$n_i = \{y_{q_{k-1}+1} \dots y_{q_k} \,|\, c_i y_{q_{k-1}+1} \dots y_{q_k} \in \hat{B}_{q_{k-1}}$ *with* $y_j \in chop_j \; \forall j : q_{k-1}+1 \le j \le q_k\}$, *where q_k is the index of the rightmost chop containing suffixes of $\hat{S}[pos(\hat{B}_{q_{k-1}})]$.*
Every string in level $k-1$ whose suffix is c_i is the source of an edge having the whole node n_i as a sink.

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

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

Example 15 (for Definition 9)**.** *The graph $G_{\hat{R},\hat{S}}$ for the GD strings $\hat{R}, \hat{S}$ of Example 14 is the following:*



$q_0 = 2$ and the strings in level 0 belong to $(chop_0 \otimes chop_1 \otimes chop_2) \cap \hat{R}[0]$. Level 1 contains suffixes of strings in $\hat{B}_2$ (and of strings in $\hat{B}_3$ as $chop_3 = \{\mathtt{A}, \mathtt{T}\}$ and indeed $q_1 = 3$), level 2 suffixes of strings in $\hat{A}_3$ (as $q_2 = 5$), level 3 suffixes of strings in $\hat{B}_5$

($q_3 = 6$), level 4 suffixes of strings in $\hat{A}_6$ ($q_4 = 7$). The three paths from level 0 to level 4 correspond to the three strings in $L(\hat{R}) \cap L(\hat{S})$: `AGCCGAATCTCG`, `AAGTCAATCTCG`, `AAGTCTAGCTCG`.

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

Lemma 24. *Let $\hat{R}, \hat{S}$ be two GD strings with $w(\hat{R}) = w(\hat{S}) = W$ and no synchronized proper prefixes. Then $L_k(\hat{S}) \cap L_k(\hat{R}) = paths(G^k_{\hat{R},\hat{S}})$ for all levels k of $G_{\hat{R},\hat{S}}$ such that $L_k(\hat{S}) \cap L_k(\hat{R}) \neq \emptyset$.*

Proof. Again, let us assume without loss of generality that $w(\hat{R}[0]) > w(\hat{S}[0])$. We prove the result by induction on k.
Level $k = 0$. By construction, n_0 contains strings in $\hat{R}[0] \cap (\text{chop}_0 \otimes \cdots \otimes \text{chop}_{q_0})$, which have length $|G^0_{\hat{R},\hat{S}}|$, and are also in $\hat{S}[0]$, and hence belong to both $L_0(\hat{S})$ and $L_0(\hat{R})$.
Level $k > 0$. By inductive hypothesis, we have that $L_{k-1}(\hat{S}) \cap L_{k-1}(\hat{R}) = \text{paths}(G^{k-1}_{\hat{R},\hat{S}})$; suppose that $L_k(\hat{S}) \cap L_k(\hat{R}) \neq \emptyset$, otherwise the graph ends at level $k-1$. We first show that $\text{paths}(G^k_{\hat{R},\hat{S}}) \subseteq L_k(\hat{S}) \cap L_k(\hat{R})$: by Definition 9, any $z \in \text{paths}(G^k_{\hat{R},\hat{S}})$ can be written as $z = z'z''$ with z' in $\text{paths}(G^{k-1}_{\hat{R},\hat{S}})$ and with z'' that belongs to some node at level k of $G^k_{\hat{R},\hat{S}}$ reached by an edge leaving a suffix of z'. By inductive hypothesis $z' \in L_{k-1}(\hat{S}) \cap L_{k-1}(\hat{R})$ and, again by Definition 9, $z'' \in \text{chop}_{q_{k-1}+1} \otimes \cdots \otimes \text{chop}_{q_k}$; since $L_k(\hat{S}) \cap L_k(\hat{R}) \neq \emptyset$ these chops are not empty, their concatenation contains the suffix of length $|G^k_{\hat{R},\hat{S}}| - |G^{k-1}_{\hat{R},\hat{S}}|$ of strings in both $L_k(\hat{R})$ and $L_k(\hat{S})$, and hence $z \in L_k(\hat{S}) \cap L_k(\hat{R})$.
We now show that $L_k(\hat{S}) \cap L_k(\hat{R}) \subseteq \text{paths}(G^k_{\hat{R},\hat{S}})$. Consider string $u \in L_k(\hat{S}) \cap L_k(\hat{R})$ that can be written as $u = u'u''$ with u' the prefix of u having length $|G^{k-1}_{\hat{R},\hat{S}}|$ which then belongs to $L_{k-1}(\hat{S}) \cap L_{k-1}(\hat{R})$; then, by inductive hypothesis, $u' \in \text{paths}(G^{k-1}_{\hat{R},\hat{S}})$ and, since $u \in L_k(\hat{S}) \cap L_k(\hat{R})$, then there is an edge linking a suffix of u' at level $k-1$ with a node at level k of $G^k_{\hat{R},\hat{S}}$ containing a $|G^k_{\hat{R},\hat{S}}| - |G^{k-1}_{\hat{R},\hat{S}}|$ long suffix u'' of u, and hence $u \in \text{paths}(G^k_{\hat{R},\hat{S}})$. □

As a special case of Lemma 24, if $L(\hat{S}) \cap L(\hat{R}) \neq \emptyset$, then $G_{\hat{R},\hat{S}}$ is built up to the last level and the following holds.

Theorem 17. *Let $\hat{R}, \hat{S}$ be two GD strings having lengths, respectively, r and s, with $w(\hat{R}) = w(\hat{S})$ and no synchronized proper prefixes. Then $G_{\hat{R},\hat{S}}$ has exactly $r + s - 1$ levels, and we have that $L(\hat{S}) \cap L(\hat{R}) = paths(G_{\hat{R},\hat{S}})$.*

$G_{\hat{R},\hat{S}}$ is thus a linear-sized representation of the possibly exponential-sized (see Fact 16) set $L(\hat{S}) \cap L(\hat{R})$.

We now show an $\mathcal{O}(N + M)$-time algorithm for the standard word RAM model, denoted by GDSC, that decides whether $L(\hat{R})$ and $L(\hat{S})$ share at least one string

(returns 1) or not (returns 0). GDSC starts with constructing the generalized suffix tree $T_{\hat{R},\hat{S}}$ of all the strings in $\hat{R}$ and $\hat{S}$. Then it scans $\hat{R}$ and $\hat{S}$ starting with $\hat{R}[0]$ and $\hat{S}[0]$, storing in $\text{chop}_{\hat{R},\hat{S}}$ the latest chop_i and in $\text{active}_{\hat{R},\hat{S}}$ the latest $\text{active}_{\hat{A}_i,\hat{B}_i}$, using $T_{\hat{R},\hat{S}}$. To efficiently implement GDSC, rather than explicitly storing suffixes in $\text{active}_{\hat{R},\hat{S}}$, they are stored as index positions of $\hat{R}[i]$ or $\hat{S}[j]$. A variable *suff* is used to keep track of the starting position of the suffixes. Note that this starting position is the same for every index that has been stored in $\text{active}_{\hat{R},\hat{S}}$. Given that $w(\hat{S[0]}) < w(\hat{R[0]})$, the next comparison is made between the corresponding suffixes of $\hat{R}[0]$ of length $w(\hat{R[0]}) - \textit{suff}$ and $\hat{S}[1]$, proceeding with the same process. The comparison of letters can be: (i) between $\hat{R}[i]$ and $\hat{S}[j]$; or (ii) between the corresponding suffixes of $\text{active}_{\hat{R},\hat{S}}$ and $\hat{R}[i]$; or (iii) between the corresponding suffixes of $\text{active}_{\hat{R},\hat{S}}$ and $\hat{S}[j]$. If the two GD strings have a synchronized proper prefix, this will result in $\text{active}_{\hat{R},\hat{S}} = \emptyset$ at positions i in $\hat{R}$ and j in $\hat{S}$. At this point, the comparison is restarted with the immediately following pair of degenerate letters.

Example 16. *Consider the GD strings $\hat{R}$ and $\hat{S}$ below. The set $L(\hat{R}) \cap L(\hat{S})$ contains the strings* ATACGACTAACGTT, ATACGACTAGCACT, TATCCGACTACGTT, TATCCGACTGCACT.

$$\hat{R} = \left\{\begin{matrix}\underline{\texttt{ATA}}\\ \underline{\texttt{TAT}}\end{matrix}\right\} \cdot \left\{\begin{matrix}\underline{\texttt{GACC}}\\ \underline{\texttt{CCGA}}\\ \underline{\texttt{CGAC}}\end{matrix}\right\} \cdot \left\{\begin{matrix}\underline{\texttt{CG}}\\ \underline{\texttt{CT}}\\ \underline{\texttt{TA}}\end{matrix}\right\} \cdot \left\{\begin{matrix}\underline{\texttt{ACGT}}\\ \underline{\texttt{GCAC}}\end{matrix}\right\} \cdot \{\underline{\texttt{T}}\}$$

$$\hat{S} = \left\{\begin{matrix}\underline{\texttt{TATCC}}\\ \texttt{TATGA}\\ \underline{\texttt{ATACG}}\end{matrix}\right\} \cdot \left\{\begin{matrix}\underline{\texttt{ACTA}}\\ \underline{\texttt{GACT}}\end{matrix}\right\} \cdot \left\{\begin{matrix}\underline{\texttt{GCA}}\\ \texttt{AAT}\\ \underline{\texttt{ACG}}\end{matrix}\right\} \cdot \left\{\begin{matrix}\underline{\texttt{C}}\\ \texttt{A}\\ \underline{\texttt{T}}\end{matrix}\right\} \cdot \{\underline{\texttt{T}}\}$$

Theorem 18. *Algorithm* GDSC *is correct. Given two GD strings $\hat{R}$ and $\hat{S}$ of total sizes N and M, respectively, over an integer alphabet, algorithm* GDSC *requires $\mathcal{O}(N+M)$ time.*

Proof. The correctness follows directly from Lemma 23, Lemma 24, and Theorem 17.

As for the complexity, we have the following: constructing the generalized suffix tree $T_{\hat{R},\hat{S}}$ can be done in time $\mathcal{O}(N+M)$ [139]; for the sets pair $(\hat{A}_i, \hat{B}_i)$ as in Definition 8, such that $w(\hat{A}_i) = k$ and $w(\hat{A}_i) \leq w(\hat{B}_i)$, we query $T_{\hat{R},\hat{S}}$ with the k-length prefixes of strings in $\hat{B}_i$; for integer alphabets, instead of spelling the strings from the root of $T_{\hat{R},\hat{S}}$, we locate the corresponding terminal nodes for $(\hat{A}_i, \hat{B}_i)$. It then suffices to find longest common prefixes between these suffixes to simulate the querying process. Since all suffixes are lexicographically sorted during the construction of $T_{\hat{R},\hat{S}}$, we can also have the suffixes considered by pair $(\hat{A}_i, \hat{B}_i)$ lexicographically ranked with respect to $(\hat{A}_i, \hat{B}_i)$. Hence we do not perform the longest common prefix operation for all possible suffix pairs, but only for the lexicographically adjacent ones within this group. This can be done in $\mathcal{O}(1)$ time per pair after $\mathcal{O}(N+M)$-time pre-processing over $T_{\hat{R},\hat{S}}$ [47]. chop_i is thus populated with the k-length prefixes of strings in $\hat{B}_i$ found in $\hat{A}_i$. The set $\text{active}_{\hat{A}_i,\hat{B}_i}$ of active suffixes can be found by chopping the suffixes of the string in $\hat{B}_i$

from their prefixes successfully queried in $T_{\hat{R},\hat{S}}$. This requires time $\mathcal{O}(|\hat{A}_i| + |\hat{B}_i|)$ for processing $(\hat{A}_i, \hat{B}_i)$.

Let $\hat{R}$ and $\hat{S}$ be of length r and s, respectively. Assume that $\hat{R}$ and $\hat{S}$ have no synchronized proper prefixes. Then Theorem 17 ensures that the total number of comparisons cannot exceed $r + s - 2$: this results in a time complexity of $\mathcal{O}(N + M + \sum_{i=0}^{r+s-2}(|\hat{A}_i| + |\hat{B}_i|)) = \mathcal{O}(N + M)$.

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

4.5 Computing Palindromes in GD Strings

Armed with the efficient GD string comparison tool, we shift our focus on computing palindromes in GD strings.

Definition 10. *A GD string $\hat{S}$ is a* GD palindrome *if there exists a string in $L(\hat{S})$ that is a palindrome.*

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

Example 17 (for Definition 10)**.** *In the GD string*

$$\hat{S} = \{\mathtt{G}\} \cdot \left\{\begin{matrix}\underline{\mathtt{A\,C\,A}}\\ \underline{\underline{\mathtt{T\,T\,T}}}\\ \mathtt{G\,T\,C}\end{matrix}\right\} \cdot \left\{\begin{matrix}\overline{\mathtt{T\,G}}\\ \mathtt{A\,G}\\ \underline{\mathtt{T\,T}}\end{matrix}\right\} \cdot \left\{\begin{matrix}\underline{\mathtt{G\,A}}\\ \overline{\mathtt{G\,T}}\end{matrix}\right\} \cdot \left\{\begin{matrix}\mathtt{C\,A\,G\,G\,C\,T\,T\,T}\\ \underline{\mathtt{C\,C\,A\,G\,T\,T\,A\,C}}\\ \mathtt{A\,T\,T\,T\,C\,A\,G\,G}\end{matrix}\right\} \cdot \{\underline{\mathtt{A}}\}$$

we have palindrome ACATTGACCAGTTACA *(underlined) at $\hat{S}[1] \dots \hat{S}[5]$ which corresponds to GD palindome (8.5,8), palindrome* TTT *at $\hat{S}[1]$ (underlined twice) which corresponds to GD palindrome (2,1.5) and palindrome* TGGT *at $\hat{S}[2] \dots \hat{S}[3]$ (overlined) which corresponds to GD palindrome (5.5,2). Observe that GD palindrome (5.5,2) is maximal for center 5.5. Note that* CTGGTC *does not correspond to a valid GD palindrome as it neither starts at the beginning of a degenerate letter, nor ends at the end of a degenerate letter.*

In this section, we consider the following problem. Given a GD string $\hat{S}$ of length n, total size N, and total width W, find all GD strings $\hat{S}[i] \dots \hat{S}[j]$, with $0 \le i \le j \le n-1$, that are GD palindromes. We give two alternative algorithms: one finds all GD palindromes seeking them for all (i, j) pairs of starting and ending positions, and the

other one finds them starting from all possible centers. The two algorithms have different time complexities: which one is faster depends on W, N, and n. In fact, they compute all GD palindromes, but report only the maximal ones.

4.5.1 Algorithms for Computing GD Palindromes

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

Since there are $\mathcal{O}(n^2)$ pairs (i, j), and since by Theorem 18 algorithm GDSC takes time proportional to the total size of $\hat{S}[i] \dots \hat{S}[j]$ to check whether $\hat{S}[i] \dots \hat{S}[j]$ is a GD palindrome, algorithm MaxPalPairs takes $\mathcal{O}(n^2 N)$ time in total. In algorithm MaxPalCenters, we consider all possible centers c of $\hat{S}$. In the case when c is in between two degenerate letters we simply try to extend to the left and to the right via applying GDSC. In the case when c is inside a degenerate letter we intuitively split the letter vertically into two letters and try to extend to the left and to the right via applying GDSC. At each extension step of this procedure we maintain two GD strings $\hat{L}$ (left of the center) and $\hat{R}$ (right of the center) such that they are of the same total width. We consider the reverse of $\hat{L}$ (similar to algorithm MaxPalPairs) for the comparison. In the case where c occurs inside a degenerate letter to make sure we do not identify palindromes which do not exist, for all j split strings of the degenerate letter, we check that $\hat{L}^R[0][j][0 \dots k-1] = \hat{R}[0][j][0 \dots k-1]$ where $\hat{L}^R = rev(\hat{L})$ and $k = \min(w(L^R[0]), w(\hat{R}[0]))$. If no matches are found, we move onto the next center. Otherwise, when a match is found, we update $rev(\hat{L})$ and $\hat{R}$ with the remainder of the split degenerate letter (if its length is greater than k), as well as the next degenerate letters. Algorithm GDSC is applied to compare $rev(\hat{L})$ and $\hat{R}$. After a positive comparison, we overwrite $\hat{L}$ and $\hat{R}$ by adding the degenerate letters of the current extension until $w(\hat{L}) = w(\hat{R})$ (or until the end of the string is reached). This process is repeated as long as GDSC returns a positive comparison, that is, until the maximal GD palindrome with center c is found. The radius reported is then the total sum of all values of $w(\hat{L})$. If GDSC returns a negative comparison at center c, we proceed with the next center, because we clearly cannot have a GD palindrome centered at c extended further if $rev(\hat{L}) \cap \hat{R}$ is empty.

By Theorem 18 and the fact that there are $2W - 1$ possible centers, we have that algorithm MaxPalCenters takes $\mathcal{O}(WN)$ time in total. We obtain the following result.

Theorem 19. *Given a GD string of length n, total size N, and total width W, over an integer alphabet, all (maximal) GD palindromes can be computed in time $\mathcal{O}(\min\{W, n^2\}N)$.*

The problem that gained significant attention recently is the factorization of a string X of length n into a sequence of palindromes [23, 143, 312, 75, 29, 14]. We say that

$X_1, X_2, \ldots, X_\ell$ is a (maximal) palindromic factorization of string X, if every X_i is a (maximal) palindrome, $X = X_1 X_2 \ldots X_\ell$, and ℓ is minimal. In biological applications we need to factorize a sequence into palindromes in order to identify *hairpins*, patterns that occur in single-stranded DNA or, more commonly, in RNA. Next, we define and solve the same problem for GD strings.

Alatabbi et al. gave an off-line $\mathcal{O}(n)$-time algorithm for finding a maximal palindromic factorization of X [23]. Fici et al. presented an on-line $\mathcal{O}(n \log n)$-time algorithm for computing a palindromic factorization of X [143]; a similar algorithm was presented by I et al. [204]. In addition, Rubinchik and Shur [312] devised an $\mathcal{O}(n)$-sized data structure that helps to locate palindromes in a string; they also showed how it can be used to compute a palindromic factorization of X in $\mathcal{O}(n \log n)$ time. Borozdin et al. recently improved this by presenting an $\mathcal{O}(n)$-time algorithm [75]. Alzamel et al. considered the factorization problem for weighted sequences [29] and Adamczyk et al. considered the factorization problem with gaps and errors [14]. In what follows, we define and solve the same problem for GD strings.

Definition 11. *A* (maximal) GD palindromic factorization *of a GD string $\hat{S}$ is a sequence $\hat{P}_1, \ldots, \hat{P}_\ell$ of GD strings, such that: (i) every $\hat{P}_i$ is either a (maximal) GD palindrome or a degenerate letter of $\hat{S}$; (ii) $\hat{S} = \hat{P}_1 \ldots \hat{P}_\ell$; (iii) ℓ is minimal.*

After locating all (maximal) GD palindromes in $\hat{S}$ using Theorem 19, we are in a position to amend the algorithm of Alatabbi et al. [23] to find a (maximal) GD palindromic factorization of $\hat{S}$. We define a directed graph $\mathcal{G}_{\hat{S}} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{i \mid 0 \leq i \leq n\}$ and $\mathcal{E} = \{(i, j+1) \mid \hat{S}[i] \ldots \hat{S}[j]$ (maximal) GD palindrome of $\hat{S}\} \cup \{(i, i+1) | 0 \leq i < n\}$. Note that $\mathcal{V}$ contains a node n that is the sink of edges representing (maximal) GD palindromes ending at $\hat{S}[n-1]$. For maximal GD palindromes, $\mathcal{E}$ contains no more than $3W$ edges, as the maximum number of maximal GD palindromes is $2W - 1$. For GD palindromes, $\mathcal{E}$ contains $\mathcal{O}(n^2)$ edges, as the maximum number of GD palindromes is $\mathcal{O}(n^2)$. A shortest path in $\mathcal{G}_{\hat{S}}$ from 0 to n gives a (maximal) GD palindromic factorization. For maximal GD palindromes, the size of $\mathcal{G}_{\hat{S}}$ is $\mathcal{O}(W)$, as $n \leq W$, and so finding this shortest path requires $\mathcal{O}(W)$ time using a standard algorithm. For GD palindromes, the size of $\mathcal{G}_{\hat{S}}$, and thus the time, is $\mathcal{O}(n^2)$.

Theorem 20. *Given a GD string $\hat{S}$ of length n, total size N, and total width W, over an integer alphabet, a (maximal) GD palindromic factorization of $\hat{S}$ can be computed in time $\mathcal{O}(\min\{W, n^2\}N)$.*

Remark 2. *As a final remark, notice that a way to check whether a GD string is a GD palindrome is to compare it with its reverse. Since it is not a simple pattern matching between two GD strings, one can reduce the number of comparisons by comparing the r-th string in $\hat{S}[s]$ and the r-th string in $\hat{S}[n-1-s]^R$, for all $0 \leq s \leq n-1$. It means that the comparison is not free, but constrained to the positions that contain the first half of the palindrome contained in $\hat{S}$.*

4.5.2 Computing GD Palindromes in Protein Sequences

We have implemented algorithm MaxPalPairs in `C++`. We present here a proof-of-concept experiment but we anticipate that the algorithmic tools developed in this chapter are applicable in a wide range of biological applications.

	Hypervariable Region			
	I		II	
V	[366]	**This chapter**	[366]	**This chapter**
V_kII	18-27	11-36	119-130	118-131
	104-113	104-113	169-180	169-180
V_kIII	18-27	11-30	132-142	131-145
V_λII	63-74	62-81	140-152	140-152
V_λIII	51-74	50-75	132-143	131-144
V_λV	96-104	95-104	134-141	134-141

Table 4.1: Coordinates of (maximal) palindromes identified within hypervariable regions I and II.

We first obtained the amino acid sequences of 5 immunoglobulins within the human V regions [161] and converted these into mRNA sequences [319]. The letters $\mathtt{X}, \mathtt{S}, \mathtt{T}, \mathtt{Y}, \mathtt{Z}, \mathtt{R}$ and $\mathtt{H}$ were replaced by degenerate letters according to IUPAC [211]. Each other letter, $c \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$, was treated as a single degenerate letter $\{c\}$. An average of 47% of the total number of positions within the 5 sequences consisted of one of the following: $\mathtt{X}, \mathtt{S}, \mathtt{T}, \mathtt{Y}, \mathtt{Z}, \mathtt{R}$ and $\mathtt{H}$. We then used algorithm MaxPalPairs to find all maximal palindromes in the 5 sequences. Table 4.1 shows the palindromes identified within hypervariable regions I and II. Our results are in accordance with Wuilmart et al [366] who presented a *statistical* (fundamentally different) method to identify the location of palindromes within regions of immunoglobulin genes. The ranges we report are greater than or equal to the ones of [366] due to the *maximality* criterion used in the computation of GD palindromes by algorithm MaxPalPairs.

4.6 A Conditional Lower Bound under SETH

In this section, we show a conditional lower bound for computing palindromes in degenerate strings. Let us first define the 2-Orthogonal Vectors problem. Given two sets $A = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ and $B = \{\beta_1, \beta_2, \ldots, \beta_n\}$ of d-bit vectors, where $d = \omega(\log n)$, the *2-Orthogonal Vectors* problem asks the following question: is there any pair α_i, β_j of vectors that is orthogonal? Namely, is $\sum_{k=0}^{d-1} \alpha_i[k] \cdot \beta_j[k]$ equal to 0 for some $i, j \in [1, n]$? For the moderate dimension of this problem, we follow [162], assuming $n^{2-\epsilon} d^{\mathcal{O}(1)} \leq n^2 d$. The following result is known.

Theorem 21 ([162, 207, 208, 359])**.** *The 2-Orthogonal Vectors problem cannot be solved in $\mathcal{O}(n^{2-\epsilon} \cdot d^{\mathcal{O}(1)})$ time, for any $\epsilon > 0$, unless the* Strong Exponential Time Hypothesis *fails.*

We next show that the 2-Orthogonal Vectors problem can be reduced to computing maximal palindromes in degenerate strings thus obtaining a conditional lower bound similar to the upper bound obtained in Theorem 19 for computing all GD palindromes.

Theorem 22. *Given a degenerate string of length $4n$ over an alphabet of size $\sigma = \omega(\log n)$, all maximal GD palindromes cannot be computed in $\mathcal{O}(n^{2-\epsilon} \cdot \sigma^{\mathcal{O}(1)})$ time, for any $\epsilon > 0$, unless the* Strong Exponential Time Hypothesis *fails.*

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

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

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

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

4.7 Concluding Remarks and Open Problems

In this chapter we solved the problem of comparing two GD strings, that is, to decide whether two GD strings have a non-empty intersection. We then applied our string comparison tool to devise a simple algorithm for computing all palindromes in a GD string. We also complemented the latter algorithm by showing a similar conditional lower bound.

In Section 4.1, we sketched how automata can be used for comparing two ED strings. Recall that an ED string is a more general notion of degenerate string, where a degenerate letter generally contains strings of arbitrary lengths as well as the empty string. For GD strings, we showed that this comparison can be done in linear time for integer alphabets (Theorem 18). An interesting open problem is whether we can devise a more efficient (than the $\mathcal{O}(NM)$-time automata-based sketched in Section 4.1) approach for deciding whether the two languages represented by two ED strings of sizes N and M have a non-empty intersection; or, more generally, whether they share a sufficiently long substring.

Part II

Phylogenetic Trees

Chapter 5

A Rearrangement Distance for Fully-Labelled Trees

Key Points

Problem. The last decade brought a significant increase in the amount of data and a variety of new inference methods for reconstructing the detailed evolutionary history of various types of cancer. This brings the need of designing efficient procedures for comparing rooted trees representing the evolution of mutations in tumor phylogenies, a different model from the well-studied classical phylogenetic trees.

Model. We consider the simplest model of tumor phylogeny: a rooted *fully-labelled* tree, i.e., a tree such that nodes are biunivocally associated to a set of mutations. We introduce a rearrangement distance for fully-labelled trees. This notion originates from two operations: one that permutes the labels of the nodes, the other that affects the topology of the tree.

Included Works

This chapter combines the results of two papers: **A Rearrangement Distance for Fully-Labelled Trees** [52], which I presented at the *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, and its follow-up, **On Two Measures of Distance Between Fully-Labelled Trees** [51], which I presented at the *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*.

5.1 Introduction

Phylogenetic trees represent a plausible evolutionary relationship between the most disparate objects: natural languages in linguistics [176, 352, 283], ancient manuscripts in archaeology [85], genes and species in biology [200, 202]. The leaves of such trees are

labelled by the entities they represent, while the internal nodes are unlabelled and stand for unknown or extinct items. A great wealth of methods to infer phylogenies have been developed over the decades [141, 334], together with various techniques to compare the output of different algorithms, e.g., by building a consensus tree that captures the similarity between a set of conflicting trees [83, 215, 214, 166] or by defining a metric between two trees [127, 81, 137, 131, 309, 310].

Fully-labelled trees, in opposition to classical phylogenies, may model an evolutionary history where the internal nodes, just like the leaves, correspond to extant entities. An important phenomenon that fits this model well is cancer progression [188, 289]. With the increasing amount of data and algorithms becoming available for inferring cancer evolution [262, 217, 373, 71, 70], there is a pressing need of methods to provide a meaningful comparison among the trees produced by different approaches. Besides the well-studied edit distance for fully-labelled trees [338, 376, 292, 270], a few recent papers proposed ad-hoc metrics for tumor phylogenies [225, 174, 126, 100]. Taking inspiration from the existing literature [123, 74, 24, 334] on phylogeny rearrangement, the study of an operational notion of distance for rearranging a fully-labelled tree is of great interest, and there are still many unexplored questions to be answered.

In this work, we open the investigation of some notions of the rearrangement distance for two rooted trees which are fully labelled by the same set of labels. Following the existing literature [320, 334] on phylogeny rearrangement, we extend to several operations for rearranging a fully-labelled tree. The distance between a pair of trees is then the shortest sequence of these operations that transforms the first tree into the second tree. We consider rooted trees on n nodes labelled with distinct labels from $[n] = \{1, 2, \ldots, n\}$, and identify nodes with their labels. The first operation we introduce is an adaptation of the SPR operation [74] to a fully-labelled tree. We call it a link-and-cut operation and define it as follows.

- **link-and-cut operation**: given u, v and w such that v is a child of u and w is not a descendant of v, the link-and-cut operation $v \,|\, u \to w$ consists of two suboperations: cut the edge (v, u) and add the edge (v, w), effectively switching the parent of v from u to w.

We introduce a second operation that consists of a permutation of the labels of the tree (notice that such an operation does not really make sense on leaf-labelled phylogenies). We call this second operation a permutation operation and we define it as follows.

- **permutation operation**: apply some permutation $\pi : [n] \to [n]$ to the nodes. If a node u was a child of v before the operation, then after the operation $\pi(u)$ is a child of $\pi(v)$.

The size $|\pi|$ of a permutation is the number of elements x s.t. $\pi(x) \neq x$.

Two trees T_1 and T_2 are isomorphic if and only if one can reorder the children of every node so as to make the trees identical after disregarding the labels. The *permutation distance* $d_\pi(T_1, T_2)$ between two isomorphic trees is the smallest size $|\pi|$ of a permutation π that transforms T_1 into T_2.

For computing the permutation distance, in Section 5.3 we connect the complexity to that of calculating the largest cardinality matching in a sparse bipartite graph. By designing two-way reductions we show that these problems are equivalent, up

to polylogarithmic factors. Due to the recent progress in the area of fine-grained complexity we now know, for many problems that can be solved in polynomial time, what is essentially the best possible exponent in the running time, conditioned on some plausible but yet unproven hypothesis [363].

For max-flow, and more specifically maximum matching, this is not the case yet, although we do have some understanding of the complexity of the related problem of computing the max-flow between all pairs of nodes [9, 238, 8]. So, even though our reductions do not tell us what is the best possible exponent in the running time, they do imply that it is the same as for maximum matching in a sparse bipartite graph. In particular, by plugging in the asymptotically fastest known algorithm [254], we obtain an $\tilde{\mathcal{O}}(n^{4/3+o(1)})$ time algorithm for computing the permutation distance between two trees on n nodes. The main technical novelty in our reduction from permutation distance is that, even though the natural approach would result in multiple instances of weighted maximum bipartite matching, we manage to keep the graphs unweighted.

The size of a sequence of link-and-cut and permutation operations is the sum of the number of link-and-cut operations and the total size of all permutations. The *rearrangement distance* $d(T_1, T_2)$ between two (not necessarily isomorphic) trees with identical roots is the smallest size of any sequence of link-and-cut and permutation operations that, without permuting the root, transform T_1 into T_2.

In Section 5.4 we show that computing the rearrangement distance between two trees is NP-hard, via a reduction from 3-dimensional matching [224]. In Section 5.4.1 we consider the special case where one of the two tree is binary and give a simple linear-time 4-approximation ratio algorithm. In Section 5.4.2 we design a linear-time constant-factor approximation algorithm that does not assume that the trees are binary. The algorithm consists of multiple phases, each of them introducing more and more structure into the currently considered instance, while making sure that we do not pay more than the optimal distance times some constant. To connect the number of steps used in every phase with the optimal distance, we introduce a new combinatorial object that can be used to lower bound the latter, inspired by the well-known algorithm for computing the majority [76]. Finally, in Section 5.5 we show that there also exists a fixed parameter algorithm for the rearrangement distance.

5.2 Preliminaries

Let $[n] = \{1, 2, \ldots, n\}$. We consider rooted trees and forests on nodes labelled with distinct labels from $[n]$, and identify nodes with their labels. The parent of u in F is denoted $p_F(u)$, and we use the convention that $p_F(u) = \perp$ when u is a root in F. $F|u$ denotes the subtree of F rooted at u, $\mathsf{children}_F(u)$ stands for the set of children of a node u in F, and $\mathsf{level}_F(u)$ is the level of u in F (with the roots being on level 0).

Two trees T_1 and T_2 are isomorphic, denoted $T_1 \equiv T_2$, if and only if there exists a bijection μ between their nodes such that, for every $u \in [n]$ with $p_{T_1}(u) \neq \perp$, it holds that $\mu(p_{T_1}(u)) = p_{T_2}(\mu(u))$, implying in particular that μ maps the root of T_1 to the root of T_2. Let $\mathcal{I}(T_1, T_2)$ denote the set of all such bijections μ. Given two isomorphic trees T_1 and T_2, we seek a permutation π with the smallest possible size that transforms T_1 into T_2. This is equivalent to finding $\mu \in \mathcal{I}(T_1, T_2)$ that maximises the number of conserved nodes $\mathsf{conserved}(\mu) = \{u : u = \mu(u)\}$, as these two values sum up to n.

We now give the following notions of *distance* between trees T_1 and T_2 labelled by $[n]$. Unless otherwise stated, throughout this chapter we will assume that the roots of T_1 and T_2 are identical, and cannot participate in any permutation operation.

Definition 12 (link-and-cut distance)**.** *The* link-and-cut distance $d_\ell(T_1, T_2)$ *is the length of the shortest sequence of link-and-cut operations which transforms* T_1 *into* T_2*.*

The following Lemma ensures that the definition of link-and-cut distance is well posed.

Lemma 25. *Given trees* T_1 *and* T_2 *each labelled by* $[n]$*, there always exists a sequence of link-and-cut operations that transforms* T_1 *into* T_2*.*

Proof. For any node v, $p_{T_1}(v) = u$, such that $p_{T_2}(v) = w$ and w is a descendant of v in T_1 — and thus the operation $v|u \to w$ is not directly applicable — we prove that there exists a node z on the path from v to w in T_1 (including w) such that $p_{T_2}(z)$ is not a descendant of v in T_2 nor a descendant of z in T_1. This implies that after applying the valid operation $z|p_{T_1}(z) \to p_{T_2}(z)$, the operation $v|u \to w$ becomes valid too. There is always such a node z because, should it not exist, w would be a descendant of v also in T_2, giving rise to the cycle $(w \to v \to \cdots \to w)$ and thus contradicting the fact that T_2 is a tree. □

Definition 13 (permutation distance)**.** *The* permutation distance $d_\pi(T_1, T_2)$ *is the size* $|\pi|$ *of the smallest permutation* π *that transforms* T_1 *into* T_2*.*

Definition 14 (rearrangement distance)**.** *The* rearrangement distance $d(T_1, T_2)$ *is the smallest size of any sequence of link-and-cut and permutation operations that transforms* T_1 *into* T_2*.*

Clearly, the permutation distance $d_\pi(T_1, T_2)$ is defined only when T_1 and T_2 are isomorphic, and it is evidently well posed. As a direct consequence of this and Lemma 25, the definition of rearrangement distance is also well posed. Moreover, since these operations are invertible, all the above distance measures are symmetric, and they satisfy by definition the triangle inequality: consider, e.g., the rearrangement distance. Given T_1, T_2 and T_3 labelled by the same set of labels, let $\mathcal{S}_{12}$ be a sequence that transforms T_1 into T_2 such that $|\mathcal{S}_{12}| = d(T_1, T_2)$, $\mathcal{S}_{23}$ a sequence that transforms T_2 into T_3 with $|\mathcal{S}_{23}| = d(T_2, T_3)$, $\mathcal{S}_{13}$ a sequence that transforms T_1 into T_3 and $|\mathcal{S}_{13}| = d(T_1, T_3)$. It is evident that the concatenation $\mathcal{S}_{12}\mathcal{S}_{13}$ of $\mathcal{S}_{12}$ and $\mathcal{S}_{23}$ is a sequence that transforms T_1 into T_3, and by Definition 14 its size is larger or equal to $d(T_1, T_3)$: thus $d(T_1, T_2) + d(T_2, T_3) \geq |\mathcal{S}_{12}\mathcal{S}_{23}| \geq d(T_1, T_3)$. A similar argument shows that the triangular inequality also holds for the link-and-cut distance and the permutation distance.

We next define two structures that capture two fundamental characteristics of the trees to compare.

Definition 15 (active set)**.** *Given trees* T_1 *and* T_2*, we call* active *the subset* $\mathcal{X} \subseteq [n]$ *of labels which have different parents in* T_1 *and* T_2*, i.e.,* $v \in \mathcal{X}$ *iff* $p_{T_1}(v) \neq p_{T_2}(v)$*.*

Given trees T_1 and T_2, for each vertex v of the active set $\mathcal{X}$, we can associate with v the pair $(p_{T_1}(v), p_{T_2}(v))$ of the parents of v in the two trees. Let $\mathcal{P}_{(u,w)}$ be the set

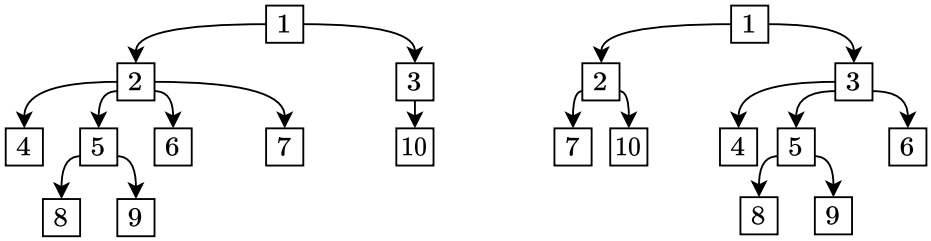


Figure 5.1: Trees T_1 and T_2 labelled by $[10]$. The link-and-cut distance $d_\ell(T_1, T_2)$ is 4, given, for example, by the sequence $4|2 \to 3$, $5|2 \to 3$, $6|2 \to 3$, $10|3 \to 2$. The rearrangement distance is 3, given, for example, by the sequence $(2\ 3), 7|3 \to 2$.

$\{v \ : \ p_{T_1}(v) = u, \ p_{T_2}(v) = w\}$: since each vertex has exactly one parent in each tree, each vertex $v \in \mathcal{X}$ belongs to one and only one set $\mathcal{P}_{(u,w)}$. This fact is formalized in the following definition and illustrated in Example 18.

Definition 16 (family partition)**.** *Let trees T_1 and T_2 each be labelled by $[n]$: for each vertex $v \in \mathcal{X}$ we denote the set $\mathcal{P}_{(u,w)} = \{v \ : \ p_{T_1}(v) = u, \ p_{T_2}(v) = w\}$. Then $\mathcal{P}$ is the partition of set $\mathcal{X}$ into the nonempty sets $\mathcal{P}_{(u,w)}$, $u, w \in V$. Partition $\mathcal{P}$ is called the* family partition *of the active set $\mathcal{X}$, and we denote its* size *$|\mathcal{P}|$ as the number of different (non-empty) subsets $\mathcal{P}_{(u,w)}$ it is composed of.*

Example 18. *Consider T_1 and T_2 of Figure 5.1. The active set is $\mathcal{X} = \{4, 5, 6, 10\}$. The family partition is composed of the following sets: $\mathcal{P}_{(2,3)} = \{4, 5, 6\}$, $\mathcal{P}_{(3,2)} = \{10\}$.*

Note that the family partition encodes the elements of any shortest sequence of link-and-cut operations for transforming T_1 into T_2: $v \in \mathcal{P}_{(u,w)}$ corresponds to operation $v|u \to w$. It is easy to see, from the proof of Lemma 25, that a shortest sequence of valid link-and-cut operations can be obtained from $\mathcal{P}$ by ordering the set of operations it encodes with respect to a *depth-first traversal* (DFT) of T_1: $u|p_{T_1}(u) \to p_{T_2}(u)$ precedes $v|p_{T_1}(v) \to p_{T_2}(v)$ if u precedes v in a DFT of T_1. Hence $d_\ell(T_1, T_2) = |\mathcal{X}|$, i.e., the link-and-cut distance is equal to the cardinality of the active set, of which $\mathcal{P}$ is a partition.

In Section 5.4.2, for ease of presentation, instead of the link-and-cut operation we will work with the cut operation, defined as follows:

- **cut operation**: let u, v be two nodes such that v is a child of u. The cut operation $(v \dagger u)$ removes the edge (v, u), effectively making v a root.

The size of a sequence of cut and permutation operations is defined similarly as for a sequence of link-and-cut and permutation operations. Since a permutation operation is essentially just renaming the nodes, we can assume that all permutation operations precede all link-and-cut (or cut) operations, or vice versa. Furthermore, multiple consecutive permutation operations can be replaced by a single permutation operation without increasing the total size.

This leads to the notion of rearrangement distance between two forests F_1 and F_2. We write $F_1 \sim F_2$ to denote that, for every $u \in [n]$, at least one of the following three conditions holds: (i) $p_{F_1}(u) = p_{F_2}(u)$, (ii) $p_{F_1}(u) = \perp$, or (iii) $p_{F_2}(u) = \perp$.

The rearrangement distance $\tilde{d}(F_1, F_2)$ is the smallest size of any sequence of cut and permutation operations that transforms F_1 into F_1' such that $F_1' \sim F_2$. This is the same as the smallest size of any sequence of cut and permutation operations that transforms F_2 into F_2' such that $F_1 \sim F_2'$, as both sizes are equal to the minimum over all permutations π that fix the original root of the following expression

$$|\{u : \pi(u) \neq u\}| + |\{u : p_{F_1}(u) \neq p_{F_2}(\pi(u)) \ \wedge \ p_{F_1}(u) \neq \perp \ \wedge \ p_{F_2}(\pi(u)) \neq \perp\}|.$$

Consequently, $\tilde{d}$ defines a metric. In Section 5.4.2 we connect $d(T_1, T_2)$ and $\tilde{d}(T_1, T_2)$, and then work with the latter.

A matching in a bipartite graph is a subset of edges with no two edges meeting at the same vertex. A maximum matching in an unweighted bipartite graph is a matching of maximum cardinality, whereas a maximum weight matching in a weighted bipartite graph is a matching in which the sum of weights is maximised. Given an unweighted bipartite graph with m edges, the well-known algorithm by Hopcroft and Karp [197] finds a maximum matching in $\mathcal{O}(m^{1.5})$ time. This has been recently improved by Liu and Sidford to $\tilde{\mathcal{O}}(m^{4/3+o(1)})$ [254].

A *heavy path decomposition* of a tree T is obtained by selecting, for every non-leaf node $u \in T$, its *heavy child* v such that $T|v$ is the largest: there will be some subtlety in how to resolve a tie in this definition that will be explained in detail later. This procedure decomposes the nodes of T into node-disjoint paths called *heavy paths*. Each heavy path p starts at some node, called its *head*, and ends at a leaf: $\mathsf{head}_T(u)$ denotes the head of the heavy path containing a node u in T. An important property of such a decomposition is that the number of distinct heavy paths above any leaf (that is, intersecting the path from a leaf to the root) is only logarithmic in the size of T [328].

5.3 Permutation Distance

Our aim is to find $\mu \in \mathcal{I}(T_1, T_2)$ that maximises $\mathsf{conserved}(\mu)$, that is $\gamma(T_1, T_2) = \max\{\mathsf{conserved}(\mu) : \mu \in \mathcal{I}(T_1, T_2)\}$. To make the notation less cluttered, we define $\gamma(x, y) = \gamma(T_1|x, T_2|y)$. Let us start by describing a simple polynomial time algorithm which illustrates the basic idea behind the procedure. We will then show how to improve it to obtain a faster algorithm that uses unweighted bipartite maximum matching. Finally, we will show a reduction from bipartite maximum matching to computing the permutation distance, establishing that these two problems are in fact equivalent, up to polylogarithmic factors.

5.3.1 Polynomial Time Algorithm

We first run the folklore linear-time algorithm of [21] for determining if two rooted trees are isomorphic. Recall that this algorithm assigns a number from $\{1, 2, \ldots, 2n\}$ to every node of T_1 and T_2 so that the subtrees rooted at two nodes are isomorphic if and only if their numbers are equal. The high-level idea is then to consider a weighted bipartite graph $G(u, v)$ for each $u, v \in [n]$ such that $\mathsf{level}_{T_1}(u) = \mathsf{level}_{T_2}(v)$ and $T_1|u \equiv T_2|v$. The vertices of $G(u, v)$ are $\mathsf{children}_{T_1}(u)$ and $\mathsf{children}_{T_2}(v)$, and there is an edge of weight $\gamma(u', v')$ between $u' \in \mathsf{children}_{T_1}(u)$ and $v' \in \mathsf{children}_{T_2}(v)$ if and only if $T_1|u' \equiv T_2|v'$

and $\gamma(u', v') > 0$. We call such graphs the *distance graphs* for T_1 and T_2 and denote them collectively by $\mathcal{G}(T_1, T_2)$. See Figure 5.3 for an example.

The weights $\gamma(u, v)$ are computed as follows, with $\mathcal{M}(G(u, v))$ denoting the weight of a (not necessarily perfect) maximum weight matching in $G(u, v)$, and $\Gamma : [n] \times [n] \to \{0, 1\}$ being a function such that $\Gamma(u, v) = 1$ if $u = v$ and $\Gamma(u, v) = 0$ otherwise.

$$\gamma(u, v) = \begin{cases} \mathcal{M}(G(u, v)) + \Gamma(u, v) & \text{if } T_1|u \equiv T_2|v, \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

The overall number of edges created in all graphs is $\mathcal{O}(n^2)$. Indeed, for each $u \in [n]$ such that $\mathsf{level}_{T_1}(u) = \mathsf{level}_{T_2}(u)$ and $T_1|u \equiv T_2|u$, and for each pair of ancestors z of u in T_1 and w of u in T_2 such that $\mathsf{level}_{T_1}(z) = \mathsf{level}_{T_2}(w)$ and $T_1|z \equiv T_2|w$, we possibly add an edge (z, w) to the graph $G(p_{T_1}(z), p_{T_2}(w))$. Since there are up to n pairs of ancestors on the same level for each label, and the labels are n, there are $\mathcal{O}(n^2)$ edges overall.

We then start from the deepest level in both trees, and we move up level by level towards the roots in both trees simultaneously. For each level k, we consider all pairs of isomorphic subtrees rooted at level k, build the corresponding distance graphs, and use Equation (5.1) to weigh the edges. After having reached the roots, we return the value of $\gamma(T_1, T_2)$. The correctness of the algorithm is given by the following lemmas, the first one stating that the permutation distance is equal to the minimum number of labels that are not conserved by any isomorphic mapping.

Lemma 26. *For any two isomorphic trees T_1, T_2, each labelled by $[n]$, it holds that $d_\pi(T_1, T_2) = n - \gamma(T_1, T_2)$.*

Proof. Consider an isomorphic mapping μ from T_1 to T_2 that has the minimum number of mismatched labels $\Delta(\mu) = n - \gamma(T_1, T_2)$ and consider the set of labels of the vertices involved in the set of mismatching vertices given by μ.

Clearly, such labels are in a permutation π which rearranges the labels of tree T_1 to obtain T_2, while, by construction of μ, all the other labels will not be perturbed by π. Then we need to show that such a permutation rearranges the minimum number of distinct labels, that is, its size $|\pi| = d_\pi(T_1, T_2)$ is the permutation distance. Indeed, assume to the contrary that the permutation distance $d_\pi(T_1, T_2) < |\pi|$. This implies the existence of a permutation π' that rearranges fewer labels than π, i.e., $|\pi'| < |\pi|$. Then we show that there exists an isomorphic mapping μ' that has mismatch number less than the one of μ, contradicting the initial assumption.

Indeed, consider the permutation π' and define the mapping μ' from T_1 to T_2 such that $\mu'(u) = v$ whenever $\pi'(v) = u$ The mapping μ' is an isomorphism by the construction of π', since $\mu'(\pi'(u)) = u$ for all nodes u of T_1, and with the application of π', the two trees are congruent, hence congruency of the labels implies isomorphism of the two trees. This concludes the proof that μ' is an isomorphism thus leading to a contradiction. □

Lemma 27. *Recursion 5.1 is correct.*

Proof. It is essentially a proof by induction. Recall that $\gamma(u, v)$ is the maximum number of labels conserved by any isomorphism between $T_1|u$ and $T_2|v$. If both u and v are

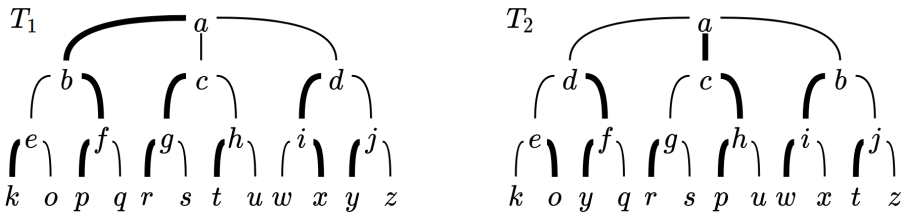


Figure 5.2: T_1 and T_2 with a possible heavy path decomposition.

leaves, then $T_1|u \equiv T_2|v$ is trivially true and $\gamma(u,v) = \Gamma(u,v)$. When $T_1|u \equiv T_2|v$ does not hold, then the permutation distance (thus $n - \gamma(u,v)$, by Lemma 26) is undefined.

Otherwise, if u and v are internal vertices, and $T_1|u \equiv T_2|v$, then let μ be a bijective mapping from the nodes of $T_1|u$ to the nodes of $T_2|v$ maximizing $\mathsf{conserved}(\mu)$. By the definition of γ and the construction of μ, $\gamma(u,v) = \sum_{z \in \mathsf{children}(u)} \gamma(z, \mu(z)) + \Gamma(u,v)$. By the inductive hypothesis, all values $\gamma(z,\mu(z))$ are correctly computed; since μ is the bijective mapping maximizing the conserved labels, no other mapping μ' can achieve a larger value of $\sum_{z \in \mathsf{children}(u)} \gamma(z,\mu(z))$, and so $\mu(z)$ is determined by a maximum weight matching on $G(u,v)$. □

The running time is polynomial if we plug in any polynomial-time maximum weight matching algorithm.

In the next subsection we show how to obtain a better running time by constructing a different version of distance graphs, so that the total weight of their edges will be subquadratic, and replacing maximum weight matching with maximum matching.

5.3.2 Reduction to Bipartite Maximum Matching

We start by finding a heavy path decomposition of T_1 and T_2, with some extra care in resolving a tie if there are multiple children with subtrees of the same size, as follows. Recall that we already know which subtrees of T_1 and T_2 are isomorphic, as the algorithm of [21] assigns the same number from $\{1, 2, \ldots, 2n\}$ to nodes of T_1 and T_2 with isomorphic subtrees. For every $u, v \in [n]$ such that $T_1|u \equiv T_2|v$, we would like the heavy child u' of u in T_1 and v' of v in T_2 to be such that $T_1|u' \equiv T_2|v'$. This can be implemented in linear time: it suffices to group the nodes with isomorphic subtrees together, and then make the choice just once for every such group.

Consider a graph $G(u,v)$ for some $u, v \in [n]$: the edge corresponding to the heavy child u' of u in T_1 and the heavy child v' of v in T_2 is called *special* (note that this edge might not exist). The key observation is that the properties of heavy path decomposition allow us to bound the total weight of non-special edges by $\mathcal{O}(n \log n)$.

Lemma 28. *The total weight of all non-special edges in $\mathcal{G}(T_1, T_2)$ is $\mathcal{O}(n \log n)$.*

Proof. Consider any $u \in [n]$ such that $\mathsf{level}_{T_1}(u) = \mathsf{level}_{T_2}(u)$ and $T_1|u \equiv T_2|u$. For each pair of ancestors z of u in T_1 and w of u in T_2 such that $\mathsf{level}_{T_1}(z) = \mathsf{level}_{T_2}(w)$, $T_1|z \equiv T_2|w$ and either $\mathsf{head}_{T_1}(z) = z$ or $\mathsf{head}_{T_2}(w) = w$, u will contribute 1 to the weight of an edge (z,w) in $G(p_{T_1}(z), p_{T_2}(w))$. Because there are at most $\log n$ heavy paths above any node of T_1 or T_2, each label $u \in [n]$ contributes 1 to the weight of at most $2 \log n$ non-special edges, making their total weight $\mathcal{O}(n \log n)$ overall. □

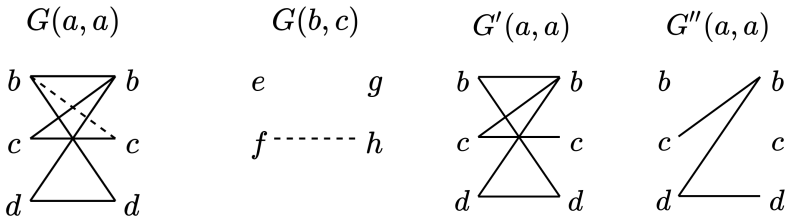


Figure 5.3: $G(a,a)$ (type 1), $G'(a,a)$, $G''(a,a)$ and $G(b,c)$ (type3) for the trees of Figure 5.2. The special edge in each graph is dashed.

We divide the graphs in $\mathcal{G}(T_1,T_2)$ into three types (see Figure 5.3, left, for an example):

Type 1: graphs $G(u,v)$ with at least one non-special edge.

Type 2: graphs $G(u,v)$ with no non-special edges, and $\Gamma(u,v)=1$.

Type 3: graphs $G(u,v)$ with no non-special edges, and $\Gamma(u,v)=0$.

We will construct only the graphs of type 1 and 2, and extract from them the information that the graphs of type 3 would have captured. In what follows we show how to construct the graphs of type 1 and 2 in $\mathcal{O}(n\log^2 n)$ time.

Constructing the Graphs of Type 1 and 2. The first step is to find all pairs of nodes that correspond to graphs of type 1 or 2, and store them in a dictionary D implemented as a balanced search tree with $\mathcal{O}(\log n)$ access time. The second step is to find the non-special edges of these graphs, and store them in a separate dictionary, also implemented as a balanced search tree with $\mathcal{O}(\log n)$ access time. Note that the weights will be found at a later stage of the algorithm. We assume that both trees have been already decomposed into heavy paths, and we already know which subtrees are isomorphic. This can be preprocessed in $\mathcal{O}(n)$ time.

Lemma 29. *All graphs of type 1 and 2 can be identified in $\mathcal{O}(n\log^2 n)$ time.*

Proof. We consider every $u\in[n]$ such that $\mathsf{level}_{T_1}(u)=\mathsf{level}_{T_2}(u)$ and $T_1|u\equiv T_2|u$ in two passes. In the first pass, we need to iterate over every ancestor z of u in T_1 and w of u in T_2 such that $\mathsf{level}_{T_1}(z)=\mathsf{level}_{T_2}(w)$, $T_1|z\equiv T_2|w$ and either $\mathsf{head}_{T_1}(z)=z$ or $\mathsf{head}_{T_2}(w)=w$, and if additionally $T_1|p_{T_1}(z)\equiv T_2|p_{T_2}(w)$ then designate $G(p_{T_1}(z),p_{T_2}(w))$ to be a graph of type 1 and insert it into D. As a non-special edge (z,w) of a graph $G(p_{T_1}(z),p_{T_2}(w))$ is such that either z or w are not on the same heavy path as their parents, this correctly determines all graphs of type 1.

To efficiently iterate over all such z and w given u, we assume that the nodes of every heavy path of a tree T are stored in an array, so that, given any node $u\in T$, we are able to access the node that belongs to the same heavy path as u and whose level is ℓ in constant time, if it exists. We denote such operation $\mathsf{access}_T(u,\ell)$. Given two nodes $u\in T_1$ and $v\in T_2$ on the same level, the procedure below shows how to iterate over every ancestor z of u and w of v such that $\mathsf{level}_{T_1}(z)=\mathsf{level}_{T_2}(w)$ and either $\mathsf{head}_{T_1}(z)=z$ or $\mathsf{head}_{T_2}(w)=w$, in $\mathcal{O}(\log n)$ time, implying that all graphs of type 1 can be identified in $\mathcal{O}(n\log^2 n)$ time.

```
1  while u ≠ ⊥ and v ≠ ⊥ do
2      if level_{T1}(head_{T1}(u)) < level_{T2}(head_{T2}(v)) then
3          output access_{T1}(u, level_{T2}(head_{T2}(v))) and head_{T2}(v);
4          v ← p_{T2}(head_{T2}(v));
5      if level_{T1}(head_{T1}(u)) > level_{T2}(head_{T2}(v)) then
6          output head_{T1}(u) and access_{T2}(v, level_{T1}(head_{T1}(u)));
7          u ← p_{T1}(head_{T1}(u));
8      else
9          output head_{T1}(u) and head_{T2}(v);
10         u ← p_{T1}(head_{T1}(u));
11         v ← p_{T2}(head_{T2}(v));
```

In the second pass, for each $u \in [n]$ such that $\mathsf{level}_{T_1}(u) = \mathsf{level}_{T_2}(u)$ and $T_1|u \equiv T_2|u$, we designate $G(u,u)$ to be a graph of type 2, unless it has been already designated to be a graph of type 1. □

Lemma 30. *All graphs of type 1 and 2 can be populated with their edges in $\mathcal{O}(n \log^2 n)$ time.*

Proof. For each such graph $G(u,v)$ such that none of u, v is a leaf, let u' be the unique heavy child of u, and v' be the unique heavy child of v. We add the special edge (u', v') to $G(u,v)$. To find the non-special edges, we again consider every $u \in [n]$ such that $\mathsf{level}_{T_1}(u) = \mathsf{level}_{T_2}(u)$ and $T_1|u \equiv T_2|u$: we iterate over the ancestors z of u in T_1 and w of u in T_2 such that $\mathsf{level}_{T_1}(z) = \mathsf{level}_{T_2}(w)$, $T_1|z \equiv T_2|w$ and either $\mathsf{head}_{T_1}(z) = z$ or $\mathsf{head}_{T_2}(w) = w$, and if additionally $T_1|p_{T_1}(z) \equiv T_2|p_{T_2}(w)$ then add a non-special edge (z,w) to $G(p_{T_1}(z), p_{T_2}(w))$. This takes $\mathcal{O}(n \log^2 n)$ time overall. □

Processing the Graphs of Type 1 and 2. Having constructed the graphs of type 1 and 2 in $\mathcal{O}(n \log^2 n)$ time, we process them level by level. Consider $G(u,v)$: for each of its edges (u', v') corresponding to $u' \in \mathsf{children}_{T_1}(u)$ and $v' \in \mathsf{children}_{T_2}(v)$, we need to extract its weight $\gamma(u', v')$. If $G(u', v')$ is of type 1 or 2, the graph can be extracted from the dictionary in $\mathcal{O}(\log n)$ time. Otherwise, $G(u', v')$ is of type 3 and we need to make up for not having processed such graphs.

To this aim, we associate a sorted list of levels with each pair of heavy paths of T_1 and T_2. The lists are stored in a dictionary indexed by the heads of the heavy paths. For every $u, v \in [n]$ such that $G(u,v)$ is of type 1 or 2, we append the levels of u and v to the lists associated with the respective heavy paths. The lists can be constructed in $\mathcal{O}(n \log^2 n)$ time by processing the graphs level by level, and allow us to efficiently use the following lemma.

Lemma 31. *Consider $u, v \in [n]$ such that $\mathsf{level}_{T_1}(u) = \mathsf{level}_{T_2}(v)$ and $T_1|u \equiv T_2|v$, but $G(u,v)$ is of type 3. Either both u and v are leaves and $\gamma(u,v) = 0$, or the heavy child of u is u', the heavy child of v is v', and $\gamma(u,v) = \gamma(u', v')$.*

Proof. First observe that $u \neq v$, as otherwise $G(u,v)$ would be of type 2. Becase $T_1|u \equiv T_2|v$, either both u and v are leaves or none of them is a leaf. In the former

case, $G(u,v)$ is empty and $\gamma(u,v) = 0$. By how we resolve ties in the heavy path decomposition, in the latter case we have $T_1|u' \equiv T_2|v'$, where u' is the heavy child of u and v' is the heavy child of v. $G(u,v)$ consists of the unique special edge corresponding to the heavy child u' of u and v' of v, so $\mathcal{M}(G(u,v))$ is equal to the cost of the special edge, and by (5.1) we obtain that $\gamma(u,v) = \gamma(u',v')$. □

Given $u, v \in [n]$ such that $\mathsf{level}_{T_1}(u) = \mathsf{level}_{T_2}(v) = \ell$ and $T_1|u \equiv T_2|v$, we extract $\gamma(u,v)$ by accessing the sorted list associated with the heavy paths of u and v: we binary search for the smallest level $\ell' \geq \ell$ such that the heavy paths of u and v respectively contain a node u' and v', both on level ℓ', with $G(u',v')$ of type 1 or 2. Then Lemma 31, together with the fact that in our heavy path decomposition the subtrees rooted at the heavy children of two nodes with isomorphic subtrees are also isomorphic, implies that $\gamma(u,v) = \gamma(u',v')$.

It remains to describe how to compute $\mathcal{M}(G(u,v))$ for every graph $G(u,v)$ of type 1 and 2. We could have used any maximum weight matching algorithm, but this would result in a higher running time. Our goal is to plug in a maximum matching algorithm. This seems problematic as $G(u,v)$ is a weighted bipartite graph, but we will show that maximum weight matching can be reduced to multiple instances of maximum matching. However, bounding the overall running time will require bounding the total weight of all edges belonging to graphs of type 1 and 2. By Lemma 28 we already know that the total weight of all non-special edges is $\mathcal{O}(n \log n)$, but such bound doesn't hold for the special edges. Therefore, we proceed as follows. Let u' be the heavy child of u and v' be the heavy child of v. We construct $G'(u,v)$ by removing the special edge from $G(u,v)$. We also construct $G''(u,v)$ from $G(u,v)$ by removing all the edges incident to u' and v' (see Figure 5.3 for an example). Equation (5.1) can then be rewritten as follows:

$$\gamma(u,v) = \max\{\mathcal{M}(G'(u,v)), \mathcal{M}(G''(u,v)) + \gamma(u',v')\} + \Gamma(u,v). \tag{5.2}$$

This is because a maximum weight matching in $G(u,v)$ either includes the special edge (u',v'), implying that no other edges incident to u' and v' can be part of the matching and thus $\mathcal{M}(G(u,v)) = \mathcal{M}(G''(u,v)) + \gamma(u',v')$, or it does not include it, thus $\mathcal{M}(G(u,v)) = \mathcal{M}(G'(u,v))$. Since the graphs $G'(u,v)$ and $G''(u,v)$ contain only non-special edges, the overall weight of all edges in the obtained instances of maximum weight matching is $\mathcal{O}(n \log n)$.

We already know that constructing all the relevant graphs takes $\mathcal{O}(n \log^2 n)$ time. It remains to analyze the time to calculate the maximum weight matching in every $G'(u,v)$ and $G''(u,v)$. We first present a preliminary lemma that connects the complexity of calculating the maximum weight matching in a weighted bipartite graph to the complexity of calculating the maximum matching in an unweighted bipartite graph.

Lemma 32 ([220])**.** *Let G be a weighted bipartite graph, and let N be the total weight of all the edges of G. Calculating the maximum weight matching in G can be reduced in $\mathcal{O}(N)$ time to multiple instances of calculating the maximum matching in an unweighted bipartite graph, in such a way that the total number of edges in all such graphs is at most N.*

Proof. Using the decomposition theorem of Kao, Lam, Sung, and Ting [220], we can reduce computing the maximum weight matching in a weighted bipartite graph such

that the total weight of all edges is N to multiple instances of calculating the largest cardinality matching in an unweighted bipartite graph. The total number of edges in all unweighted bipartite graphs is $\sum_i m_i = N$ and the reduction can be implemented in $\mathcal{O}(N)$ time by maintaining a list of edges with weight w, for every $w = 1, 2, \ldots, N$. □

Theorem 23. *Let $f(m)$ be the complexity of calculating the maximum matching in an unweighted bipartite graph on m edges, and let $f(m)/m$ be nondecreasing. The permutation distance can be computed in $\tilde{\mathcal{O}}(f(n))$ time.*

Proof. The total number of edges in all constructed graphs is $\mathcal{O}(n \log n)$, and the total time to construct the relevant graphs and extract the costs of their edges is $\mathcal{O}(n \log^2 n)$. Thus, the total running time is $\mathcal{O}(n \log^2 n)$ plus the time to compute the maximum weight matching in every graph of type 1 and type 2. Let N_i be the total weight of all non-special edges in the i-th of these graphs. By Lemma 28, $\sum_i N_i = \mathcal{O}(n \log n)$. Additionally, $N_i \leq n$. Let $m_{i,j}$ be the number of edges in the j-th instance of unweighted bipartite matching for the i-th graph. By Lemma 32, the overall time is hence $\sum_{i,j} f(m_{i,j})$, where $\sum_{i,j} m_{i,j} \leq \sum_i N_i = \mathcal{O}(n \log n)$ and $m_{i,j} \leq N_i \leq n$. We upper bound $\sum_{i,j} f(m_{i,j})$ using the assumption that $f(m)/m$ is nondecreasing as follows:

$$\sum_{i,j} f(m_{i,j}) = \sum_{i,j} m_{i,j} \cdot f(m_{i,j})/m_{i,j} \leq \sum_{i,j} m_{i,j} \cdot f(n)/n = \mathcal{O}(f(n) \log n). \qquad \square$$

Corollary 24. *The permutation distance can be computed in $\tilde{\mathcal{O}}(n^{4/3+o(1)})$ time.*

5.3.3 Reduction from Bipartite Maximum Matching

We complement the algorithm described in Subection 5.3.2 with a reduction from bipartite maximum matching to computing the permutation distance: see Figure 5.4 for an example.

Theorem 25. *Given an unweighted bipartite graph on m edges, we can construct in $\mathcal{O}(m)$ time two trees with permutation distance equal to the cardinality of the maximum matching.*

Proof. We first modify the graph so that the degree of every node is at most 3. This can be ensured in $\mathcal{O}(m)$ time by repeating the following transformation: take a node u with neighbours $v_1, v_2, \ldots, v_k$, $k \geq 4$. Replace u with u' and u'' both connected to a new node v, connect u' to $v_1, v_2, \ldots, v_{k-2}$ and u'' to v_{k-1}, v_k. It can be verified that the cardinality of the maximum matching in the new graph is equal to that in the original graph increased by 1. By storing, for every node, the incident edges in a linked list, we can implement a single step of this transformation in constant time, and there are at most m steps.

We will now first construct two unlabelled trees and then explicitly assign appropriate labels to their nodes. Without loss of generality, let the nodes of the graph be $u_1, u_2, \ldots, u_m$ and $v_1, v_2, \ldots, v_m$. In the first tree we create m nodes, labelled with $u_1, u_2, \ldots, u_m$, connected to a common unlabelled root. In the second tree we do the same with nodes $v_1, v_2, \ldots, v_m$. Then, for every edge (u_i, v_j) of the graph, we attach a

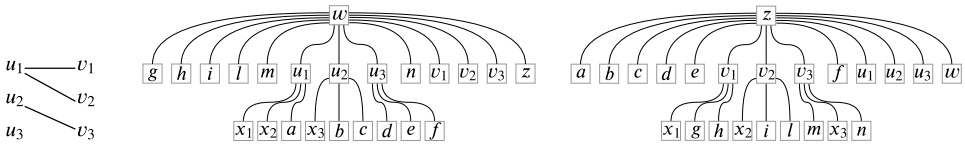


Figure 5.4: The two trees built for the graph on the left, according to Theorem 25.

new leaf to u_i in the first tree and to v_j in the second tree, and assign the same label to both of them. Finally, we attach enough unlabelled leaves to every u_i and v_j to make their degrees all equal to 3. To make both trees fully-labelled on the same set of labels, we further attach $1 + m + 3m - m = 3m + 1$ *extra leaves* to the roots of both trees. For every unlabelled leaf attached to $u_1, u_2, \ldots, u_m$ of the first tree, we choose an unlabelled extra leaf of the second tree, and assign the same label to both of them. We then assign the same label to the root of the first tree and an extra leaf of the second tree, and label the last m extra leaves of the second tree with $u_1, u_2, \ldots, u_m$. We finally swap the trees and repeat the same procedure: see Figure 5.4 for an example.

The permutation distance between the two trees is equal to the cardinality of the maximum matching. Indeed, the trees are clearly isomorphic; moreover, any isomorphism must match extra leaves with extra leaves, and every u_i to a $v_{\pi(j)}$, for some permutation π on $[m]$. The extra leaves do not contribute to the number of conserved nodes, while u_i and $v_{\pi(j)}$ contribute 1 if and only if $(u_i, v_{\pi(j)})$ was an edge in the original graph. Thus, the distance is equal to the maximum over all permutations π of the number of edges $(u_i, v_{\pi(j)})$. This in turn is equal to the cardinality of the maximum matching in the original graph. □

5.4 Rearrangement Distance

We first show that deciding the rearrangement distance between two trees is NP-hard. We show this by reduction from 3-dimensional matching, one of Karp's 21 NP-complete problems [224].

In 3-dimensional matching, we are given three disjoint sets A, B and C, along with a set $\mathcal{T}$ of triples (a, b, c), such that $a \in A$, $b \in B$ and $c \in C$: essentially, a 3-uniform hypergraph H. A *matching* is then a subset $\mathcal{M} \subseteq \mathcal{T}$ such that for every two triples $(a, b, c) \in \mathcal{M}$, $(a', b', c') \in \mathcal{M}$, it follows that $a \neq a'$, $b \neq b'$ and $c \neq c'$, that is, all triples of $\mathcal{M}$ are pairwise disjoint. It is then NP-hard to decide for a given k if there is a matching $\mathcal{M}$ of size k [224]. It has been proved that the problem remains NP-hard even in the case of 3-bounded 1-common 3-dimensional matching, which is a restriction of the problem where the number of occurrences of an element in the triples is at most 3, and each pair of triples has at most one element in common [216]. We also make use of the following structure in this proof.

Definition 17 (movements graph)**.** *Given trees T_1 and T_2, the* movements graph G *has an edge for every element $\mathcal{P}_{(u,w)}$ of the family partition $\mathcal{P}$ of T_1 and T_2, that is, $E_G = \{(u, w) \ : \ \mathcal{P}_{(u,w)} \in \mathcal{P}\}$, while its vertex set is $V_G = \bigcup_{(u,w)\in E_G} \{u, w\}$.*

We now prove that computing the rearrangement distance is NP-hard.

Theorem 26. *Given trees T_1 and T_2 and some integer k, it is NP-hard to decide if $d(T_1, T_2) \leq k$.*

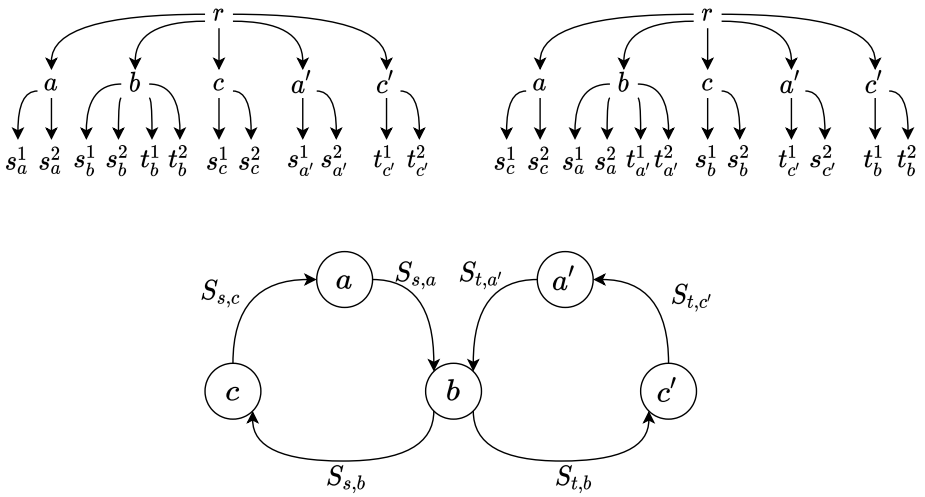


Figure 5.5: The trees T_1 and T_2 given instance H of 3-dimensional matching with $A = \{a, a'\}$, $B = \{b\}$ and $C = \{c, c'\}$ and $\mathcal{T} = \{s = \{a, b, c\},\ t = \{a', b, c'\}\}$ (top), and the corresponding movements graph for the trees T_1 and T_2 (bottom).

Proof. Reduction from 3-bounded 1-common 3-dimensional matching. We are given an instance H of 3-dimensional matching consisting of a set $\mathcal{T}$ of m triples (a, b, c) over the disjoint sets A, B, C. We construct two trees T_1 and T_2 each with $|A|+|B|+|C|+6m+2$ vertices, for which the rearrangement distance $d(T_1, T_2) \leq 3n + 6(m - n)$ if and only if H has a 3-dimensional matching of size n.

Consider such an instance H of 3-dimensional matching as above. In the construction, the trees T_1 and T_2 each have a root vertex r, and a vertex for every element of A, B and C — each of which have r as the parent. To each $v \in \{a, b, c\}$ in T_1 and triple t, we add a set $S_{t,v} = \{t_v^1, t_v^2\}$ of two (uniquely labelled) children. In T_2, we add the sets $S_{t,v}$ of two children to each of these three vertices, but cyclically shifted, with respect to T_1, i.e., we add $S_{t,a}$ to b, $S_{t,b}$ to c, $S_{t,c}$ back to a again. Note that this induces in the movements graph G a cycle $\mathcal{C}_t = \{(a, b), (b, c), (c, a)\}$ — see Figure 5.5. Now, observe that the movements graph G will have cycles of length 3 corresponding to each triple and two cycles may share one common vertex v if the triples share element v. A sequence of operations of total size $d(T_1, T_2)$ will consist of a permutation followed by a sequence of link-and-cut operations. Observe that any permutation involves for each cycle $\mathcal{C}_t$ an edge, two edges or all three edges. Now, the rearrangement distance aims to solve cycles in the movements graph in the sense that after the operations, the movements graph has no edges. Observe that given a cycle $\mathcal{C}_t$ of the movements graph G, then the minimum cost rearrangement to solve $\mathcal{C}_t$ consists of applying a permutation of size 3 involving the three vertices of the cycle, thus of total cost 3. Observe that two cycles sharing a common vertex cannot both be solved by a permutation that is a cyclic shift of the vertices of the cycle, that is they cannot be both solved with cost 3.

Moreover, permutations of vertices cannot solve more than one vertex of a cycle $\mathcal{C}_t$ if it is not a cyclic shift of the vertices of $\mathcal{C}_t$, as cycles do not share edges. In case of a cyclic shift of two vertices of $\mathcal{C}_t$, (1) a permutation of size 2 and then four link-and-cut operations are required. If instead (2) at most a single vertex of $\mathcal{C}_t$ is involved in a permutation, then six link-and-cut operations are required. We now detail how this implies that $d(T_1, T_2) \leq 3n + 6(m - n)$ if and only if H has a 3-dimensional matching of size n.

($\Rightarrow$) Assume that $d(T_1, T_2) \leq 3n + 6(m - n)$. By the above observation on how cycles of the movements graph G are solved by the sequence of permutations, cases (1) and (2) for solving cycles have the same cost equal to 6. Thus the only possible way to have a rearrangement distance less than or equal to $3n+6(m-n)$ is by taking n disjoint cycles solved by the permutation operation of cost 3. This implies a 3-dimensional matching of size n.

($\Leftarrow$) Now, suppose that H has a 3-dimensional matching $\mathcal{M} \subseteq \mathcal{T}$ of size n. This implies that there are n triples that are disjoint and thus the movements graph G has n disjoint cycles. By solving each cycle with a permutation of size 3, $m - n$ cycles are left in the movements graph. The remaining cycles, in the worst case, share common vertices with the cycles solved by the permutations, and thus they can be solved with a cost that is 6 in the worst case. Thus we obtain that $d(T_1, T_2) \leq 3n + 6(m - n)$, completing the proof. □

5.4.1 A 4-Approximation Algorithm for Binary Trees

We start by giving a lemma which states that when we apply a permutation to the labels of T_1 obtaining T_1', the size of the resulting family partition $\mathcal{P}'$ cannot increase or decrease too much with respect to the size of $\mathcal{P}$.

Lemma 33. *Given trees T_1 and T_2 with corresponding active set $\mathcal{X}$ and family partition $\mathcal{P}$, if T_1' is the tree (isomorphic to T_1) resulting from the application of permutation π of the labels of T_1, and $\mathcal{X}'$ and $\mathcal{P}'$ are the active set and the family partition of T_1' and T_2, respectively, then $|\mathcal{P}| - 2|\pi| \leq |\mathcal{P}'| \leq |\mathcal{P}| + 2|\pi|$.*

Proof. Let a be some label of T_1 which has been perturbed by permutation π, i.e., $\pi(a) = b \neq a$. The new family partition $\mathcal{P}'$ is obtained from $\mathcal{P}$ by means of deletions, insertions and substitutions of subsets. The crucial observation is that such an operation will only affect the *neighborhood* of a, namely the (possibly empty) set of its children $c_{T_1}(a)$ and its parent $p_{T_1}(a)$ in T_1. Let us consider each child $v \in c_{T_1}(a)$ first. We have the following cases.

- ($v \in \mathcal{P}_{(a,b)} \subseteq \mathcal{X}$): since π makes b the parent of v, which is exactly the parent of v in T_1, $v \notin \mathcal{X}'$ and $\mathcal{P}_{(a,b)}$ will be missing from $\mathcal{P}'$;
- ($v \in \mathcal{P}_{(a,c)} \subseteq \mathcal{X}$, $c \neq b$): after applying π, v will belong to set $\mathcal{P}'_{(b,c)}$, thus $\mathcal{P}_{(a,c)}$ will be replaced by $\mathcal{P}'_{(b,c)}$ in $\mathcal{P}'$;
- ($v \notin \mathcal{X}$): then $v \in \mathcal{P}'_{(b,a)}$ in $\mathcal{P}'$, thus $\mathcal{P}'$ might have an extra element with respect to $\mathcal{P}$.

Consider now the possible effects of π on $p_{T_1}(a)$. There are two possible scenarios:

- ($b \in \mathcal{P}_{(p_{T_1}(b),p_{T_2}(b))} \subseteq \mathcal{X}$): if $p_{T_2}(b) = p_{T_1}(a)$, then $b \notin \mathcal{X}'$ and if b was the only element of $\mathcal{P}_{(p_{T_1}(b),p_{T_2}(b))}$, the latter will be missing from $\mathcal{P}'$; else, $b \in \mathcal{P}'_{(p_{T_1}(a),p_{T_2}(b))}$, thus $\mathcal{P}_{(p_1(b),p_2(b))}$ will be replaced by $\mathcal{P}'_{(p_{T_1}(a),p_{T_2}(b))}$ in $\mathcal{P}'$;
- ($b \notin \mathcal{X}$): then $b \in \mathcal{P}'_{(p_{T_1}(a),p_{T_2}(b))}$ in $\mathcal{P}'$, thus $\mathcal{P}'$ might have an extra element with respect to $\mathcal{P}$.

In summary, $\mathcal{P}'$ is obtained from $\mathcal{P}$ with up to two deletions and two additions of sets in the family partition for each label involved in the permutation π, thus the result follows. □

In the special case where one of the trees, e.g., T_1, is *binary*, that is, each node has up to two children, we have the following lemma connecting link-and-cut and rearrangement distance.

Lemma 34. *Given T_1 a binary tree, T_2 any tree, we have that $d_\ell(T_1, T_2) \leq 4 \cdot d(T_1, T_2)$.*

Proof. Suppose that T_2 is optimally obtained from T_1 by applying a permutation π of the labels followed by a number of link-and-cut operations. Let T_1' be the tree resulting from the application of permutation π of the labels of T_1, $\mathcal{X}'$ and $\mathcal{P}'$ the active set and family partition of T_1' and T_2, respectively. By the construction of the family partition, the optimal number of link-and-cut operations to obtain T_2 from T_1' is at least $|\mathcal{P}'|$; we thus have that $d(T_1, T_2) = |\pi| + |\mathcal{X}'| \geq |\pi| + |\mathcal{P}'|$. Moreover, Lemma 33 says that $|\pi| \geq \frac{|\mathcal{P}|-|\mathcal{P}'|}{2}$, thus $d(T_1, T_2) \geq \frac{|\mathcal{P}|-|\mathcal{P}'|}{2} + |\mathcal{P}'| = \frac{|\mathcal{P}|}{2} + \frac{|\mathcal{P}'|}{2} \geq \frac{|\mathcal{P}|}{2}$. Now, since T_1 is binary, each set in the family partition $\mathcal{P}$ consists of up to two elements (the elements of $\mathcal{P}_{(x,y)}$ are the ones among the children of x in T_1 that becomes the children of y in T_2, thus they cannot be more than the number of children of x). It follows that $|\mathcal{X}| = d_\ell(T_1, T_2) \leq 2 \cdot |\mathcal{P}|$, hence $d(T_1, T_2) \geq \frac{d_\ell(T_1,T_2)}{4}$. □

Importantly, we note that Lemma 34 states that the link-and-cut distance algorithm provides a 4-approximation for the rearrangement distance when at least one of the trees involved is binary. We now show that we can compute the link-and-cut distance between two trees in linear time by showing that the family partition can be built in linear time.

Lemma 35. *The link-and-cut distance $d_\ell(T_1, T_2)$ between trees T_1 and T_2 each labelled by $[n]$ can be computed in time $O(n)$.*

Proof. Since the link-and-cut distance is $|\mathcal{X}|$, it suffices to demonstrate that the family partition $\mathcal{P}$ can be built in time $O(n)$. The procedure is as follows: we first do a DFT of tree T_1, building an array $p_{T_1}(v)$ of the parents in T_1, indexed by the child v. We build the same array $p_{T_2}(v)$ for tree T_2. Then we go through the set $[n]$ of labels, in some order: at each label v, should $p_{T_1}(v) = u \neq p_{T_2}(v) = w$, we add v to $\mathcal{P}_{(u,w)}$ of the family partition $\mathcal{P}$. Then we just sum up the sizes of the non-empty subsets $\mathcal{P}_{(u,w)}$ of $\mathcal{P}$ in order to obtain $|\mathcal{X}| = d_\ell(T_1, T_2)$. Clearly each tree traversal can be done in time $O(n)$, as both T_1 and T_2 have n vertices. In going through the labels, for each label v, we either add or do not add the single vertex v to $\mathcal{P}$, and so this procedure takes time $O(n)$. □

We thus have the following corollary from Lemma 34 and Lemma 35.

Corollary 27. *There exists a linear time 4-approximation algorithm for the rearrangement distance problem for binary trees.*

5.4.2 A General Constant-Factor Approximation Algorithm

In what follows we do not make any assumptions on the degrees of T_1 and T_2. We will actually consider the rearrangement distance generalized to forests, $\tilde{d}(F_1, F_2)$ (see the definition at Section 5.2), and show how to approximate it within a constant factor. This allows us to approximate $d(T_1, T_2)$ within a constant factor using the following procedure. First, we add n leaves $n+1, n+2, \ldots, n$ attached to the (identical) roots of T_1 and T_2 to obtain T_1' and T_2', respectively. We call the resulting trees *anchored*. Because T_1 and T_2 are assumed to have the same root that cannot be permuted, we have $d(T_1, T_2) = d(T_1', T_2')$. We claim that $\tilde{d}(T_1', T_2') = d(T_1', T_2')$.

Lemma 36. *For any two anchored trees T_1 and T_2, $\tilde{d}(T_1, T_2) = d(T_1, T_2)$.*

Proof. Consider a sequence s of link-and-cut and permutation operations that transforms T_1 into T_2. We convert it into a sequence s' of cut and permutation operations by simply replacing every link-and-cut operation $v \,|\, u \to w$ with a cut operation $(v \dagger u)$. Let T_1' be the forest obtained after applying s' on T_1. We claim that $T_1' \sim T_2$. Consider any $v \in [n]$. Because we can assume that the permutation operation precedes all the link-and-cut operations in s, if $p_{T_1'}(v) \neq \perp$ then we must have $p_{T_2}(v) = \perp$ or $p_{T_1'}(v) = p_{T_2}(v)$, as $p_{T_1'}(v)$ is the same as the parent of v after applying s on T_1. This shows that indeed $T_1' \sim T_2$, and so $\tilde{d}(T_1, T_2) \leq d(T_1, T_2)$.

For the other direction, we use the assumption that T_1 and T_2 are anchored trees on $2n$ nodes: in both trees r is the root and there are n leaves $n+1, n+2, \ldots, 2n$ attached to r. Observe that $\tilde{d}(T_1, T_2) < n$. We claim that an optimal sequence of cut and permutation operations doesn't permute r. Assume otherwise, then for every $u = n+1, n+2, \ldots, n$ either u is also permuted, or we have a cut operation $(u \dagger r)$, so the size of the sequence must be at least n. Now, let s be an optimal sequence consisting of a permutation π and then some cut operations, and let T_1'' be the tree obtained after applying s on T_1. We obtain a sequence s' of link-and-cut and permutation operations from s as follows. For every $v \in [n]$, if $p_{T_1''}(v) = \perp$ and $p_{T_2}(v) \neq \perp$, we locate the cut operation $(v \dagger u)$ in s (there must be such operation, as T_1 and T_2 have the same root that is not permuted). In s', we replace this operation with $v \,|\, u \to w$, where $w = p_{T_2}(v)$. Additionally, we reorder all link-and-cut operations to ensure that w is not a descendant of v, which can be guaranteed by considering v in the decreasing order of their levels in T_2. Let T_1' be the result of applying s' on T_1, and consider any $v \in [n]$. If $p_{T_1''}(v) \neq \perp$ and $p_{T_2}(v) \neq \perp$ then $p_{T_1'}(v) = p_{T_2}(v)$ because $T_1'' \sim F_2$, and if $p_{T_1''}(v) = \perp$ and $p_{T_2}(v) \neq \perp$ then $p_{T_1'}(v) = p_{T_2}(v)$ by the choice of w. This shows that s' transforms T_1 into T_2, thus $d(T_1, T_2) \leq \tilde{d}(T_1, T_2)$. □

We can thus approximate $\tilde{d}(T_1', T_2')$ within a constant factor to obtain a constant factor approximation of $d(T_1, T_2)$. In the remaining part of this section we design an approximation algorithm for $\tilde{d}(F_1, F_2)$, where F_1 and F_2 are two arbitrary forests.

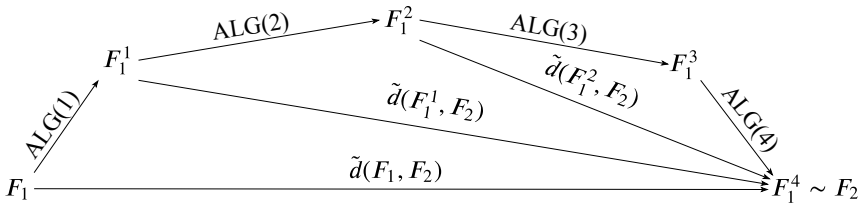


Figure 5.6: The four steps of the approximation algorithm.

We start with describing the notation. Consider two forests F_1 and F_2. For every $i \in [n]$, let $a[i] \in [n]$ be the parent of a non-root node i in F_1, and $a[i] = 0$ if i is a root in F_1. Formally, $a[i] = p_{F_1}(i)$ when $p_{F_1}(i) \neq \perp$ and $a[i] = 0$ otherwise; $b[i]$ is defined similarly but for F_2. We think of a and b as vectors of length n.

The algorithm consists of four steps, with step j transforming forest F_1^{j-1} into F_1^j by performing $\mathrm{ALG}(j)$ operations, starting from $F_1^0 = F_1$. We will guarantee that $\mathrm{ALG}(j) = \mathcal{O}(\tilde{d}(F_1^{j-1}, F_2))$. Then, by triangle inequality and symmetry, $\tilde{d}(F_1^j, F_2) \leq \tilde{d}(F_1^{j-1}, F_1^j) + \tilde{d}(F_1^{j-1}, F_2) \leq \mathrm{ALG}(j) + \tilde{d}(F_1^{j-1}, F_2) = \mathcal{O}(\tilde{d}(F_1^{j-1}, F_2))$, so by induction $\tilde{d}(F_1^j, F_2) = \mathcal{O}(\tilde{d}(F_1, F_2))$. Consequently, $\mathrm{ALG}(j) = \mathcal{O}(\tilde{d}(F_1, F_2))$, making the overall cost $\sum_j \mathrm{ALG}(j) = \mathcal{O}(\tilde{d}(F_1, F_2))$. In the j-th step of the algorithm $a[i]$ refers to the parent of i in F_1^{j-1}. To analyse each step of the algorithm we will use the following two structures, the first of which is a streamlined version of family partitions defined in 5.2.

Definition 18 (family partition of two forests)**.** *Given two forests F_1 and F_2, their* family partition $P(F_1, F_2)$ *is the set* $\{(a[i], b[i]) : a[i], b[i] \neq 0 \ \wedge \ a[i] \neq b[i]\}$.

Definition 19 (migrations graph)**.** *Given two forests F_1 and F_2, the* migrations graph $MG(F_1, F_2)$ *consists of edges* $\{(i, j) : a[i], a[j], b[i], b[j] \neq 0 \ \wedge \ a[i] = a[j] \ \wedge \ b[i] \neq b[j]\}$.

For a multiset S, let $|S|$ denote its cardinality, that is, the sum of multiplicities of all distinct elements of S. The mode of S, denoted $\mathsf{mode}(S)$, is any element $s \in S$ with the largest multiplicity $\mathsf{freq}_S(s)$. We will use the following combinatorial lemma.

Lemma 37. *Given any multiset S, let $f = \min\{|S| - \mathsf{freq}_S(\mathsf{mode}(S)), \lfloor |S|/2 \rfloor\}$. All $|S|$ elements of S can be partitioned into f pairs $(x_1, y_1), \ldots, (x_f, y_f)$, $x_i \neq y_i$, for every $i \in [f]$, and the remaining $|S| - 2f$ elements.*

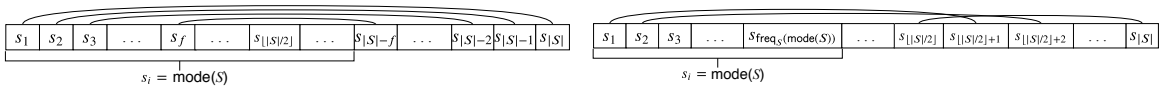


Figure 5.7: Pairing in the case $f = |S| - \mathsf{freq}_S(\mathsf{mode}(S))$ (left) and $f = \lfloor |S|/2 \rfloor$ (right).

Proof. Number the elements of S so that $s_1 = \ldots = s_{\mathsf{freq}_S(\mathsf{mode}(S))} = \mathsf{mode}(S)$ and all of the others are sorted and numbered from $\mathsf{freq}_S(\mathsf{mode}(S)) + 1$ to $|S|$ accordingly. Then, if $f = |S| - \mathsf{freq}_S(\mathsf{mode}(S))$, pairs $(s_i, s_{|S|-i+1})$, $i \in [f]$ are s.t. $s_i \neq s_{|S|-i+1}$ (Figure 5.7, left); if $f = \lfloor |S|/2 \rfloor$, pairs $(s_i, s_{\lfloor |S|/2 \rfloor + i})$, $i \in [\lfloor |S|/2 \rfloor]$ are s.t. $s_i \neq s_{\lfloor |S|/2 \rfloor + i}$ (Figure 5.7, right). □

5.4.3 Step 1

Roughly speaking, the aim of the first step is to ensure that all nodes that might be possibly involved in a permutation, i.e., the nodes with different children in F_1 and F_2, are roots. This is so that we do not need to worry about the relationship with their parents. For every $i \in [n]$ such that $a[i]$ and $b[i]$ are both defined and different, we cut the edges from $a[i]$ and $b[i]$ to their parents in F_1, thus making both of them roots. In other words, for every i such that $a[i], b[i] \neq 0$ and $a[i] \neq b[i]$, we cut edges $(a[i], a[a[i]])$ and $(b[i], a[b[i]])$. The resulting forest F_1^1 has the following property: for each $i \in [n]$ such that the parents of i in F_1^1 and in F_2 are both defined and different, $a[a[i]] = a[b[i]] = \perp$.

The number of cuts in this step is by definition at most twice the size of the family partition $P(F_1, F_2)$. By Lemma 33 it follows immediately that $|P(T_1, T_2)| \leq 2d(T_1, T_2)$ for two trees T_1 and T_2. We show that this still holds for forests and $\tilde{d}$: for completeness, we provide a self-contained proof.

Lemma 38. $|P(F_1, F_2)| \leq 2\tilde{d}(F_1, F_2)$, *implying* $\mathrm{ALG}(1) \leq 4\tilde{d}(F_1, F_2)$.

Proof. It is enough to verify that applying a single cut operation might decrease the size of the family partition by at most one, while applying a permutation operation π might decrease the size of the family partition by at most $2s$, where $s = |\{u : u \neq \pi(u)\}|$.

Consider a cut operation $(v \dagger u)$. The only change to a is that $a[v]$ becomes 0, so indeed the size of the family partition might decrease by at most one.

Now consider a permutation π. After applying π, an edge $(i, a[i])$ becomes $(\pi(i), \pi(a[i]))$, making $\pi(a[\pi^{-1}(i)])$ the parent of i. This transforms the family partition P into

$$P' = \{(\pi(a[i]), b[\pi(i)]) : a[i] \neq 0 \ \wedge \ b[\pi(i)] \neq 0 \ \wedge \ \pi(a[i]) \neq b[\pi(i)]\}.$$

To lower bound the size of $|P'|$, we first focus on the subset of P corresponding to the nodes that are fixed by π. We therefore define

$$P_f = \{(a[i], b[i]) : a[i] \neq 0 \ \wedge \ b[i] \neq 0 \ \wedge \ a[i] \neq b[i] \ \wedge \ \pi(i) = i\}.$$

By definition, we can equivalently rewrite P_f as

$$P_f = \{(a[i], b[\pi(i)]) : a[i] \neq 0 \ \wedge \ b[\pi(i)] \neq 0 \ \wedge \ a[i] \neq b[\pi(i)] \ \wedge \ \pi(i) = i\}.$$

Now consider all pairs with the same second coordinate y in P_f: $(x_1, y), (x_2, y), \ldots, (x_k, y)$, where $x_i \neq y$ for every $i \in [k]$. P' contains all pairs $(\pi(x_i), y)$ such that $\pi(x_i) \neq y$. If $\pi(y) = y$ then $\pi(x_i) = y$ cannot happen and P' contains all pairs with the second coordinate y from P_f; otherwise, P' contains all such pairs except possibly one. Overall, $|P'| \geq |P_f| - s$, and $|P_f| \geq |P| - s$ so indeed $|P'| \geq |P| - 2s$. □

Example 19. *Consider F_1 and F_2 depicted in Figure 5.8. Step 1 consists of cut operations $(2 \dagger 1)$ (because, e.g., $a[4] \neq b[4]$ and $a[4] = 2$), $(3 \dagger 1)$ (because $b[4] = 3$) and $(7 \dagger 2)$ (because, e.g., $a[11] \neq b[11]$ and $a[11] = 7$). The resulting forest F_1^1 is shown in Figure 5.9a.*

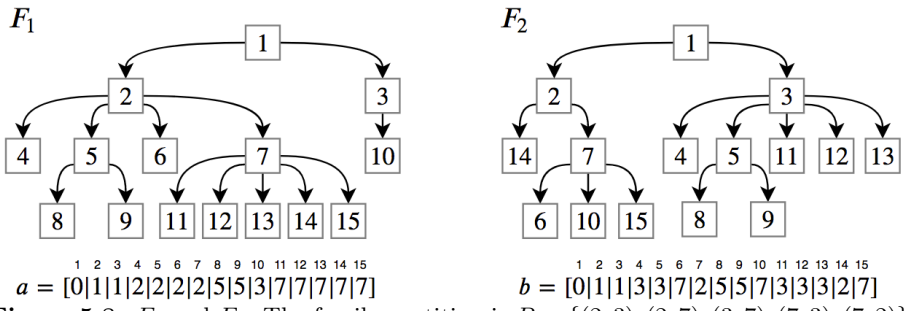


Figure 5.8: F_1 and F_2. The family partition is $P = \{(2,3),(2,7),(3,7),(7,3),(7,2)\}$.

5.4.4 Step 2

Consider $u \in [n]$, and let $\mathsf{children}_{F_1^1}(u) = \{v_1, \ldots, v_k\}$. We define the multiset $B(u) = \{b[v_i] : b[v_i] \neq 0\}$ containing the parents in F_2 of the children of u in F_1^1. Recall that $\mathsf{mode}(B(u))$ is the most frequent element of $B(u)$ (ties are broken arbitrarily). We cut all edges (v_i, u) such that $b[v_i] \neq 0$ and $b[v_i] \neq \mathsf{mode}(B(u))$, and define, for each $u \in [n]$, its representative $\mathsf{rep}(u) = \mathsf{mode}(B(u))$. Intuitively, $\mathsf{rep}(u)$ is the node that might be convenient to replace u with using a permutation. Roughly speaking, in this step we get rid of all of the children of u that would be misplaced after permuting u and $\mathsf{rep}(u)$, for each $u \in [n]$. The resulting forest F_1^2 has the following property: for each $u \in [n]$, for any child v of u in F_1^2, either $b[v] = 0$ or $b[v] = \mathsf{rep}(u)$, i.e., the children of each node u of F_1^2 have all the same parent $\mathsf{rep}(u)$ in F_2.

To bound the number of cuts in this step we first need a technical lemma relating the rearrangement distance of two forests and the size of any matching in their migrations graph.

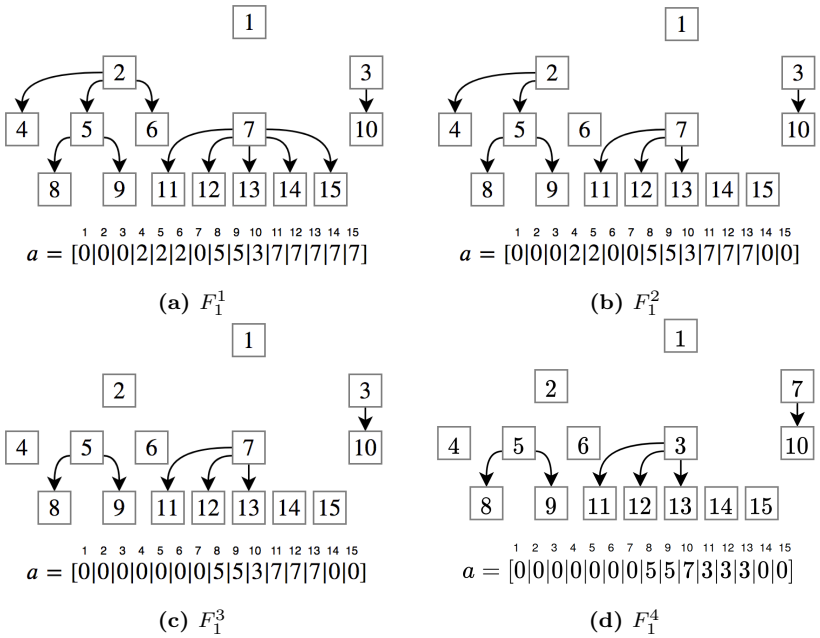
Lemma 39. *Consider two forests F_1 and F_2 and their migrations graph $MG(F_1, F_2)$. For any matching M in $MG(F_1, F_2)$ it holds that $|M| \leq \tilde{d}(F_1, F_2)$.*

Proof. By definition, there is an edge between i and j in $MG(F_1, F_2)$ if and only if $a[i] = a[j]$, but $b[i] \neq b[j]$. Let M be any matching in $MG(F_1, F_2)$. If $|M| > 0$ then $\tilde{d}(F_1, F_2) \geq 1$, so it is enough to show that, for a single operation transforming F_1 into F_1', the graph $MG(F_1', F_2)$ contains a matching M' of size at least $|M| - s$, where $s = 1$ for a cut operation and $s = |\{u : u \neq \pi(u)\}|$ for a permutation operation π.

First, consider a cut operation $(v \dagger u)$. The only change in $MG(F_1', F_2)$ is removing all edges incident to v. M contains at most one edge incident to v, so we construct M' of size at least $|M| - 1$ from M by possibly removing a single edge. Second, consider a permutation operation π: we construct M' from M by removing every edge (v, w) such that $v \neq \pi(v)$ or $w \neq \pi(w)$. Because there is at most one edge incident to every u such that $u \neq \pi(u)$, M' contains at least $|M| - s$ edges. M' is a matching in $MG(F_1', F_2)$, as for every $(v, w) \in M'$ we have $p_{F_1'}(v) = p_{F_1}(v)$ and $p_{F_1'}(w) = p_{F_1}(w)$. □

Lemma 40. $\mathrm{ALG}(2) \leq 2\tilde{d}(F_1^1, F_2)$.

Proof. We consider each $u \in [n]$ separately. Let $m = \mathsf{freq}_{B_u}(\mathsf{mode}(B_u))$ and MG_u be the subgraph of $MG(F_1^1, F_2)$ induced by B_u. We will first construct a matching



(a) F_1^1 **(b)** F_1^2 **(c)** F_1^3 **(d)** F_1^4

Figure 5.9: The forests obtained after Step 1 (5.9a), Step 2 (5.9b), Step 3 (5.9c) and Step 4 (5.9d).

of appropriate size in every MG_u. We cut every (v_i, u) such that $b[v_i] \neq 0$ and $b[v_i] \neq \mathsf{mode}(B_u)$, making $|B_u| - m$ cuts. Let $f = \min(|B_u| - m, \lfloor |B_u|/2 \rfloor)$. By Lemma 37, we can partition a subset of B_u into f pairs $(b[v_i], b[v_j])$ such that $b[v_i] \neq b[v_j]$. We add every edge (v_i, v_j) to the constructed matching. We claim that $|B_u| - m \leq 2f$. This holds because $|B_u| - m \leq 2(|B_u| - m)$ and $|B_u| - m \leq |B_u| - 1 \leq 2\lfloor |B_u/2| \rfloor$ for nonempty B_u.

We take the union of all such matchings to obtain a single matching M. As argued above, the total number of cuts is at most $2|M|$. Together with Lemma 39, this implies that $\mathrm{ALG}(2) \leq 2|M| \leq 2\tilde{d}(F_1^1, F_2)$. □

Example 20. *Consider again F_1 and F_2 of Figure 5.8. $B(7) = \{3, 3, 3, 2, 7\}$, thus we cut $(14 \dagger 7)$ and $(15 \dagger 7)$. $B(2) = \{3, 3, 7\}$, implying $(6 \dagger 2)$. The resulting F_1^2 is shown in Figure 5.9b.*

5.4.5 Step 3

If after Step 2 all of the children of a node u of F_1 have the same parent $\mathsf{rep}(u)$ in F_2, it still may be the case where $\mathsf{rep}(u) = \mathsf{rep}(v)$ with $u \neq v$, i.e., all of the children of two distinct nodes of F_1 have the same parent in F_2. In this case, it is not clear how to choose whether to replace u or v with $\mathsf{rep}(u) = \mathsf{rep}(v)$ in a permutation. This step aims at resolving this situation by cutting the ambiguous edges.

Consider thus $u \in [n]$, and let $\mathsf{children}_{F_2}(u) = \{v_1, v_2, \ldots, v_k\}$. We define the multiset $B'(u) = \{a[v_i] : a[v_i] \neq 0\}$ containing the parents in F_1^2 of the children of u in

F_2. We cut all edges $(v_i, a[v_i])$ such that $a[v_i] \neq 0$ and $a[v_i] \neq \mathsf{mode}(B'(u))$, breaking ties arbitrarily, and define $\mathsf{rep}'(u) = \mathsf{mode}(B'(u))$. The resulting forest F_1^3 has the following property: for each $u \in [n]$, for any child v of u in F_2, we have $a[v]) = \perp$ or $a[v] = \mathsf{rep}'(u)$.

We observe that the number of cuts performed by the above procedure is the same as if we had applied Step 2 on F_2 and F_1^2. Therefore, Lemma 40 implies the following.

Lemma 41. $\mathrm{ALG}(3) \leq 2\tilde{d}(F_1^2, F_2)$.

Example 21. *Consider again F_1 and F_2 of Figure 5.8. We have $B'(3) = \{2, 2, 7, 7, 7\}$, we thus cut $(4 \dagger 2)$ and $(5 \dagger 2)$. The resulting forest F_1^3 is shown in Figure 5.9c.*

Step 4 We summarize the properties of F_1^3 and F_2:

1. For each $u \in [n]$ such that $a[u], b[u] \neq 0$ and $a[u] \neq b[u]$, $a[u]$ and $b[u]$ are roots in F_1^3.
2. For each $u \in [n]$ we can define $\mathsf{rep}(u) \in [n]$ in such a way that, for any child v of u in F_1^3, we have $b[v] = 0$ or $b[v] = \mathsf{rep}(u)$.
3. For each $u \in [n]$ we can define $\mathsf{rep}'(u) \in [n]$ in such a way that, for any child v of u in F_2, we have $a[v] = 0$ or $a[v] = \mathsf{rep}'(u)$.

To finish the description of the algorithm, we show how to find a permutation operation π of size $\mathcal{O}(\tilde{d}(F_1^3, F_2))$ that transforms F_1^3 into F_1^4 such that $F_1^4 \sim F_2$.

For every u such that $a[u], b[u] \neq 0$ and $a[u] \neq b[u]$, we require that $\pi(a[u]) = b[u]$. Due to Property 1, for every such u we have ensured that $a[u]$ and $b[u]$ are roots of F_1^3. So, if we can find a permutation π that satisfies all the requirements and does not perturb the non-roots of F_1^3, then it will transform F_1^3 into F_1^4 such that $F_1^4 \sim F_2$. Furthermore, if for every x perturbed by π there exists u such that $a[u], b[u] \neq 0$ and $a[u] \neq b[u]$ with $x = a[u]$ or $x = b[u]$ then by Lemma 38 $|\pi| \leq 2|P(F_1^3, F_2)| \leq 4\tilde{d}(F_1^3, F_2)$ as required.

To see that there indeed exists such π, observe that due to Property 2 there cannot be two requirements $\pi(x) = y$ and $\pi(x) = y'$ with $y \neq y'$. Similarly, due to Property 3 there cannot be two requirements $\pi(x) = y$ and $\pi(x') = y$ with $x \neq x'$. Thinking of the requirements as a graph, the in- and out-degree of every node is hence at most 1, so we can add extra edges to obtain a collection of cycles defining a permutation π that does not perturb the nodes not participating in any requirement.

Example 22. *Consider F_1 and F_2 of Figure 5.8. $\pi = (3\ 7)$ transforms F_1^3 into $F_1^4 \sim F_2$. The final F_1^4 is shown in Figure 5.9d.*

5.5 Fixed parameter tractability

This section is devoted to showing that computing the rearrangement distance between trees T_1 and T_2 is fixed-parameter tractable, essentially via the bounded search tree technique [152]. In this case, the instance also contains a parameter k: in time

$O((4k)^{2k^2}n)$ we (1) determine if $d(T_1, T_2) \leq k$ and, if this is the case, (2) find the minimum sequence of operations transforming T_1 into T_2.

The main idea of our algorithm is that, since a permutation operation is essentially just renaming the nodes, we can reorder the sequence of operations that transforms T_1 into T_2 so that all permutations precede the link-and-cut operations. Let T^* be the tree obtained from T_1 using only permutations and such that we can optimally obtain T_2 from T^* using only link-and-cut operations. Then $d(T_1, T_2) = d_\pi(T_1, T^*) + d_\ell(T^*, T_2)$. Our algorithm consists of showing that $d_\pi(T_1, T^*)$ is related to the size of the family partition, and that we can compute $d_\ell(T^*, T_2)$ in linear time. Finding such a tree T^* is easier when we want to determine if the rearrangement distance $d(T_1, T_2)$ is at most k.

In fact, a consequence of Lemma 33 is that $d(T_1, T_2) \geq d_\pi(T_1, T^*) \geq |\mathcal{P}|/2$, where $\mathcal{P}$ is the family partition associated with T_1 and T_2. Notice that any sequence of operations that transforms T_1 into T_2 also transforms $\mathcal{X}$ into the empty set, and thus $\mathcal{P}$ into the empty partition.

Since $d(T_1, T_2) \geq |\mathcal{P}|/2$, the first step of our algorithm is to compute the family partition $\mathcal{P}$ of T_1 and T_2 and verify that $k \geq |\mathcal{P}|/2$. If that inequality is not satisfied, then, since as stated above $d_\pi(T_1, T^*) \geq |\mathcal{P}|/2$, it would follow that $d(T_1, T_2) > k$. Hence we can focus on the instances where $k \geq |\mathcal{P}|/2$, that is $|\mathcal{P}| \leq 2k$. Since the family partition is sufficiently small, we can compute all sequences of permutations of at most k labels of $\mathcal{X}$ in time $O((4k)^{2k^2})$. In fact, each of the permutations involves one of the 2^{2k} subsets of vertices of $\mathcal{X}$, and there can be at most $(2k)!$ permutations of a set of $2k$ elements. Overall there are at most $\left(2^{2k}(2k)!\right)^k$ such sequences: it is trivial to organize them in a search tree that can be generated and traversed in linear time, and some crude upper bound results in the desired time bound. Let $\mathcal{T}$ be the set of trees that are obtained by applying to T_1 the sequence of operations corresponding to a node of the search tree.

The second part of our algorithm is to compute $d_\ell(T, T_2)$ for each tree in $T \in \mathcal{T}$, which, by Lemma 35, requires $O(n)$ time for each tree, keeping track of the tree T^* minimizing $d_\pi(T_1, T^*) + d_\ell(T^*, T_2)$. The algorithm has therefore $O((4k)^{2k^2}n)$ time complexity.

Chapter 6

MP3: Triplet-Based Similarity Score for Tumor Phylogenies

Key Points

Problem. The latest advances in cancer sequencing, and the availability of a wide range of methods to infer the evolutionary history of tumors, have made it important to evaluate, reconcile and cluster different tumor phylogenies. None of the distance measures for tumor phylogenies that have been proposed in the literature so far is able to manage mutations occurring multiple times in the tree, a circumstance often occurring in real cases.

Model. We design a similarity measure for tumor phylogenies, modelled as fully-labelled trees that can have multiple labels at each node, and such that each label can be assigned to multiple nodes. Our measure, that generalizes the notion of rooted triples similarity for classical phylogenies to tumor phylogenies, is able to effectively manage cases where multiple mutations can occur at the same time and mutations can occur multiple times.

Included Works

This chapter is devoted to the paper **Triplet-based similarity score for fully multi-labeled trees with poly-occurring labels** [100], presented at the conference *RECOMB-CCB 2020* and published in the journal *Bioinformatics*.

6.1 Introduction

Recent methods to accurately infer the clonal evolution and progression of cancer have made it possible to develop targeted therapies for treating the disease. As discussed in several studies [276, 355], understanding the history of accumulation and the prevalence

of somatic mutations during cancer progression is a fundamental step to devise these treatment strategies.

Given the importance of the task, a multitude of methods for cancer phylogeny reconstruction have been developed over the years. The increasing number of tools created has been encouraged by the diversity of data available; for instance, we are witnessing a shift from bulk sequencing data [188, 189, 70, 71, 373] towards single-cell data [212, 101, 374, 135] and hybrid approaches [261, 262].

Having many different tools accomplishing the same task requires solid methods to compare their results. In contrast with classical phylogenetic trees, whose leaves, and only leaves, are labeled (with the species they represent), the trees that model tumor phylogenies are *fully-labeled*, i.e., they also have labels (corresponding to the mutations) on the internal nodes. While there is a wide range of measures to compare leaf-labeled trees in the literature, ad-hoc methods for tumor phylogenies are starting to appear in the last few years [126, 225, 174]; in particular, a detailed study of some notions of distance [126] has introduced two new measures complementing some more established definitions used in various cancer inference studies [102, 101]. Those new measures are more nuanced, in order to capture some aspects of the mutation inheritance process, while still being very efficient to compute. A common trait of all the latter distances is their reliance on the analysis of *pairs* of nodes.

On the other hand, some of the most widely used distances on classical phylogenies are based on rooted triples [81, 127, 22] (for rooted phylogenies) or quartets [131] (for unrooted phylogenies) of labeled leaves. Although such metrics have major limitations for our purposes, as they do not apply directly to fully-labeled trees, they also have some desirable properties that we would like to transfer in our setting. Specifically, this kind of metric captures well the differences in the topology of the trees; a feature that, to the best of our knowledge, lacks in most of the existing methods for tumor phylogenies. Therefore we expect a triplet-based measures to provide additional insights on the different evolutionary histories, when applied to cancer progression.

In this chapter, we generalize the notion of rooted triples similarity for classical phylogenies to tumor phylogenies. Moreover, we further extend this to multi-labeled trees (that is, where each node is labeled by a set of labels) and poly-occurring labels (that is, each label can be assigned to more than one node). The latter feature is needed since recent studies [239, 82] suggest widespread recurrence and loss of mutations, and more and more methods designed to infer tumor phylogenies considering such a possibility are starting to appear [135, 102, 101]. In a phylogenetic tree a mutation loss is represented by a special character in the label, such as a minus sign: the design of our measure allows to handle such evolutionary events effectively, as they uniquely correspond to their label like any other kind of mutation.

Through an extensive experimental analysis, we show that our novel measure is able to overcome the limitations in the existing literature and to provide a better alternative to both the direct comparison of evolutionary histories and the application to established clustering techniques, following the approach of [126]. Such a performing measure can also be incorporated in recent works [19, 174] designed to cluster and build consensus across multiple cancer progressions.

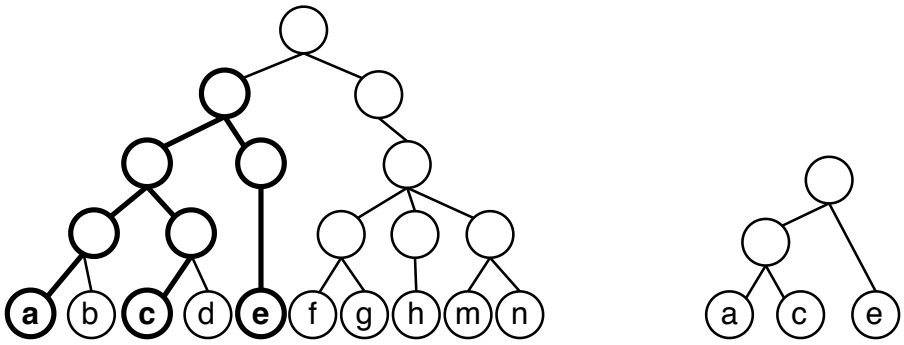


Figure 6.1: Rooted triplet on labels (a, c, e). (*Left*) Tree T where the smallest subtree that contains all three labels is highlighted. (*Right*) The minimal topology induced by (a, c, e).

6.2 Methods

A classical phylogenetic tree is a rooted, unordered, leaf-labeled tree. The set of all the labels occurring in T is denoted by $\lambda(T)$, and a function $\mathtt{N}(\cdot)$ maps each element of $\lambda(T)$ to a leaf of T. We denote with $\mathtt{LCA}(u, v)$ the Lowest Common Ancestor of nodes u and v. Given three leaves $u, v, z \in V_T$, the *minimal tree topology* they induce on T, denoted as $\mathtt{MTT}_T(u, v, z)$, is the smallest subtree of T that includes the nodes $V_T^{u,v,z} = \{u, v, z\} \cup \mathtt{LCA}(u, v) \cup \mathtt{LCA}(v, z) \cup \mathtt{LCA}(u, z)$, and where all the nodes with degree 2 not in $V_T^{u,v,z}$ are contracted.

The rooted triplet distance measures the dissimilarity between two leaf-labeled trees with identical labels. It is given by the number of rooted triplets that induce different minimal topologies (Figure 6.1) in the two trees over the total number of triplets [213]. As tumor progression trees are fully-labeled, such metric cannot be directly applied: in this section we propose a novel similarity measure, inspired by the triplet distance, specifically designed for these more general trees.

6.2.1 Extension to fully labeled trees and multi-labeled trees

A tree T on a set V_T of n nodes is *fully-labeled* by a set $\lambda(T)$ of labels if there is a bijection $\mathtt{N} : \lambda(T) \to V_T$. The definition of minimal topology of three leaves can be trivially extended to the minimal topology of three nodes: we next show that there are only five possible configurations (see Figure 6.2).

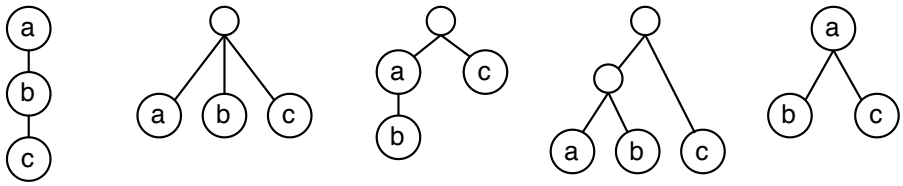


Figure 6.2: The five possible configurations for the minimal tree topology induced by three nodes.

Lemma 42. *Given nodes $u, v, z \in V_T$, there exist only five possible configurations for* $\texttt{MTT}_T(u, v, z)$.

Proof. We start by dividing two possible cases: (i) $\texttt{LCA}(u, v) = \texttt{LCA}(v, z) = \texttt{LCA}(u, z)$, or (ii) just two LCAs are the same, say $\texttt{LCA}(v, z) = \texttt{LCA}(u, z)$. There are no other possibilities, as $\texttt{LCA}(u, v) \neq \texttt{LCA}(v, z) \neq \texttt{LCA}(u, z)$ is impossible: indeed, suppose without loss of generality that $\texttt{LCA}(u, v)$ is a descendant of $\texttt{LCA}(u, z)$, $\texttt{LCA}(u, v) \neq \texttt{LCA}(u, z)$: they cannot be unrelated, as by definition they are both ancestors of u. $\texttt{LCA}(u, z)$ is thus a common ancestor for v and z. Suppose towards a contradiction that $\texttt{LCA}(v, z) \neq \texttt{LCA}(u, z)$, thus it is a descendant of $\texttt{LCA}(u, z)$ and an ancestor of $\texttt{LCA}(u, v)$. But then it is an ancestor of both u an z and it is lower than $\texttt{LCA}(u, z)$, a contradiction.

Case (i) has two subcases: either $\texttt{LCA}(u, v) \in \{u, v, z\}$, corresponding to the rightmost configuration in Figure 6.2, or $\texttt{LCA}(u, v) \notin \{u, v, z\}$, corresponding to the second configuration from the left. Case (ii) has three subcases: either both the distinct LCAs are in $\{u, v, z\}$, or none of the two is, or finally one is in $\{u, v, z\}$ and the other is not. The first subcase corresponds to the leftmost configuration in Figure 6.2, the second subcase to the fourth configuration from the left. For the third subcase, either the external LCA is an ancestor of all of the three $\{u, v, z\}$, corresponding to the third configuration, or it is an ancestor of two nodes and a descendant of the third one, say u. In the latter case, though, the external node would be the only child of u, and thus would be contracted by definition of $\texttt{MTT}_T(u, v, z)$, leading again to the rightmost configuration of Figure 6.2. □

In the case of fully-labeled trees, the definition of LCA of two nodes and MTT of three nodes can trivially be extended to the LCA of two labels and the MTT of three labels, as there is a one-to-one correspondence between nodes and labels. From now on, for ease of presentation, given two nodes u and v and their respective labels a and b, we will use $\texttt{LCA}(u, v)$ and $\texttt{LCA}(a, b)$ interchangeably. When modeling tumor progression, though, to have a bijection between nodes and labels (i.e., mutations) is quite a strong assumption, as multiple mutations often appear at the same time in the evolutionary history of cancer. We thus relax our assumptions and consider *multi-labeled* instead of fully-labeled trees.

A rooted, unordered tree T is multi-labeled if there exists a surjective function $\texttt{N} : \lambda(T) \to V_T$ that labels each node of T with a set of labels from $\lambda(T)$: note that, in this model, each label is assigned to one and only one node of T. We extend the definition of lowest common ancestor of two labels for a multi-labeled tree as follows: if $a \in \lambda(T)$ and $b \in \lambda(T)$ label the same node u, then $\texttt{LCA}(a, b) = u$; if they label two distinct nodes u, v, then $\texttt{LCA}(a, b) = \texttt{LCA}(u, v)$. This allows us to straightforwardly extend the definition of minimal tree topology of three labels for multi-labeled trees. There are only four possible additional configurations for the minimal tree topology of multi-labeled trees, shown in Figure 6.3.

Lemma 43. *Given T multi-labeled and $a, b, c \in \lambda(T)$, there exist nine configurations for* $\texttt{MTT}_T(a, b, c)$.

Proof. Besides the five possible configurations already listed in the proof of Lemma 1, the multi-labeled model admits four additional configurations, due to the extension of the definition of LCA of two labels. In the first three additional cases, two labels are assigned

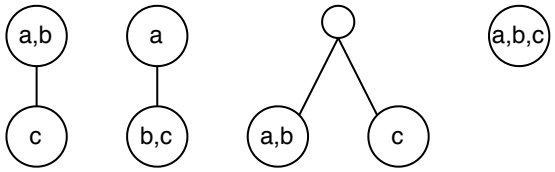


Figure 6.3: The four additional possible configurations for the minimal tree topology of multi-labeled trees induced by three nodes.

to the same node and the third one to a different one. Withouth loss of generality, suppose that a and b label the same node. These cases are: (i) $\texttt{LCA}(a,b) = \texttt{LCA}(b,c) = \texttt{LCA}(a,c)$ and the LCA is a node in $\{a,b,c\}$, (ii) $\texttt{LCA}(a,b) \neq \texttt{LCA}(b,c) = \texttt{LCA}(a,c)$ and both LCA are nodes in $\{a,b,c\}$, and (iii) $\texttt{LCA}(a,b) \neq \texttt{LCA}(b,c) = \texttt{LCA}(a,c)$ and only $\texttt{LCA}(a,b)$ is a node in $\{a,b,c\}$. The remaining case (iv) is the simplest one in which a, b, c label the same node of T, implying that $\texttt{LCA}(a,b) = \texttt{LCA}(b,c) = \texttt{LCA}(a,c)$. □

Figure 6.3 reports cases (i) to (iv) from left to right.

6.2.2 Extension to poly-occurring labels

We further extend our model of tumor phylogeny by allowing the same label of $\lambda(T)$ to be assigned to multiple nodes of T. An element of $\lambda(T)$ that labels more than one node of T is said to be a *poly-occurring* label. To the best of our knowledge, none of the existing tools is able to handle poly-occurring labels: indeed, although some of them accept input trees with poly-occurring labels, they simply disregard the multiple occurrences of a same label.

Since it is often the case where the inferred evolutionary history involves the appearance of the same mutation in multiple events, a meaningful comparison between tumor phylogenies cannot overlook such a phenomenon. To consider poly-occurring labels in our similarity measure, we extend the definition of minimal tree topology. First, note that if a label occur multiple times in the tree, then N maps each label to one or more nodes in V_T. Then, we define the minimal tree topology of poly-occurring labels a, b, c, denoted by M, as follows, where $\sqcup$ indicates the multiset union:

$$\texttt{M}_T(a,b,c) = \bigsqcup_{u \in \texttt{N}(a), v \in \texttt{N}(b), z \in \texttt{N}(c)} \texttt{MTT}_T(u,v,z)$$

In other words, the minimal tree topology of three labels is the multiset of all the minimal tree topologies of the nodes where a, b, and c appear. We remark that in this setting $\texttt{M}_T$ is a multiset of configurations, thus the same configuration may appear multiple times in $\texttt{M}_T$.

6.2.3 Similarity measure between trees

We are now able to define a similarity measure between fully-labeled trees with poly-occurring labels. Let S be a multiset and let $|S|$ be its cardinality. We define the number of shared configurations of labels a, b, c between two trees T_1 and

T_2 as $N(a,b,c) = |\mathtt{M}_{T_1}(a,b,c) \sqcap \mathtt{M}_{T_2}(a,b,c)|$, i.e., the cardinality of the multiset intersection, and the maximum number of configurations of the triplet in the trees as $D(a,b,c) = \max\{|\mathtt{M}_{T_1}(a,b,c)|, |\mathtt{M}_{T_2}(a,b,c)|\}$.

Based on these two values we define multiple variations of the Multi Poly-occurring labels triplet-based (MP3) similarity measure that we will later combine into a single score. We define $\mathtt{MP3}_\cap$ as the similarity computed between triplets of labels shared by the two trees:

$$\mathtt{MP3}_\cap = \frac{\sum_{(a,b,c)\in I} N(a,b,c)}{\sum_{(a,b,c)\in I} D(a,b,c)} \tag{6.1}$$

where I is the set of triples in $\lambda(T_1) \cap \lambda(T_2)$. Due to the nature of only considering the subset of labels that appears in both trees, $\mathtt{MP3}_\cap$ is a conservative measure, therefore we present a variation that consider all possible configurations in both trees, thus having a wider view:

$$\mathtt{MP3}_\cup = \frac{\sum_{(a,b,c)\in J} N(a,b,c)}{\sum_{(a,b,c)\in J} D(a,b,c)} \tag{6.2}$$

where J is the set of triples in $\lambda(T_1) \cup \lambda(T_2)$. Differently from $\mathtt{MP3}_\cap$, $\mathtt{MP3}_\cup$ weighs also the the labels that appear only in one of the trees. Note that, for every pair of trees, $\mathtt{MP3}_\cup \leq \mathtt{MP3}_\cap$, as the numerator remains identical in both, while the denominator of $\mathtt{MP3}_\cup$ has all the elements in $\mathtt{MP3}_\cap$ with the addition of the values of D for the triples present only in one of the input trees.

Although $\mathtt{MP3}_\cap$ and $\mathtt{MP3}_\cup$ are closely related, they provide two different views of a tumor phylogeny. Indeed, on one hand $\mathtt{MP3}_\cap$ measures how similar the shared history of two tumor phylogenies is, i.e., it provides an idea of how well the two progressions can be reduced to the same subsequence of common mutations. On the other hand, $\mathtt{MP3}_\cup$ measures how similar the whole history of the two evolutions are, i.e., it considers the impact of mutations acquired only in one progression.

Since the previous measures capture different aspects of the progressions, we want to combine them into a single, usable and powerful similarity measure that couples the strengths of both. The most intuitive method is to simply use a mean. We opted for the geometric mean: $\mathtt{MP3}_G = \sqrt{\mathtt{MP3}_\cap \cdot \mathtt{MP3}_\cup}$.

This function is not completely satisfactory, as a uniform function of 6.1 and 6.2 is not able to comprehensively capture the nuances in the input trees. Therefore we developed a weighted mean with an intentional bias towards $\mathtt{MP3}_\cap$ to catch inner similarities in different trees. Such combination then tends to be closer to $\mathtt{MP3}_\cap$ when the trees are similar while moving towards $\mathtt{MP3}_\cup$ as the trees are less similar:

$$\mathtt{MP3}_\sigma = \mathtt{MP3}_\cup + \sigma(\mathtt{MP3}_\cap) \cdot \min\{\mathtt{MP3}_\cap - \mathtt{MP3}_\cup, \mathtt{MP3}_\cup\},$$

where $\sigma(x) = (1 + e^{\mu(x-\frac{1}{2})})^{-1}$ is the classic sigmoid function centered in $1/2$ and μ is used to adjust the slope of the curve; we set $\mu = 10$ in our experimentation. In addition the sigmoid polarizes the values close to $1/2$, thus helping decide whether they are closer to 1 or 0, therefore moving the final score closer to $\mathtt{MP3}_\cap$ or $\mathtt{MP3}_\cup$.

While all four measures are available in our implementation, we decided to use $\texttt{MP3}_\sigma$ as default measure and is denoted simply as `MP3`. An experimental comparison of all four measures is shown in Appendix B.

6.3 Results

6.3.1 Simulated Data

To perform our experiments we follow an approach similar to the one performed in [126]. We start from a base tree on which we apply a series of perturbations selected from: label swapping, label removal, label duplication, node swapping and node removal. Both the perturbations and the nodes and labels on which they are applied are chosen at random: our procedure allows to select a user-specified total number of actions and a probability vector that will be used to select the perturbations from the previous list.

For the measure comparison experiments, we generated 30 perturbations from each of the 5 base trees, for a total of 150 trees. For the clustering evaluation, 3 base trees are entirely different from each other, and another 2 are perturbations of two of the others, to simulate similar sub-families of the same tumor type: we perform a total of 10 perturbations on such 5 trees. More details on the perturbation parameters will be described in each section, while the entire configuration is available and reproducible at `https://github.com/AlgoLab/mp3treesim_supp`.

6.3.2 Measures comparison

We compared `MP3` against all the different versions of DISC and CASet from [126] and MLTD [225]. While `MP3` and MLTD provide similarity scores, DISC and CASet compute a dissimilarity score, that we convert into a similarity measure by simply subtracting their value from 1.

Effect of changes in the tree topology

A key feature a measure on tumor phylogenies should have is to discern changes at different tree depths; indeed, a change close to the root should be more impactful than a change towards the leaves. Such a behavior is fundamental, as driver mutations are often acquired early in the evolutionary history, while less important passenger mutations usually happen at later stages: to mistake the two types of mutations should therefore have a high impact on a good similarity measure.

To estimate this effect on all the measures, we start from a linear base tree (T_0 in Figure 6.4 (*Left*)); we then raise its only leaf one level at the time and compute its similarity to the base tree, expecting a drop in similarity as the leaf raises to the root, similarly to experiment proposed in [126]. Figure 6.4 (*Left*) clearly displays such effect for `MP3`, showing that it has the highest similarity decrease among all measures; DISC and CASet also have similar trends, but to a lower extent. Since the set of labels is the same for all trees, there is no difference between union and intersection versions of DISC and CASet. Contrarily, as already observed in [126], MLTD plateaus after the first change.

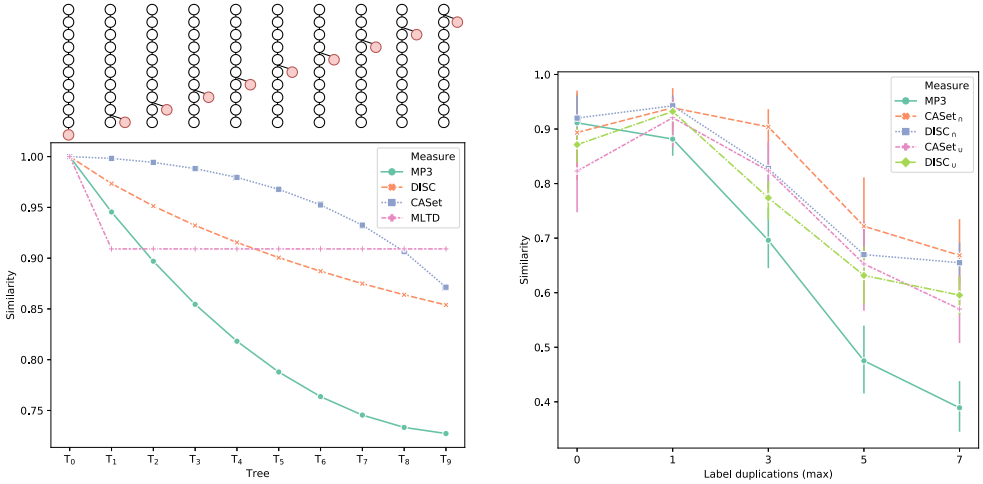


Figure 6.4: (*Left*) Effect of a node (highlighted in red) that ascends from leaf to child of the root, T_0 is the base tree to which the others are compared. (*Right*) Effect of label duplication on the similarity scores. Similarities are the average of 15 trees generated from the same base with the specified maximum number of duplications. MLTD was excluded since it failed to run on instances with poly-occurring labels.

Another interesting aspect to investigate is how the presence of poly-occurring labels influences the similarity scores, as the more sophisticated the inference tools get, the more is common to have tumor phylogenies with multiple acquisitions or losses of the same mutation. To evaluate this aspect we started from a multi-labeled base tree with all labels occurring only once. We then created 15 perturbed trees for 5 different configurations. In the first one (on the abscissa 0 in Figure 6.4 (*Right*)) we allowed one operation excluding label duplication; for the others we allowed a total of 1, 3, 5 and 7 operations with much higher chance of selecting a label duplication. Since perturbations occur randomly, we are only sure that at most the specified number of duplication occurred, and not necessarily to the same label.

Figure 6.4 (*Right*) shows that $\text{CASet}_\cap$, $\text{CASet}_\cup$, $\text{DISC}_\cap$ and $\text{DISC}_\cup$ have similar trends in this setting, `MP3` being the only one that differs. In particular, the other measures assign an higher similarity score to the second configuration than to the first one, despite they are both obtained with one perturbing operation, allowing label duplication only in the second one. `MP3` is the only measure that positively displays a monotonic decrease in similarity as the number of poly-occurring labels increases, being markedly steeper than the others. We believe that a larger steepness will be more informative than a plateauing curve, since while being true that after many of poly-occurrences no more information is gained, all the duplications will inevitably add more and more noise to the tree. Since MLTD assumes that every label appears only once, it failed to run on this experiment and was therefore excluded.

Results on simulated data

To analyze the differences between all measures we designed two experimental settings: from 5 different base trees (available in Appendix B) we generated 30 perturbations for each class and computed similarities scores between all the 150 resulting trees. In the first configuration we allowed a total of 3 operations excluding label duplications, while in the second one we allowed them. All the parameters and the different probabilities used for applying perturbations are available at our supplementary repository `https://github.com/AlgoLab/mp3treesim_supp`.

Results for the first configuration are shown in Figure 6.5. The heatmaps (*Left*) show that `MP3` discerns the best between the trees in the same class (main diagonal) and the others: the results of $\text{DISC}_\cup$ are really close to ours, but there is a more noticeable noise outside the main diagonal. $\text{DISC}_\cap$ and $\text{CASet}_\cup$ present even more noise than the others, but are still mostly able to distinguish the different classes; $\text{CASet}_\cap$ seems to struggle the most on this setting, while MLTD displays high values of similarities for every couple of trees, but it is still able to differentiate between the bases.

The boxplots in Figure 6.5 (*Top-Right*) show the same result quantitatively: the crucial feature is to correctly distinguish the different classes. The values represent the distribution of the similarities between the trees in the same class (Intra-similarity) and in different classes (Inter-similarity). `MP3` differentiates better between intra and inter similarity, exhibiting the most compact distribution for the inter-similarities scores, while being a little more dispersed on the intra-similarity due to the action of the sigmoid, that pulls apart the values around $1/2$. Similarly to the previous case, $\text{DISC}_\cup$, $\text{CASet}_\cup$ and MLTD show similar trends, while $\text{CASet}_\cap$ displays the most overlapping distributions.

Lastly, in Figure 6.5 (*Bottom-Right*), we computed a silhouette score from the data using a hierarchical linkage clustering with cuts from 2 to 15 to simulate a clustering scenario. Once again, `MP3` performs the best expressing the maximum value for 5 cuts, being the 5 classes. $\text{DISC}_\cap$, $\text{DISC}_\cup$ also show the largest value at the same cut. MLTD was excluded from the plot since it scored values close to -1 for every cut, thus causing the figure to be hard to interpret.

In the second experimental setting we introduced poly-occurring labels to the simulation. Figure 6.6 exhibits results very similar to the previous ones. The main difference is that in the silhouette score (*Bottom-Right*) `MP3`, while still having its maximum value in correspondence of 5 cuts, is slightly lower than the other measures. On this experiment MLTD, not allowing poly-occurring labels, failed to compute the score in most of the instances, shown in grey in the heatmaps (*Left*); it was excluded from the other plots given the high amount of failed runs.

6.3.3 Application to clustering of trees

A very important application of a tree similarity measure is clustering, e.g., to classify cancer type of patients by the similarity of their inferred phylogenies. This is of crucial interest for the development of precision therapies based on the topological structure and the evolution of mutations. Since to curate such classifications manually would be unfeasible as the size and the number of mutations increases, a good measure to use in conjunction with a clustering method is necessary.

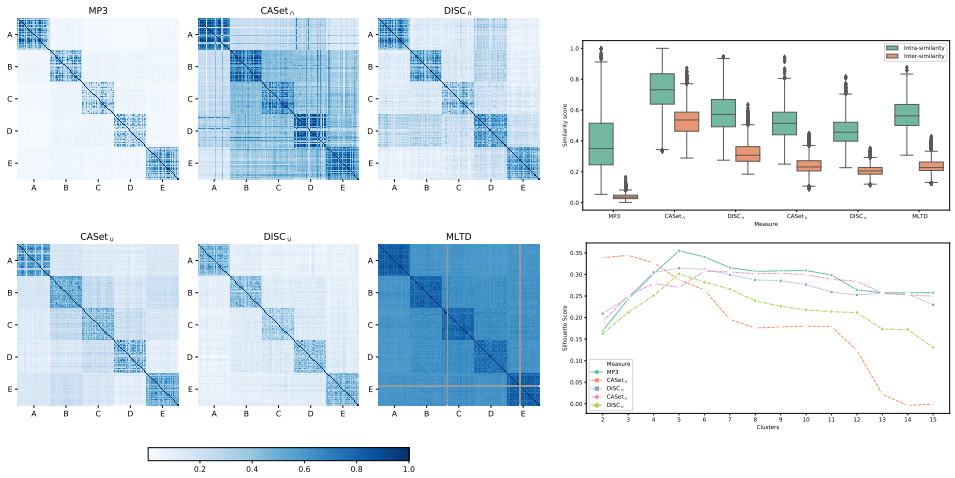


Figure 6.5: Results for the first experimental configuration: (*Left*) Heatmaps displaying the scores between all-pairs 150 simulate trees. (*Top-right*) Distribution of the similarities between the trees in the same class (Intra-similarity) and in different classes (Inter-similarity). (*Bottom-right*) Silhouette score computed using a hierarchical linkage clustering with cuts from 2 to 15.

To evaluate a similar scenario we started from 3 different bases, then perturbing two of such trees chosen at random; these new trees are then considered as additional base trees. Given this 5 bases we created a total of 10 perturbed trees from each class. The goal was to simulate an experiment with three separate classes, with two of them further split in two subclasses, to obtain subtypes of the same cancer families. The five resulting bases are available in Appendix B and the parameters used for the simulations are in our supplementary repository.

Results for the clustering experiment are reported in Figure 6.7; (*a*) shows the clustermaps computed using hierarchical linkage clustering. `MP3`, $\text{DISC}_\cap$ and $\text{DISC}_\cup$ correctly cluster the three main families as well as the two sub-families, while both versions of CASet struggle the most in this experiment. Figure 6.7 (*b*) displays the distribution of intra- and inter-similarity between the five bases; `MP3` has the most compact inter-similarity distribution and is the only method that completely separates intra- and inter-distributions. The high number of outliers for all methods is due to the high similarity of the two subclasses. To confirm this hypothesis we computed the same distributions only for the three main classes, remapping the subclasses to the original corresponding base class in (*d*), where we note that the number of outliers is significantly reduced. Finally, Figure 6.7 (*c*) shows the silhouette scores for the dataset; all measures express a higher score with 3 cuts, suggesting that the two subclasses are very similar to the two main bases they are derived from. The scores are very similar for all measures, with $\text{DISC}_\cup$ having a higher value with 3 cuts and `MP3` having a slightly higher with 5 clusters. $\text{CASet}_\cap$ is the only method that have a much higher score in 5, however, as shown in (*a*), the five clusters it reports are not the correct ones. MLTD

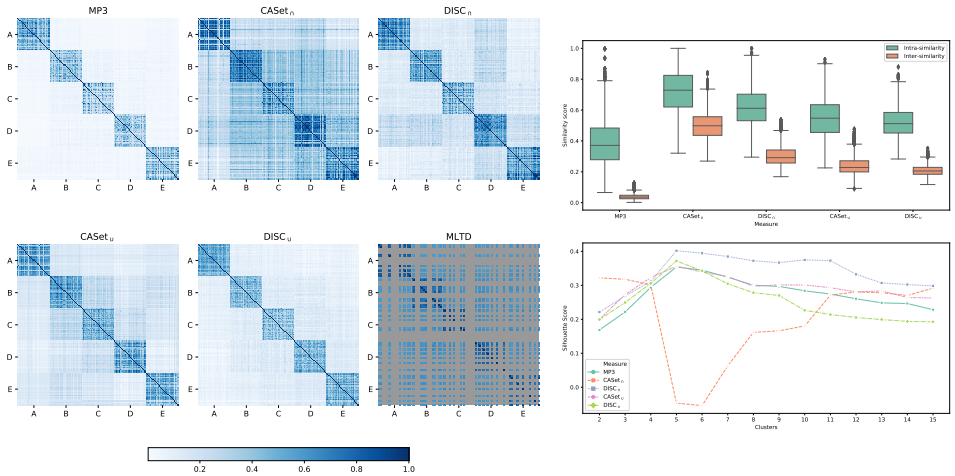


Figure 6.6: Results for the second experimental configuration: (*Left*) Heatmaps displaying the scores between all the 150 simulate trees. (*Top-Right*) Distribution of the similarities between the trees in the same class (Intra-similarity) and in different classes (Inter-similarity). (*Bottom-Right*) Silhouette score computed using a hierarchical linkage clustering with cuts from 2 to 15.

was excluded from this experiment because it failed to run on most instances due to poly-occurring labels.

6.3.4 Application to real dataset

To further evaluate our similarity measure, we applied it to two publicly available real datasets: breast cancer xenoengraftment in immunodeficient mice [134] and ultra-deep-sequencing of clear cell renal cell carcinoma [168]. Both datasets were previously considered for analyses by the two cancer phylogeny reconstruction methods LICHeE [300] and MIPUP [201]. Data from [134] was also used in [126] for evaluation. An interesting feature of the data in [168] is that most samples in the study present poly-occurring labels, suggesting recurrent mutations at different evolutionary stages. We recall that DISC and CASet compute dissimilarity scores, that we convert into a similarity measure subtracting their value from 1. All the analyzed trees are available in Appendix B.

To evaluate the effectiveness of the measures in real scenarios, we selected the manually curated trees, published in the corresponding original sequencing studies, for case SA501 from [134] and for patient RMH002 from [168]. We then computed similarities between these reference trees and the ones inferred by LICHeE and MIPUP, as reported in [201].

The reference RMH002 is very similar to the evolutions inferred by LICHeE and MIPUP, thus most of the measures agree on a high similarity score, as reported in Figure 6.8 (*Left*), with the exception of CASet$_\cup$. The scores computed by `MP3` are higher than the others, possibly because it is the only method to correctly identify and

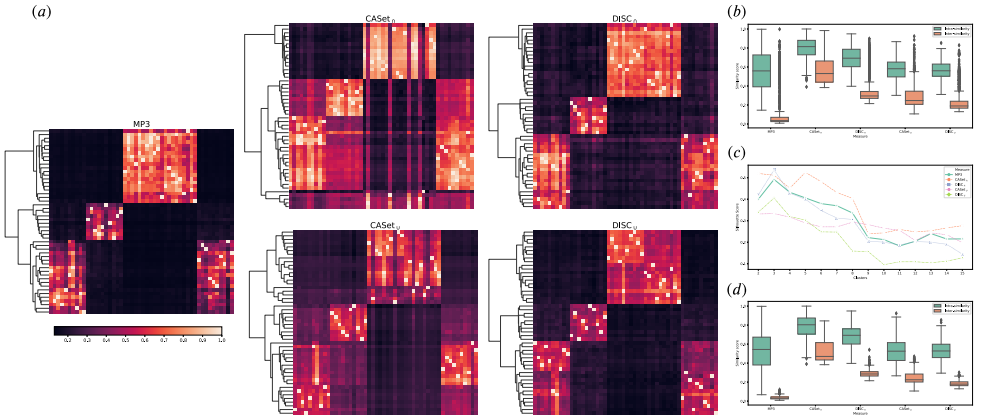


Figure 6.7: Results for the clustering experiment: (*a*) Clustermaps of the 50 simulated trees computed using hierarchical linkage clustering. (*b*) Distribution of the similarities between the trees in the same class (Intra-similarity) and in different classes (Inter-similarity) for the 5 classes. The high number of outliers for all methods is due to the high similarity of the two subclasses. (*c*) Silhouette score computed using a hierarchical linkage clustering with cuts from 2 to 15. (*d*) Distribution of the similarities between the trees in the same class (Intra-similarity) and in different classes (Inter-similarity) for the three main classes, remapping the subclasses to the original corresponding base. MLTD was excluded from this experiment because it failed to run on most instances due to the presence of poly-occurring labels.

process poly-occurring labels in the reference trees, due to the discovered recurring mutations. Differently from the previous analysis, the measures disagree considerably for SA501, as depicted in in Figure 6.8 (*Center*). Indeed, `MP3` reports a similarity value close to 0, suggesting that the considered trees are quite different, whereas the other measures report a higher similarity, especially DISC scoring up to 60% similarity.

To thoroughly investigate this behavior, we defined some naïve approaches used as a proxy to analyze some basic aspects of the trees, such as the count of pairs of labels appearing in the same node in both trees. Even with such a naïve measure, the reference tree for SA501 from [134] and the trees inferred by MIPUP and LICHeE disagree considerably. The base tree contains only 50 labels, whereas the trees inferred by LICHeE and MIPUP contain 95 and 158 labels, respectively; of these, the reference shares a total of 24 label with LICHeE and 37 with MIPUP. Most importantly, only 54 out of 1759 pairs of labels appear in the same node both in the reference and LICHeE and 124 out of 8424 in MIPUP. Such evaluations, albeit very simplistic, suggest that the trees are indeed dissimilar and thus a lower score, as provided by `MP3`, is more reasonable than a high value of similarity.

To better understand this phenomenon, we created the edge case of a single-node tree with all the 158 labels from MIPUP, and compared it against the reference SA501. The resulting values in Figure 6.8 (*Right*) show a high similarity score for DISC with values up to 69%, with CASet and MLTD being less influenced by this aspect with scores up to 11% and 20%. On the other hand, `MP3` clearly defines the trees as extremely dissimilar, with a score of 0.4%. Such results for trees that are clearly extremely different

show a strong bias for DISC towards high similarity values.

	LICHeE	MIPUP
MP3	0.997	0.897
$\text{CASet}_{\cap}$	0.805	0.779
$\text{DISC}_{\cap}$	0.930	0.876
$\text{CASet}_{\cup}$	0.569	0.551
$\text{DISC}_{\cup}$	0.764	0.725
MLTD	0.842	0.807

	LICHeE	MIPUP
MP3	0.017	0.004
$\text{CASet}_{\cap}$	0.139	0.111
$\text{DISC}_{\cap}$	0.627	0.624
$\text{CASet}_{\cup}$	0.260	0.113
$\text{DISC}_{\cup}$	0.405	0.610
MLTD	0.182	0.205

	Edge case
MP3	0.0004
$\text{CASet}_{\cap}$	0.0927
$\text{DISC}_{\cap}$	0.5571
$\text{CASet}_{\cup}$	0.1120
$\text{DISC}_{\cup}$	0.6933
MLTD	0.2046

Figure 6.8: (*Left*) Similarities between the manually curated tree reported in [168] for patient RMH002 and the trees inferred by LICHeE and MIPUP. (*Center*) Similarities between the manually curated tree reported in [134] for sample SA501 and the trees inferred by LICHeE and MIPUP. (*Right*) Similarities between the manually curated tree reported in [134] for sample SA501 and the edge case with all mutations appearing in a single node.

6.4 Discussion

We identified two major limitations in the existing methods to compare tumor phylogenies: first, they are not sensitive enough to detect even major differences in the topology of the trees, as we demonstrated with ad-hoc experiments. Second, they are not able to meaningfully compare trees where the same label is assigned to more than one node.

We addressed the latter by representing tumor phylogenies as multi-labeled trees with poly-occurring labels. Such model is best suited to cancer progression than the ones previously adopted, as it allows the same mutation to appear in multiple evolutionary events, a circumstance often occurring in real applications. Being inspired by the triplet distance for classical phylogenies, our new similarity measure correctly detects differences in the topology of the trees.

Our experiments show that our method performs very well both on synthetic and real data and, unlike the other existing tools, it is able to detect differences regarding poly-occurring labels and it suitably distinguish trees with different topologies. Moreover, when applied to hierarchical clustering, it outperforms every other method.

Chapter 7

Incomplete Directed Perfect Phylogeny in Linear Time

Key Points

Problem. Reconstructing the evolutionary relationship of a set of species is a central task in computational biology. In real data, it is often the case that some information is missing: the Incomplete Directed Perfect Phylogeny (IDPP) problem asks, given a collection of species with some missing information, to complete it in such a way that the result can be explained with a perfect directed phylogeny.

Model. A rooted phylogenetic tree models the evolutionary history of a set of species: the leaves are in a one-to-one correspondence with the species, all of which have a common ancestor represented by the root. We adopt the model describing the species by a set of binary characters, so that each species is described by the states of its characters. Such a representation is naturally encoded by a matrix $\mathcal{A}$, $a_{i,j}$ being the state of character j in species i. A directed perfect phylogeny is such that the set of all nodes that have the same character state induces a connected subtree, and on any path from the root to a leaf a character can change its state from 0 to 1, but the opposite cannot happen. The input of the problem we are considering is a matrix of character vectors in which some character states are unknown, and the question is whether it is possible to complete the missing states in such a way that the result can be explained with a directed perfect phylogeny.

Included Works

This chapter presents the ongoing work **Incomplete Directed Perfect Phylogeny in Linear Time** [50].

7.1 Introduction

A rooted phylogenetic tree models the evolutionary history of a set of species: the leaves are in a one-to-one correspondence with the species, all of which have a common ancestor represented by the root. A way of describing the species is by a set of characters that can assume several possible states, so that each species is described by the states of its characters. Such a representation is naturally encoded by a matrix $\mathcal{A}$, $a_{i,j}$ being the state of character j in species i.

When, for each possible character state, the set of all nodes that have the same state induces a connected subtree, a phylogeny is called *perfect*. The problem of reconstructing a perfect phylogeny from a set of species is known to be linearly-solvable in the case when the characters are binary [183], and it is NP-hard in the general case [66]. A popular variant of binary perfect phylogeny requires that the characters are directed, that is, on any path from the root to a leaf a character can change its state from 0 to 1, but the opposite cannot happen [87].

In this chapter, we study the Incomplete Directed Perfect Phylogeny problem (IDPP for short) introduced by Pe'er et al. [293], assuming that the characters are binary, directed, and can be gained only once. The input of this problem is a matrix of character vectors in which some character states are unknown, and the question is whether it is possible to complete the missing states in such a way that the result can be explained with a directed perfect phylogeny.

Related work. Besides being relevant in its own right [288, 46, 230, 318, 335], the problem of handling phylogenies with missing data arises in various tasks of computational biology, like resolving genotypes with some missing information into haplotypes [228] and inferring tumor phylogenies from single-cell sequencing data with mutation losses [317]. A generalization of the perfect phylogeny model where a character can be gained only once and can be lost at most k times, called the k-Dollo model [69, 185, 70, 135], has also been extensively studied. It should be clear that different and efficient solutions for the IDPP problem may highlight novel approaches for the above mentioned computational frameworks.

The approach of Pe'er et al. [293] to the IDPP problem is graph theoretic: their algorithm relies on maintaining the connected components of a graph under a sequence of edge deletions. The use of pre-existing dynamic connectivity data structures for this purpose is the bottleneck in the overall time complexity. A connectivity data structure is *fully-dynamic* when both edge insertion and deletion are allowed, and *decremental* when only edge deletion is considered. A long line of results brought down the computational time required for updating the data structure after edge insertions and/or deletions, and for answering connectivity queries, to roughly logarithmic: the following table summarizes the results for both fully-dynamic and decremental connectivity on a graph consisting of N nodes and M edges. For fully-dynamic connectivity we report the update time required for a single edge insertion or deletion, while for decremental connectivity we report the overall time required to eventually delete all the edges. All of the listed results, except for [193], assume that edge deletions can be interspersed with connectivity queries. The algorithm of Henzinger et al. [193], in contrast, deletes edges in batches (b_0 is the number of batches that do not result in a new component)

and connectivity queries can be only asked between one batch of deletions and another.

Fully-Dynamic	**Update time**	**Query time**
Holm et al. [195]	$\mathcal{O}(\log^2 N)$, amortized	$\mathcal{O}(\log N / \log\log N)$
Gibb et al. [169]	$\mathcal{O}(\log^4 N)$, worst case	$\mathcal{O}(\log N / \log\log N)$ w.h.p.
Huang et al. [199]	$\mathcal{O}(\log N (\log\log N)^2)$, exp. amortized	$\mathcal{O}(\log N / \log\log\log N)$
Decremental	**Total update time**	**Query time**
Even et al. [138]	$\mathcal{O}(MN)$	$\mathcal{O}(1)$
Thorup [347]	$\mathcal{O}(\min\{N^2, M\log N\} + \sqrt{MN}\log^{2.5} N)$, exp.	$\mathcal{O}(1)$
Henzinger et al. [193]	$\mathcal{O}(N^2 \log N + b_0 \min\{N^2, M\log N\})$	$\mathcal{O}(1)$

By plugging in an appropriate dynamic connectivity structure, the worst case running time of the approach of Pe'er et al. [293], given a matrix describing n species and m characters, becomes deterministic $\mathcal{O}(nm\log^2(n+m))$ (using fully dynamic connectivity structure of Holm et al. [195]), expected $\mathcal{O}(nm\log((n+m)^2/nm) + (n+m)\log^3(n+m)\log\log(n+m))$ (using decremental connectivity structure of Thorup [347]), expected $\mathcal{O}(nm\log(n+m)(\log\log(n+m))^2)$ (using fully dynamic connectivity structure of Huang et al. [199]), or deterministic $\mathcal{O}((n+m)^2\log(n+m))$ (using decremental structure of Henzinger et al. [193]). This should be compared with a lower bound of $\Omega(nm)$, following from the work of Gusfield on directed binary perfect phylogeny [183] (under the natural assumption that the input is given as a matrix). For $n = m$, the second algorithm achieves this lower bound at the expense of randomisation (and being very complicated), while for the general case the asymptotically fastest solution is still at least one log factor away from the lower bound.

Inspecting the algorithm of Pe'er et al. [293], we see that it operates on bipartite graphs and only needs to deactivate nodes on one of the sides. It seems plausible that some of the known dynamic connectivity structures are actually asymptotically more efficient on such instances. However, all of them are very complex (with the result of Holm et al. [195] being the simplest, but definitely not simple), and this is not clear. Furthermore, recently Fernández-Baca and Liu [142] performed an experimental study of the algorithm of Pe'er et al. for IDPP [293] with the aim of assessing the impact of the underlying dynamic graph connectivity data structure on their solution. Specifically, they tested the use of the data structure of Holm et al. [195] against a simplified version of the same method, and showed that, in this context, simple data structures perform better than more sophisticated ones with better asymptotic bounds.

Our results and techniques. We are motivated to look for simple, ad-hoc methods that make use of the properties of the decremental connectivity as used in IDPP. In this case, the graph is bipartite, and the required updates are vertex deletions from just one of the two sides. We thus start by describing a simple data structure that dynamically maintains the connected components of a bipartite graph with N nodes on each side, whilst vertices are removed from one side of the graph. The starting point for our solution is an application of a version of the sparsification technique of Eppstein et al. [136]: we define a hierarchical decomposition of the graph, and maintain a forest representing the connected components of each subgraph in this decomposition.

Recall that the original description of this technique focused on inserting and deleting edges, while we are interested in deleting nodes (and only from one side of the graph). Therefore, the decomposition needs to be appropriately tweaked for this particular use case. This allows us to obtain an extremely simple data structure with $\mathcal{O}(N^2 \log N)$ total update time, which we show to imply an $\mathcal{O}(nm \log n)$ algorithm for IDPP.

The main technical part of this chapter refines this solution to shave the logarithmic factor and thus obtain an asymptotically optimal algorithm. We stress that while Eppstein et al. [136] did manage to avoid paying any extra log factors by applying a more complex decomposition of the graph than a complete binary tree (used in the conference version of their paper), this does not seem to translate to our setting, as we operate on the nodes instead of the edges. The high-level idea is to amortize the time spent on updating the forest representing the components of every subgraph with the progress in disconnecting its nodes, and re-use the results from the subgraph on the previous level of the decomposition to update the subgraph on the next level. As a consequence, the IDPP problem can be solved in time linear in the input size:

Theorem 28. *Given an incomplete matrix $\mathcal{A}_{n\times m}$, the* IDPP *problem can be solved in time $\mathcal{O}(nm)$.*

Under the natural assumption that the input is given as a matrix, this is asymptotically optimal [183].

Chapter organization. In Section 7.2 we provide a description of the algorithm of Pe'er et al. [293] and a series of preliminary observations. In Section 7.3 we show a simple and self-contained dynamic connectivity data structure that implies an $\mathcal{O}(nm \log n)$ time solution for the IDPP problem for an incomplete matrix $\mathcal{A}_{n\times m}$. Finally, in Section 7.4 we present the main result of this chapter and describe a dynamic connectivity data structure that implies a linear-time algorithm for IDPP.

7.2 Preliminaries

Basic definitions. Let $G = (V, E)$ be a graph. The subgraph induced by $V' \subseteq V$ is the graph $G_{V'} = (V', E \cap (V' \times V'))$. We say that a forest $F = (V, E')$ represents the connected components of $G = (V, E)$ when the connected components of F and G are the same (note that we do not require that $E' \subseteq E$). Throughout this chapter, we will use the term node for trees, and vertex for other graphs. We denote by $S = \{s_1, \ldots, s_n\}$ the set of species and by $C = \{c_1, \ldots, c_m\}$ the set of characters. A matrix of character states $\mathcal{A}_{n\times m} = [a_{ij}]_{n\times m}$, where each entry is a state from $\{0, 1, ?\}$ and the rows correspond to the species, is said to be *incomplete*. The state a_{ij} of a character j for a species i is one, zero or ? depending on whether character j is present, absent or unknown for species i. A completion $\mathcal{B}_{n\times m}$ of such $\mathcal{A}_{n\times m}$ is obtained by replacing the ? entries of $\mathcal{A}_{n\times m}$ with either 0 or 1: formally, $\mathcal{B}_{n\times m}$ is a binary matrix with entries $b_{ij} = a_{ij}$ for each i, j such that $a_{ij} \neq ?$.

A phylogenetic rooted tree $\mathcal{T}$ for a binary matrix $\mathcal{B}_{n\times m}$ has the n species of S at the leaves, and there is a surjection from the set of characters C to the internal nodes of $\mathcal{T}$ such that, if a character c_j is associated with a node x, then s_i belongs to the

leaf set of the subtree rooted at x if and only if $b_{ij} = 1$. In other words, all and only the species in a subtree associated with a character c_j have the character c_j. We say that an incomplete matrix admits a phylogenetic tree if there exists a completion of the matrix that has such a tree. The Incomplete Directed Perfect Phylogeny problem (IDPP for short), introduced by Pe'er et al. in [293], asks, given an incomplete matrix $\mathcal{A}$, to find a phylogenetic tree for $\mathcal{A}$, or determine that no such tree exists.

For a character c_j, the 1-set (resp. 0-set and ?-set) of c_j in an incomplete matrix $\mathcal{A}$ is the set of species $\{s_i | a_{ij} = 1\}$ (resp. $a_{ij} = 0$ and $a_{ij} = ?$). For a subset $S' \subseteq S$ of species, a character c is S'-semiuniversal in $\mathcal{A}$ if its 0-set does not intersect S', that is, if $\mathcal{A}[s, c] \neq 0$ for all $s \in S'$. It is convenient to represent the character state matrix as a graph: the vertices are $V = S \cup C$ and the edges are $S \times C$, partitioned into $E_1 \cup E_? \cup E_0$, with $E_x = \{(s_i, c_j) | a_{ij} = x\}$ for $x \in \{0, 1, ?\}$. The edges of $E_1, E_?, E_0$ are called *solid*, *optional*, and *forbidden*, respectively. We denote by $G(\mathcal{A}) = (S \cup C, E_1)$ the bipartite graph consisting only of the solid edges.

Previous solutions. The existence of a phylogenetic tree for $\mathcal{A}$ is linked with the existence, in its graph representation, of a subset of edges with certain properties. Specifically, Pe'er et al. show that finding a subset $D \subseteq (E_1 \cup E_?)$ such that $E_1 \subseteq D$ and $(S \cup C, D)$ is Σ-*free* (where a Σ is a path consisting of four edges induced by three vertices from S and two vertices from C), or determining that such D does not exist, is equivalent to solving the IDPP problem for $\mathcal{A}$.

Pe'er et al. proposed two algorithms for solving the IDPP problem, both working on the graph representation of $\mathcal{A}$ and relying on some graph dynamic connectivity data structure, the main difference between the two being the data structure they use. For ease of presentation, in what follows we will only consider the algorithm they refer to as Alg_A. The algorithm relies on the following key properties: if an incomplete matrix $\mathcal{A}$ admits a phylogenetic tree, and c is a S-semiuniversal character (meaning that there are no 0s in its column), then the incomplete matrix obtained by setting to 1 all of the entries of column c still admits a phylogenetic tree. Moreover, given a partition $(K_1, \ldots, K_r)$ of $S \cup C$ where each K_i is a connected component of $G(\mathcal{A})$, the incomplete matrix obtained by setting to 0 all entries corresponding to the edges between K_i and K_j, for $i \neq j$, still admits a phylogenetic tree. Then, there is no interaction between the species and characters belonging to different connected components, and the whole reasoning can be repeated on each such component separately.

We denote by $S(K)$ and $C(K)$ the set of species and characters, respectively, of a connected component K of $G(\mathcal{A})$; $\mathcal{A}|_K$ denotes the submatrix of $\mathcal{A}$ corresponding to the species and characters in K. *Deactivating* a character c in $G(\mathcal{A})$ consists in deleting c together with all its incident edges. At a high level, Alg_A works as follows. At each step, for each connected component K_i of $G(\mathcal{A})$, it computes the $S(K_i)$-semiuniversal characters. If, for some K_i, no $S(K_i)$-semiuniversal character exists, it can be proven that, for any $D \subseteq (E_1 \cup E_?)$ such that $E_1 \subset D$, the graph $(S \cup C, D)$ is not Σ-free, therefore the process halts and reports that $\mathcal{A}$ does not admit a phylogenetic tree. Otherwise, it sets to 1 all of the entries of $\mathcal{A}|_{K_i}$ corresponding to the $S(K_i)$-semiuniversal characters, and sets to 0 the entries of $\mathcal{A}$ between vertices that lay in different connected components. It then deactivates all of the $S(K_i)$-semiuniversal characters and updates the connected components of $G(\mathcal{A})$ using some dynamic connectivity data structure.

Algorithm 10 summarizes the high-level structure of Alg_A: for the sake of clarity, we only included the steps that compute the information needed for determining whether $\mathcal{A}$ has a phylogenetic tree, and we left out the operations that actually reconstruct the tree. A complete pseudocode and a proof of correctness of the algorithms can be found in [293].

Algorithm 10: The high-level structure of Alg_A [293].

```
1 while there is at least one character in G(A) do
2     Find the connected components of G(A);
3     for each connected component K_i of G(A) with at least one character do
4         Compute the set U of all characters in K_i which are S(K_i)-semiuniversal
           in A;
5         if U = ∅ then return FALSE;
6         Deactivate every c ∈ U;
7     return TRUE
```

Preliminary results. Our goal is to improve Alg_A by optimizing its bottleneck, that is maintaining the connected components of $G(\mathcal{A})$. We will represent the connected components of a bipartite graph G using the following lemma, and call the resulting representation a *list-representation* of G.

Lemma 44. *The connected components of a bipartite graph $G = (S \cup C, E)$ can be represented in $\mathcal{O}(|S| + |C|)$ space so that, given a vertex, we can access its component, including the size and a pointer to the list of species and characters inside, in constant time, and move a vertex to another component (or remove it from the graph) also in constant time.*

Proof. Each component of G is represented by a doubly-linked list of its vertices (more precisely, a list of species and a list of characters), and also stores the size of the list. An array of length $n + m$, indexed by the vertices of G, stores a pointer from each vertex to its component and another pointer from each vertex to its position in the list of that component. The components are, in turn, organised in a doubly-linked list. Such representation takes space linear in the number of vertices and allows us to access all the required information in constant time. Further, removing or moving a vertex to another component takes constant time. □

Given a list-representation of G, we represent its connected components with another graph $F = (V, E')$ consisting of *rooted stars* [324] as follows. For each component K, we define the central vertex $v \in K$ to be the first vertex on the list of K. Then, we add an edge (u, v) to E', for any $u \in K$ with $u \neq v$. This construction can be implemented in $\mathcal{O}(|V|)$ time. Observe that we only guarantee that the connected components of G and F are the same, but E' is not required to only consist of the edges of G. We can use the list-representation of G to simulate access to the adjacency lists of F without constructing it explicitly, as stated by the following lemma.

Lemma 45. *Given a bipartite graph $G = (S \cup C, E)$ and a list-representation of G, the access to the adjacency lists of a star forest F representing the connected components of G can be simulated in constant time without constructing F explicitly.*

Proof. To access the adjacency list of a vertex v we first look up its component K and retrieve the first vertex u on the list of K. By Lemma 44, this operation requires constant time. If $u = v$, then the adjacency list of v is the list of vertices of K stored in the list-representation of G. Otherwise, the adjacency list of v consists only of a single vertex u. □

We are interested in solving the following special case of decremental connectivity:

Problem: (N_ℓ, N_r)-DC
Input: a bipartite graph $G = (S \cup C, E)$ with $N_\ell = |S|$ and $N_r = |C|$.
Update: deactivate a character $c \in C$.
Query: return the connected components of the subgraph induced by S and the remaining characters.

When analysing the complexity of (N_ℓ, N_r)-DC, we allow preprocessing the input graph G in $\mathcal{O}(N_\ell N_r)$ time, and assume that all characters are eventually deactivated when analysing the total update time. We can of course deactivate multiple characters at once by deactivating them one-by-one. The overall time complexity of Algorithm 10 depends on the complexity of (N_ℓ, N_r)-DC as follows.

Lemma 46. *Consider an $n \times m$ incomplete matrix $\mathcal{A}$. If the (n, m)-DC problem can be solved in $f(n, m)$ total update time and $g(n, m)$ query time, then the* IDPP *problem can be solved for $\mathcal{A}$ in time $\mathcal{O}(nm + f(n, m) + \min\{n, m\} \cdot g(n, m))$.*

Proof. There are three nontrivial steps in every iteration of the while loop: finding the connected components in line 2, computing the semiuniversal characters of every connected component in line 4, and finally deactivating characters in line 6. Every character is deactivated at most once, so the overall complexity of all deactivations is $\mathcal{O}(f(n, m))$. We claim that in every iteration of the while loop, except possibly for the very last, (1) at least one character is deactivated, and (2) there exist two species that cease to belong to the same connected component. (1) is immediate, as otherwise we have a connected component K_i with no $S(K_i)$-semiuniversal characters and the algorithm terminates. To prove (2), assume otherwise, then we have a connected component K_i such that $S(K_i)$ does not change after deactivating all $S(K_i)$-semiuniversal characters. But then in the next iteration the set of $S(K_i)$-semiuniversal characters is empty and the algorithm terminates. (1) and (2) together imply that the number of iterations is bounded by $\min\{n, m\}$. The overall complexity of finding the connected components is thus $\mathcal{O}(\min\{n, m\} \cdot g(n, m))$.

It remains to bound the overall complexity of computing the semiuniversal characters by $\mathcal{O}(nm)$. This has been implicitly done in [293, proof of Theorem 12], but we provide a full explanation for completeness. For every character $c \in C$, we maintain the count of solid and optional edges connecting c (in the graph representation of $\mathcal{A}$) with the species that belong to its same connected component (of $G(\mathcal{A})$). Assuming that we can

indeed maintain these counts, in every iteration all the semiuniversal characters can be generated in $\mathcal{O}(m)$ time, so in $\mathcal{O}(\min\{n,m\} \cdot m) = \mathcal{O}(nm)$ overall time.

To update the counts, consider a connected component K that, after deactivating some characters, is split into possibly multiple smaller components $K_1, K_2, \ldots, K_k$. Note that we can indeed gather such information in $\mathcal{O}(n+m)$ time, assuming access to a representation of the connected components before and after the deactivation. We assume that the connected components are maintained with the list-representation described in Lemma 44, and therefore we can access a list of the vertices in every K_i. Then, we consider every pair $i, j \in \{1, 2, \ldots, k\}$ such that $i \neq j$, $C(K_i) \neq \emptyset$ and $S(K_j) \neq \emptyset$. We iterate over every $c \in K_i$ and $s \in K_j$, and if (s, c) is an edge in the graph of $\mathcal{A}$ (observe that it cannot be a solid edge, as K_i and K_j are distinct connected components) we decrease the count of c. By first preparing lists of components K_i such that $C(K_i) \neq \emptyset$ and $S(K_i) \neq \emptyset$, this can be implemented in time bounded by the number of considered possible edges (s, c), and every such possible edge is considered at most once during the whole execution. Therefore, the overall complexity of maintaining the counts is $\mathcal{O}(nm)$. Additionally, we need $\mathcal{O}(nm)$ time to initialise the (n, m)-DC structure. □

Before we proceed to design an efficient solution for the (N_ℓ, N_r)-DC problem, we first show that it is in fact enough to consider the (N, N)-DC problem.

Lemma 47. *Assume that the (N, N)-DC problem can be solved in $f(N)$ total update time and $g(N)$ query time. Then, for any $N' \geq N$, both the (N, N')-DC problem and the (N', N)-DC problem can be solved in $\mathcal{O}(N'/N \cdot f(N))$ total update time and $\mathcal{O}(N'/N \cdot g(N))$ query time.*

Proof. We first consider the (N, N')-DC problem. We create $\lceil N'/N \rceil$ instances of (N, N)-DC by partitioning C into groups of N vertices (except for the last group that might be smaller). In each instance we have the same set of species S. Deactivating a character $c \in C$ is implemented by deactivating it in the corresponding instance of (N, N)-DC. Overall, this takes $\mathcal{O}(N'/N \cdot f(N))$ time. Upon a query, we query all the instances in $\mathcal{O}(N'/N \cdot g(N))$ time. The output of each instance can be converted to a star forest representing the connected components in $\mathcal{O}(N)$ time. We take the union of all these forests to obtain an auxiliary graph on at most $\lceil N'/N \rceil \cdot (N-1) = \mathcal{O}(N')$ edges, and find its connected components in $\mathcal{O}(N')$ time. Assuming that $f(N) \geq N$, this takes $\mathcal{O}(N'/N \cdot f(N))$ overall time and gives us the connected components of the whole graph.

Now we consider the (N', N)-DC problem. We create $\lceil N'/N \rceil$ instances of (N, N)-DC by partitioning S into groups of N vertices, and in each instance we have the same set of characters C. Thus, deactivating a character $c \in C$ is implemented by deactivating it in every instance. Overall, this takes $\mathcal{O}(N'/N \cdot f(N))$ time. A query is implemented exactly as above by querying all the instances and combining the results in $\mathcal{O}(N'/N \cdot f(N))$ time. □

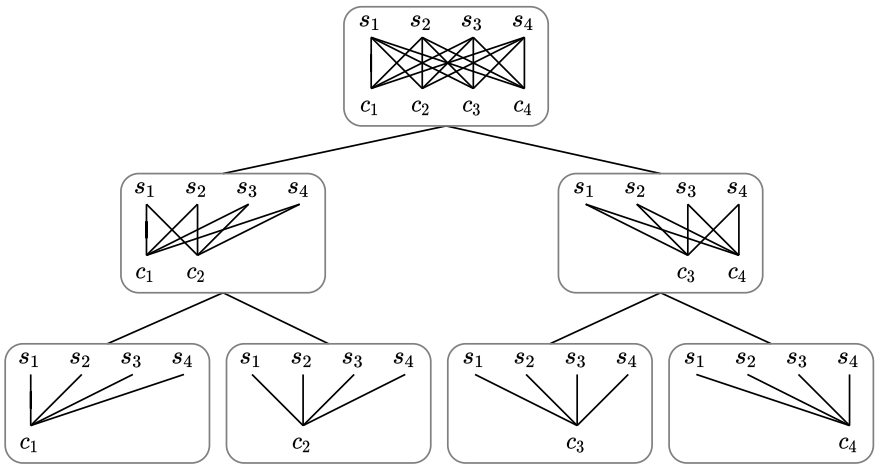


Figure 7.1: The decomposition tree of $K_{4,4}$.

7.3 (N, N)-DC in $\mathcal{O}(N^2 \log N)$ Total Update Time and $\mathcal{O}(N)$ Time per Query

Our solution for the (N, N)-DC problem is based on a hierarchical decomposition of G into multiple smaller subgraphs as in the sparsification technique of Eppstein et al. [136] (as mentioned in the introduction, appropriately tweaked for our use case). The decomposition is represented by a complete binary tree $\mathrm{DT}(G)$ of depth $\log N$. We identify the leaves of $\mathrm{DT}(G)$ with the characters C. Each node v corresponds to the set of characters C_v identified with the leaves in the subtree of v, and is responsible for the subgraph G_v of G induced by C_v and the whole set of species S. Thus, the root is responsible for the whole G, see Figure 7.1. Each node v explicitly maintains a list-representation of the connected components of G_v, denoted $\mathsf{components}(v)$. We stress that, while $\mathsf{components}(v)$ is explicitly maintained, we do not explicitly store G_v at every node v. The initial preprocessing required to construct $\mathrm{DT}(G)$ together with $\mathsf{components}(v)$ for every node v, given G, takes $\mathcal{O}(B^2)$ time by the following argument. First, we construct $\mathsf{components}(v)$ for every leaf c. This can be done in $\mathcal{O}(B)$ time per leaf by simply iterating the neighbours of c in G. Second, we proceed bottom-up and compute $\mathsf{components}(v)$ for every inner node v in $\mathcal{O}(B)$ time using the following lemma.

Lemma 48. *Let v be an inner node of $\mathrm{DT}(G)$, and v_ℓ, v_r be its children. Given $\mathsf{components}(v_\ell)$ and $\mathsf{components}(v_r)$ we can compute $\mathsf{components}(v)$ in $\mathcal{O}(B)$ time.*

Proof. We construct star forests representing the connected components of $\mathsf{components}(v_\ell)$ and $\mathsf{components}(v_r)$ in $\mathcal{O}(B)$ time and take their union. Then we find the connected components of this union in $\mathcal{O}(B)$ time and save them as $\mathsf{components}(v)$. □

We proceed to explain how to solve the (N, N)-DC problem in $\mathcal{O}(N \log N)$ time per update and $\mathcal{O}(N)$ time per query. The query simply returns $\mathsf{components}(r)$, where r is the root of $\mathrm{DT}(G)$. The update is implemented as follows. Deactivating a character c possibly affects $\mathsf{components}(v)$ for all ancestors v of the leaf corresponding to c. In

particular, $\mathsf{components}(c)$ becomes a collection of isolated nodes and can be recomputed in $\mathcal{O}(1+|S|)=\mathcal{O}(N)$ time. We iterate over all proper ancestors v, starting from the parent of c. For each such v, let v_ℓ and v_r denote its left and right child, respectively. We can assume that $\mathsf{components}(v_\ell)$ and $\mathsf{components}(v_r)$ have been already correctly updated. We compute $\mathsf{components}(v)$ from $\mathsf{components}(v_\ell)$ and $\mathsf{components}(v_r)$ by applying Lemma 48 in $\mathcal{O}(N)$ time. When summed over all the ancestors, the update time becomes $\mathcal{O}(N\log N)$, so $\mathcal{O}(N^2\log N)$ over all deactivations.

By Lemmas 46 and 47, this implies that, given an incomplete matrix $\mathcal{A}_{n\times m}$, the IDPP problem can be solved in time $\mathcal{O}(nm\log\min\{n,m\})$ without using any dynamic connectivity data structure as a blackbox.

7.4 (N,N)-DC in $\mathcal{O}(N^2)$ Total Update Time and $\mathcal{O}(N)$ Time per Query

Our faster solution is also based on a hierarchical decomposition $\mathrm{DT}(G)$ of G. As before, every node v stores $\mathsf{components}(v)$, so a query simply returns $\mathsf{components}(r)$. The difference is in implementing an update. We observe that, if for some ancestor v of the leaf corresponding to c, the only change to $\mathsf{components}(v)$ is removing c from its connected component, then this also holds for all of the subsequent ancestors and they can be updated in constant time each. This suggests that we should try to amortise the cost of an update with the progress in splitting $\mathsf{components}(v)$ into smaller components.

We will need to compare the situation before and after the update, and so introduce the following notation. A node v of $\mathrm{DT}(G)$ is responsible for the subgraph G_v before the update and for the subgraph G'_v after the update; $\mathsf{components}(v)$ and $\mathsf{components}'(v)$ denote the connected components of G_v and G'_v, respectively. The crucial observation is that $\mathsf{components}'(v)$ is obtained from $\mathsf{components}(v)$ by removing c from its connected component and, possibly, splitting this connected component into multiple smaller ones, while leaving the others intact.

Deactivating a character c begins with updating naively $\mathsf{components}(c)$ in $\mathcal{O}(N)$ time. Then we iterate over the ancestors of c in $\mathrm{DT}(G)$. Let v_{i+1} be the currently considered ancestor, v_i the ancestor considered in the previous iteration, and u_i be the other child of v_{i+1} (sibling of v_i). Let the component of G_{v_i} containing c be K. As observed above, the components of G'_{v_i} are the same as the components of G_{v_i}, except that K is replaced by possibly multiple components $K_1,K_2,\ldots,K_k$, where $\bigcup_{j=1}^{k}K_j=K\setminus\{c\}$. If $k=1$ then we trivially remove c from its connected component in every G_{v_j}, for $j=i+1,i+2,\ldots$ and terminate the update, so we can assume that $k\geq 2$. We further assume that, after having updated the components of G_{v_i}, we obtained a list of pointers to $K_1,K_2,\ldots,K_k$. Let L be the connected component of c in $G_{v_{i+1}}$, with $K\subseteq L$ because the subgraphs are monotone with respect to inclusion on any leaf-to-root path. Now the goal is to transform $G_{v_{i+1}}$ into $G'_{v_{i+1}}$, to update its components (using $\mathsf{components}'(v_i)$ and $\mathsf{components}(u_i)$), and additionally to obtain a list of pointers to the components obtained by splitting L. See Figure 7.2 for an illustration.

We start by initialising $G'_{v_{i+1}}$ to be $G_{v_{i+1}}$, and by removing c from L. As in the proof of Lemma 48, we will work with an auxiliary graph consisting of the union of two star

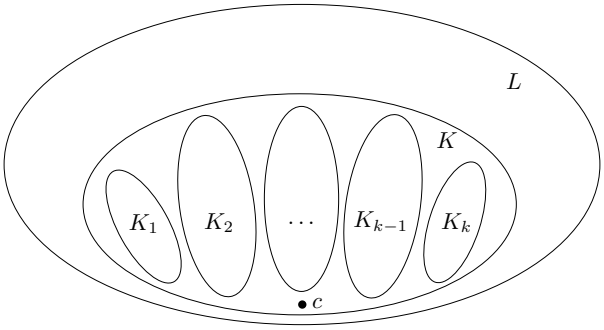


Figure 7.2: After having removed c from K to obtain $K_1, K_2, \ldots, K_k$, we want to remove c from L.

forests representing the connected components of G'_{v_i} and G_{u_i}, respectively. However, instead of explicitly constructing these forests, we simulate access to the adjacency lists of every vertex in both forests using $\mathsf{components}'(v_i)$ and $\mathsf{components}(u_i)$, as explained in the proof of Lemma 45. In turn, this allows us to simulate access to the adjacency list of every vertex in the auxiliary graph. See Figure 7.3 for an example of the auxiliary graph.

By renaming the components we can assume that $|K_1| \geq |K_2|, |K_3|, \ldots, |K_k|$. We will visit the vertices of L in order to determine the new connected components after the removal of c: when doing so, we will use different colours to represent vertices whose new connected component contains K_1 (red), vertices whose new component is different from the one of K_1 (black) and vertices whose new component is still unknown (white). Initially, the vertices of K_1 are red and all of the other vertices of the auxiliary graph are white. This initialisation is done implicitly, meaning that we will assume that all the vertices of K_1 are red and the rest are white without explicitly assigning the colours, and whenever retrieving the colour of a node u we first check if $u \in K_1$, and if so assume that it is red. This allows us to implement the initialisation in constant time instead of $\mathcal{O}(N)$ time. We will perform the visit of L by running the following search procedure from an arbitrarily chosen vertex of each K_j, for $j = 2, 3, \ldots, k$.

The search procedure run from a vertex x first checks if x is white, and immediately terminates otherwise. Then, it starts visiting the vertices of the connected component of x in the auxiliary graph: at any moment, each vertex in such component is either white or red. As soon as the search encounters a red vertex, it is terminated and all the vertices visited in the current invocation are explicitly coloured red. Otherwise, the procedure has identified a new connected component K' of $G'_{v_{i+1}}$. The vertices of K' are removed from L, all vertices of K' are coloured black in the auxiliary graph, and a new component K' of $G'_{v_{i+1}}$ is created in $\mathcal{O}(|K'|)$ time. Inspect Figure 7.3 for an example.

Lemma 49. *The total time spent on all calls to the search procedure in the current iteration is* $\mathcal{O}(|L| - |K_1|)$.

Proof. All vertices visited in the current iteration belong to L. The search is terminated as soon as we encounter a red vertex, and all vertices of K_1 are red from the beginning.

Therefore, each run of the search procedure encounters at most one vertex of K_1, and we can account for traversing the edge leading to this vertex separately paying $\mathcal{O}(k-1) = \mathcal{O}(|L| - |K_1|)$ overall. It remains to bound the number of all other traversed edges. This is enough to bound the overall time of the traversal, because every edge is traversed at most twice, and the number of visited isolated vertices is at most $k - 1 = \mathcal{O}(|L| - |K_1|)$.

For any other edge $e = \{u, v\}$, we have $u, v \in L$ but $u, v \notin K_1$. These edges can be partitioned into two forests by considering whether they originate from $\mathsf{components}'(v_i)$ or $\mathsf{components}(u_i)$. Consequently, we must analyse the total number of edges in a union of two forests spanning $L \setminus K_1$. But this is of course $\mathcal{O}(|L| - |K_1|)$, proving the lemma. □

We now need to analyse the sum of $|L| - |K_1|$ over all the iterations. Because $\bigcup_{j=1}^{k} K_j \subseteq L$, we can split this expression into two parts:

1. $L \setminus \bigcup_{j=1}^{k} K_j$,
2. $\sum_{j=2}^{k} |K_j|$.

Because the sets $L \setminus \bigcup_{j=1}^{k} K_j$ considered in different iterations are disjoint, the first parts sum up to $\mathcal{O}(n)$. It remains to bound the sum of the second parts. This will be done by the following argument. Consider an arbitrary G_v corresponding to a subgraph induced by all the species and a subset of 2^d characters. Whenever its connected component K is split into smaller connected components $K_1, K_2, \ldots, K_k$ after deactivating a character c in the subtree of v, the second part $\sum_{j=2}^{k} |K_j|$ is distributed among the vertices of $\bigcup_{j=2}^{k} K_j$. That is, each node of $\bigcup_{j=2}^{k} K_j$ pays 1. Observe that the size of the connected component containing such a node decreases by a factor of at least 2, because $|K_2|, |K_3|, \ldots, |K_k| \leq |K|/2$. To bound the sum of second parts, we analyse the total cost paid by all the nodes of G_v due to deactivating the characters in the subtree of v (recall that in the end all such characters are deactivated).

Lemma 50. *The total cost paid by the nodes of G_v, over all 2^d deactivations affecting v, is $\mathcal{O}(N \cdot d)$.*

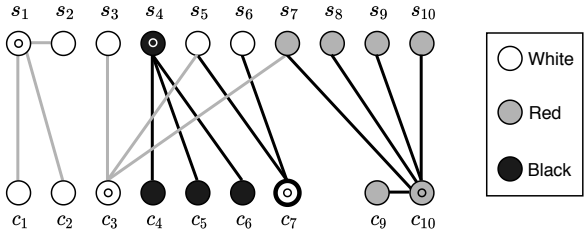


Figure 7.3: The auxiliary graph implicitly constructed for a node v_{i+1} after deactivating c_8. Black edges are used for the star forest of v_i, grey edges for the star forest of u_i; an inner circle identifies the central vertices. K_1 is the rightmost component; c_7 is the next vertex to be considered, and it will eventually become red.

Proof. We claim that in the whole process there can be at most 2^{t+1} deactivations incurring a cost from $[N/2^{t+1}, N/2^t)$. Assume otherwise, then there exists a vertex x charged twice by such deactivations. As a result of the first deactivation, the size of the connected component containing x drops from less than $N/2^t$ to below $N/2^{t+1}$. Consequently, during the next deactivation that charges x the cost must be smaller than $N/2^{t+1}$, a contradiction. As we have 2^d deactivation overall, the total cost can be at most:

$$\sum_{t=0}^{d} 2^{t+1} \cdot N/2^t = \mathcal{O}(N \cdot d)$$

as claimed. □

To complete the analysis, we observe that there are $N/2^d$ nodes of $\mathrm{DT}(G)$ such that we have 2^d deactivations affecting v. The sum of the second parts is thus:

$$\sum_{d=0}^{\log n} N/2^d \cdot n \cdot d < N^2 \sum_{d=0}^{\infty} d/2^d = \mathcal{O}(N^2).$$

Overall, the total update time is hence $\mathcal{O}(N^2)$. By Lemmas 46 and 47, it implies the following:

Theorem 28. *Given an incomplete matrix $\mathcal{A}_{n\times m}$, the* IDPP *problem can be solved in time $\mathcal{O}(nm)$.*

Part III

Appendices

Appendix A

Fundamental Definitions and Data Structures

A.1 Strings

An *alphabet* Σ of size $\sigma = |\Sigma|$ is a nonempty finite set of σ *letters*. An *integer* alphabet is an ordered alphabet whose letters are integers from 1 to σ. A *string* $T = T[1]T[2]\dots T[|T|]$ over Σ is a finite sequence of letters from Σ: $|T|$ denotes its length, ϵ denotes the *empty string*. For two positions i and j on T, we denote by $T[i\,..\,j] = T[i]\dots T[j]$ the *substring* of T that starts at position i and ends at position j. A *prefix* of T is a substring of the form $T[1\,..\,j]$, and a *suffix* of T is of the form $T[i\,..\,|T|]$. A *proper* prefix (suffix) of a string is not equal to the string itself. A string $X = X[1]X[2]\dots X[r]$ is a subsequence of a string T if X can be obtained by deleting zero or more letters from T. T^r denotes the reverse of T, that is, $T[|T|]T[|T|-1]\dots T[1]$. Σ^* denotes the set of all possible finite strings over Σ, while Σ^k is the set of all strings of length k over Σ.

Given two strings U and V, their concatenation $U \cdot V$ (also simply denoted by UV) is given by the letters of U followed by the letters of V: the concatenation of $k > 1$ copies of some string U is denoted by U^k, and we call $T = U^k$ a *power* of U.

Period. A *period* of a string X is any integer $p \in [1, |X|]$ such that $X[i] = X[i+p]$ for every $i = 1, 2, \dots, |X| - p$, and *the period*, denoted by $\mathrm{per}(X)$, is the smallest such p. We call a string X *periodic* if $\mathrm{per}(X) \le |X|/2$. The period $\mathrm{per}(X)$ can be computed in $\mathcal{O}(|X|)$ time for any string X [113].

Palindrome. A string P is a *palindrome* if and only if $P = P^R$. If factor $X[i\,..\,j]$, $1 \le i \le j \le |X|$, of a string X is a palindrome, then $\frac{i+j}{2}$ is the *center* of $X[i\,..\,j]$ in X and $\frac{j-i+1}{2}$ is the *radius* of $X[i\,..\,j]$. In other words, a string P is a palindrome if $P = YaY^R$ where Y is a string, Y^R is the reversal of Y and a is either a single letter (when the center is an integer) or the empty string (when it is not). Moreover, $X[i\,..\,j]$ is called a *palindromic factor* of X, and it is a *maximal palindrome* if there

is no other palindrome in X with center $\frac{i+j}{2}$ and larger radius. Hence X has exactly $2|X|-1$ maximal palindromes. We say that $X_1, X_2, \ldots, X_\ell$ is a (maximal) palindromic factorization of string a X if every X_i is a (maximal) palindrome, $X = X_1 X_2 \ldots X_\ell$, and ℓ is minimal. A maximal palindromic factorization of X can be computed in $\mathcal{O}(|X|)$ time (see, e.g., [23])

Alignment [259]. An alignment of $X = X[1] \ldots X[m] \in \Sigma^*$ and $Y = Y[1] \ldots Y[n] \in \Sigma^*$ is a pair of strings $U = U[1] \ldots U[h]$ and $V = V[1] \ldots V[h]$ of length $h \in [\max\{m, n\}, n+m]$ such that X is a subsequence of U, Y is a subsequence of V, U contains $h-m$ characters –, and V contains $h-n$ characters –, where – is a special character not appearing in X or Y. Multiple sequence alignment (MSA) is the generalization of pair-wise alignment to more than two strings.

Hamming distance. Given two strings U and V of equal length, the *Hamming distance* $d_H(U, V)$ is defined as the number of mismatching letters between U and V. The time complexity of computing the Hamming distance between two strings of length n is $\mathcal{O}(n)$ [184].

Edit distance. Given any two strings U and V, the *edit distance* $d_E(U, V)$ is defined as the minimum number of elementary edit operations (letter insertion, deletion, or substitution) to transform U to V. The edit distance d_E between two strings of length $\Omega(n)$ cannot be computed in $\mathcal{O}(n^{2-\delta})$ time without violating the Strong Exponential Time Hypothesis SETH [38], and hence the well-known quadratic-time solution of [245] for computing the edit distance between two strings of length $\mathcal{O}(n)$ is optimal up to subpolynomial factors. This is also true for weighted edit distance [80], where each operation (insertion, deletion, substitution and match) has a corresponding fixed non-negative cost (respectively $c_{\mathrm{i}}, c_{\mathrm{d}}, c_{\mathrm{s}}, c_{\mathrm{m}}$), and the following conditions hold: (i) $c_{\mathrm{i}} + c_{\mathrm{d}} > c_{\mathrm{m}}$, (ii) $c_{\mathrm{i}} + c_{\mathrm{d}} > c_{\mathrm{s}}$, and (iii) $c_{\mathrm{m}} \neq c_{\mathrm{s}}$.

Longest common extension. Given two strings X and Y and a pair (i, j), with $1 \leq i \leq |X|$ and $1 \leq j \leq |Y|$, the *longest common extension* at (i, j), denoted by $lce_{X,Y}(i, j)$, is the length of the longest substring of X starting at position i that matches a substring of Y starting at position j. We define $lce_{X,Y}(i, j) = 0$ when either $i \notin \{1, 2, \ldots, |X|\}$ or $j \notin \{1, 2, \ldots, |Y|\}$. For each index pair (i, j), $lce_{X,Y}(i, j)$ queries can be computed in constant time per query, after an $\mathcal{O}(|X| + |Y|)$ time and space pre-processing [184].

Regular expression [279]. The set of regular expressions over an alphabet Σ is defined recursively as follows: (I) $a \in \Sigma \cup \{\varepsilon\}$ is a regular expression. (II) If E and F are regular expressions, then so are EF, $E|F$, and E^*, where EF denotes the set of strings obtained by concatenating a string in E and a string in F, $E|F$ is the union of the strings in E and F, and E^* consists of all strings obtained by concatenating zero or more strings from E. Parentheses are used to override the natural precedence of the operators, which places the operator * highest, the concatenation next, and the operator $|$ last. A string T *matches* a regular expression E, if T is equal to one of the

strings in E. Given a string W and a regular expression E, the approximate regular expression matching problem can be solved in $\mathcal{O}(|W| \cdot |E|)$ time [279].

A.2 Graphs and Data Structures

A *tree* is an undirected connected graph $T = (V_T, E_T)$ without cycles: its degree-one vertices are called *leaves*, while the remaining vertices are called *internal nodes*. Trees T_1 and T_2 are is *isomorphic*, and we write $T_1 \equiv T_2$, if there is a bijective mapping $m : V_{T_1} \to V_{T_2}$ such that $(u, v) \in E_{T_1}$ iff $(m(u), m(v)) \in E_{T_2}$. Such a mapping is referred to as an *isomorphic mapping*, or an *isomorphism*. A *rooted* tree has a special node, called the *root*, that implicitly directs the edges, e.g., away from the root. We can hence define the *parent* and *child* relationships as follows: for $(u, v) \in E_T$ and u on the path from the root to v, we say that u is the parent of v, and write $p_T(v) = u$, and v is a child of u, and write $v \in \mathsf{children}(u)$). Note that we consider the children $\mathsf{children}(u)$ of some vertex u as a *set*, hence they are *unordered*. More generally, we say that a node u is an *ancestor* of a node v if u is on the path from the root to v, and conversely, v is a *descendant* of u. We extend the above notion of isomorphism to a pair T_1, T_2 of rooted trees by adding the condition that $m(p_{T_1}(u)) = p_{T_2}(m(u))$, implying in particular that m maps the root of T_1 to the root of T_2. Moreover, we denote by $\mathsf{level}_T(u)$ the *level* of u in T, that is, the number of edges on the path from the root to u (with the root itself being on level 0). A *forest* is a collection of trees.

Weighted ancestor. For a rooted tree T on n nodes with an integer weight $\mathcal{D}(v)$ assigned to every node u, such that the weight of the root is zero and $\mathcal{D}(u) < \mathcal{D}(v)$ if u is the parent of v, we say that a node v is a weighted ancestor of a node v at depth ℓ, denoted by $\mathrm{WA}_T(u, \ell)$, if v is the highest ancestor of u with weight of at least ℓ. Such queries can be answered in $\mathcal{O}(\log n)$ time after an $\mathcal{O}(n)$ time preprocessing [140].

Lowest common ancestor. For a rooted tree T, the lowest common ancestor $\mathrm{LCA}_T(u, v)$ is the lowest (i.e., farthest from the root) node that is an ancestor of both u and v. Such queries can be answered in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$ time preprocessing [47].

Heavy path decomposition. A *heavy path decomposition* of a tree T is obtained by selecting, for every non-leaf node $u \in T$, its *heavy child* v such that $T|v$ is the largest. This procedure decomposes the nodes of T into node-disjoint paths called *heavy paths*. Each heavy path p starts at some node, called its *head*, and ends at a leaf. An important property of such a decomposition is that the number of distinct heavy paths above any leaf (that is, intersecting the path from a leaf to the root) is only logarithmic in the size of T [328].

Bipartite maximum matching. A *bipartite graph* is an undirected graph $G = (A \cup B, E)$, such that there are no edges between vertices in A and there are no edges between vertices in B. A *matching* in a bipartite graph is a subset of edges with no two edges meeting at the same vertex. A maximum matching in an unweighted bipartite

graph is a matching of maximum cardinality, whereas a maximum weight matching in a weighted bipartite graph is a matching in which the sum of weights is maximised. Given an unweighted bipartite graph with m edges, the well-known algorithm by Hopcroft and Karp [197] finds a maximum matching in $\mathcal{O}(m^{1.5})$ time. This has been recently improved by Liu and Sidford to $\tilde{\mathcal{O}}(m^{4/3+o(1)})$ [254].

Trie. A *trie* is a tree in which every edge is labeled with a single letter, and every two edges outgoing from the same node have different labels. The label of a node u in such a tree T, denoted by $\mathcal{L}(u)$, is defined as the concatenation of the labels of all the edges on the path from the root of T to u. Thus, the label of the root of T is ε, and a trie is a representation of a set of strings consisting of the labels of all its leaves. By replacing each path p consisting of nodes with exactly one child by an edge labeled by the concatenation of the labels of the edges of p we obtain a *compact trie*. The nodes of the trie that are removed after this transformation are called *implicit*, while the remaining ones are referred to as *explicit*.

Suffix Tree. The *suffix tree* ST_X of a string X of length n over an alphabet Σ is a compact trie representing all the suffixes of $X\$$, where $\$ \notin \Sigma$ is a terminating letter used for technical purposes. ST_X has n leaves labelled from 1 to n: a leaf j corresponds to the suffix $X[j\,..\,n-1]$. Each internal node, other than the root, has at least two children, and each edge is labelled with a non-empty factor of $X\$$ encoded as an $[i, j]$ interval over $[0, n]$. No two edges out of a node can have labels beginning with the same letter. If u is a node of ST_X, then the string obtained by concatenating the edge labels along the path from the root to u is the *path-label* of u, denoted by $\mathcal{L}(u)$; the length of the path-label is the *string-depth* of node u. For any $i \in [1, n]$, the path-label of the leaf i is precisely the suffix $X[i\,..\,n-1]\$$. An additional set of edges, called *suffix links*, are extremely useful both for constructing ST_X efficiently and for performing a number of tasks on ST_X. Let the label $\mathcal{L}(u)$ of the path from the root to an explicit node u be aU, with $a \in \Sigma$: there is a suffix link from u to the unique explicit node v such that $\mathcal{L}(v) = U$ (such a node v always exists, as U occurs in X wherever aU occurs and u is an explicit node of ST_X). The suffix tree ST_X of a string X can be constructed in $\mathcal{O}(|X|)$ time for constant-sized and integer alphabets [139], or in $\mathcal{O}(|X| \log |X|)$ time if no assumptions on the size of the alphabet are made [313]. The suffix tree for a set of strings is called *generalized suffix tree*.

De Bruijn Graph. The *weighted de Bruijn graph* of order k over a string S of length n is a directed multigraph $G_{S,k} = (V_{S,k}, E_{S,k})$, where the set of vertices $V_{S,k}$ is the set of length-$(k-1)$ substrings of S and $E_{S,k}$ is the multiset of edges from vertex u to vertex v for every occurrence of u and v as consecutive length-$(k-1)$ substrings of S. More formally, there is a multi-edge $(u, v) \in E_{S,k}$ with multiplicity m if and only if $u[0] \cdot v = u \cdot v[k-2]$ and this string occurs in S exactly m times. Thus $G_{S,k}$ has exactly $n-k+1$ edges; in general, $G_{S,k}$ contains self-loops and multi-edges (inspect Fig. E.3 for an example).

Appendix B

Additional Experiments of Chapter 6

We show here an experimental comparison of the different versions of MP3, demonstrating the reason we decided to use MP3 $_{\sigma}$ as our default measure. We show in Figure B.1 (a) that MP3 $_{\sigma}$ combines the best aspect of MP3 $_{\cap}$ and MP3 $_{\cup}$ while MP3 $_{G}$ is, as expected, the average of the two. The same result can be seen in Figure B.1 (b), while Figure B.1 (c) display the effect of the sigmoid, where as the trees become less similar the value move towards the union.

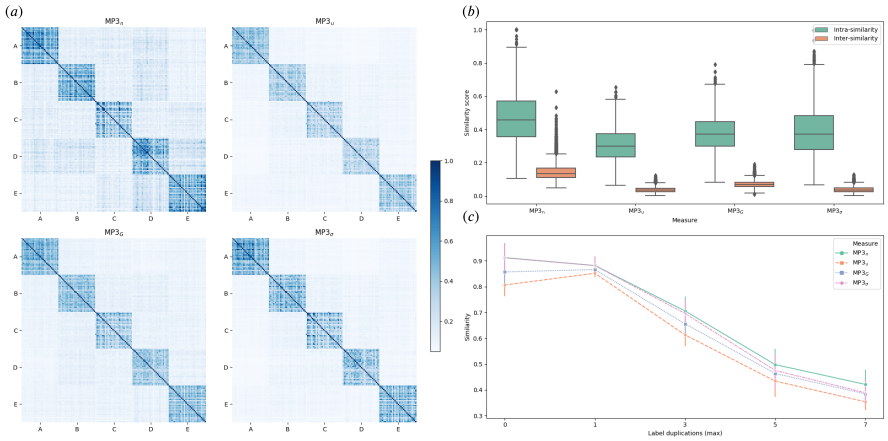


Figure B.1: (a) Heatmaps displaying the scores between all the 150 simulate trees from the second experimental setting. (b) Distribution of the similarities between the trees in the same class (Intra-similarity) and in different classes (Inter-similarity) for the 5 classes. (c) Effect of label duplication on the similarity scores. Similarities are the average of 15 trees generate from the same base with the specified value of maximum duplication from the previous experiment.

B.1 Effect of label sliding

We present a comparison of the measure in the case of a label sliding from left to right on the lowest level of a binary tree. The trees are compared against the first tree T_0. When the label slide to the same subtree no difference is found for all measure, as expected. On the other hand we show a higher decrease in similarity for `MP3` with respect to the other measure as the label slides. Interestingly for the last two trees CASet and DISC collapse to the same value showing no difference between the two. MLTD express very little difference between the last four trees.

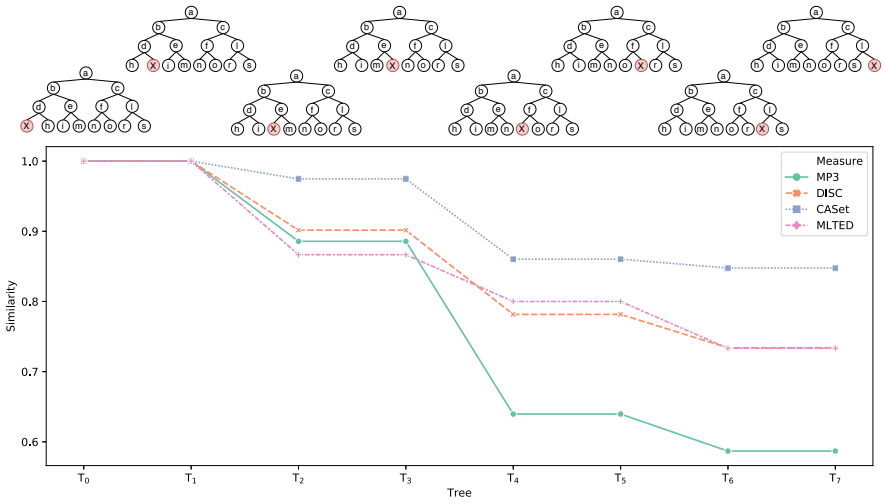


Figure B.2: Effect of a label sliding from left to right on the lowest level of a binary tree.

B.2 Base tree for poly-occurring label experiment

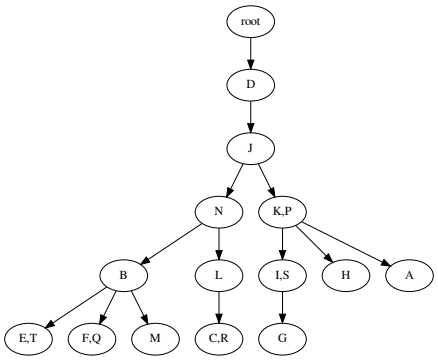


Figure B.3: Base tree used for evaluating the effect of poly-occurring labels on the similarity scores.

B.3 Base trees for Exp1 and Exp2

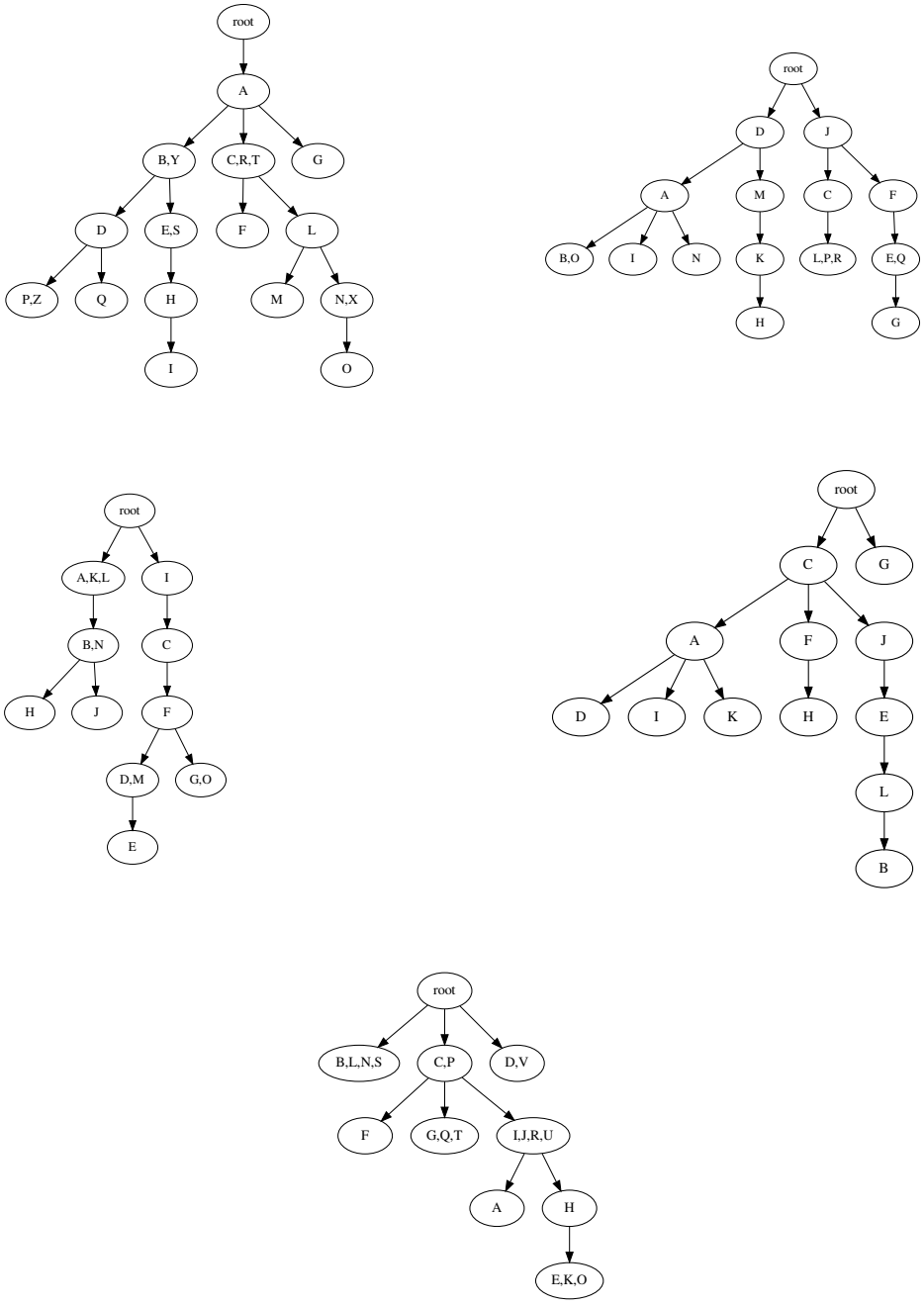


Figure B.4: Base trees used in Experiments 1 and 2.

B.4 Base trees for Clustering experiment

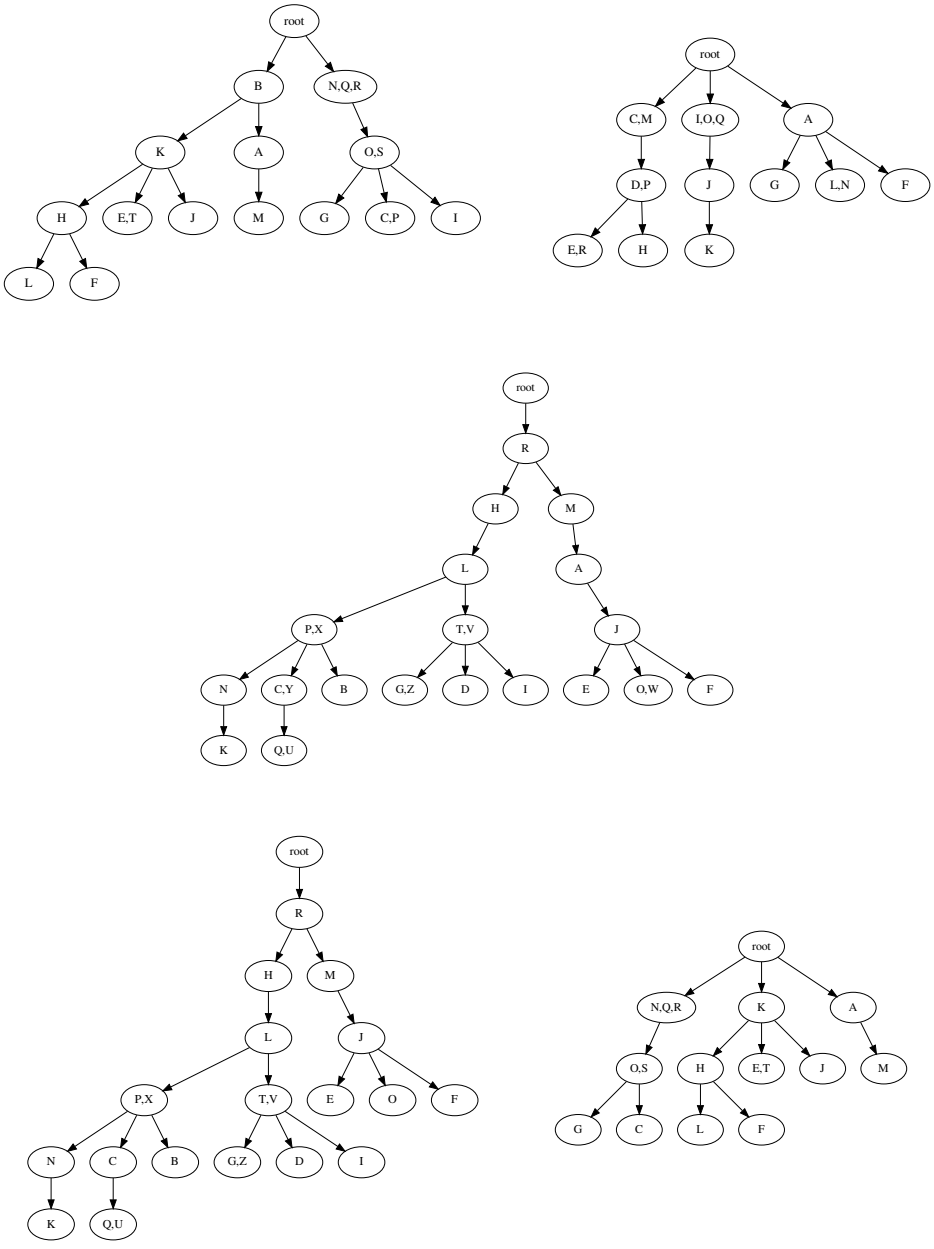


Figure B.5: Base trees used in the clustering experiment. Tree 4 is a perturbation of Tree 3 and Tree 5 is a perturbation of Tree 1.

B.5 Trees for real data experiment

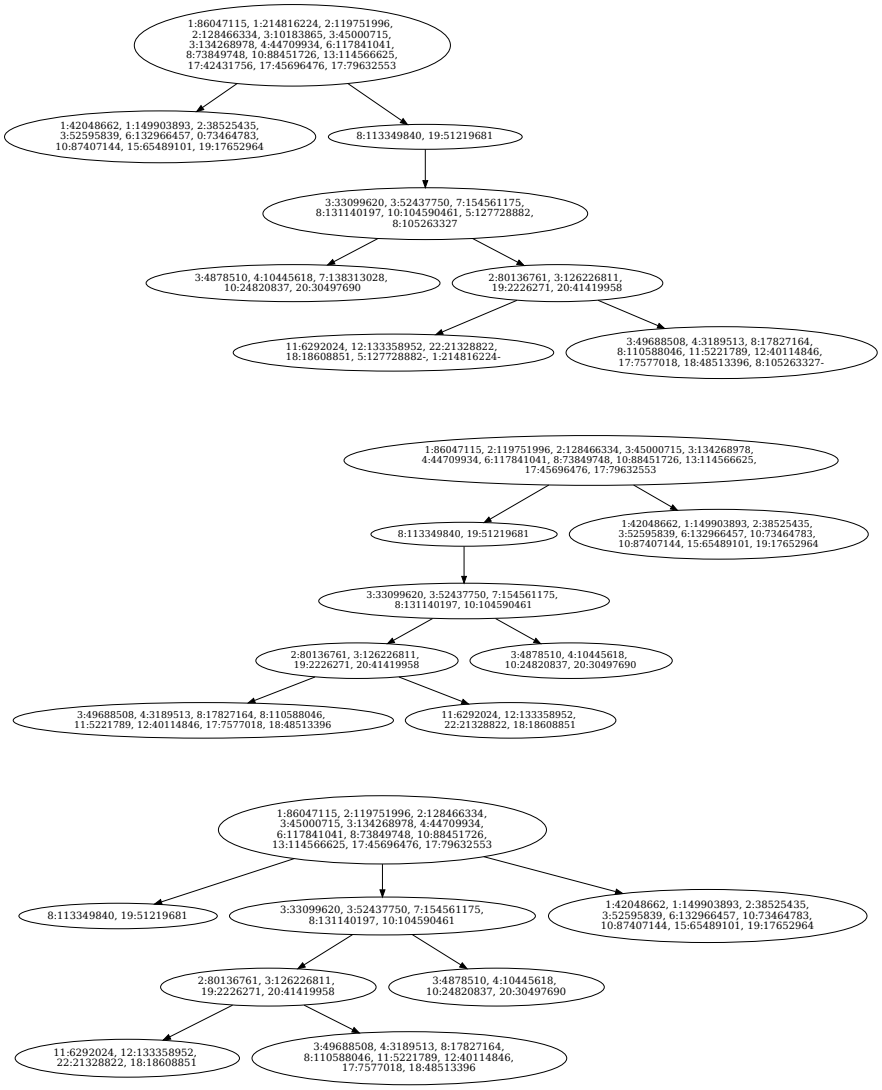


Figure B.6: Trees used in the experiment on real data from [168]. The upper tree is the base tree proposed in [168] for patient RMH002, the second one is the tree inferred by LICHeE, and the last one is the tree inferred by MIPUP (ipd parameter). We crafted the first tree starting from the supplementary material of the corresponding paper whereas we built the other two considering the trees reported in the MIPUP github repository.

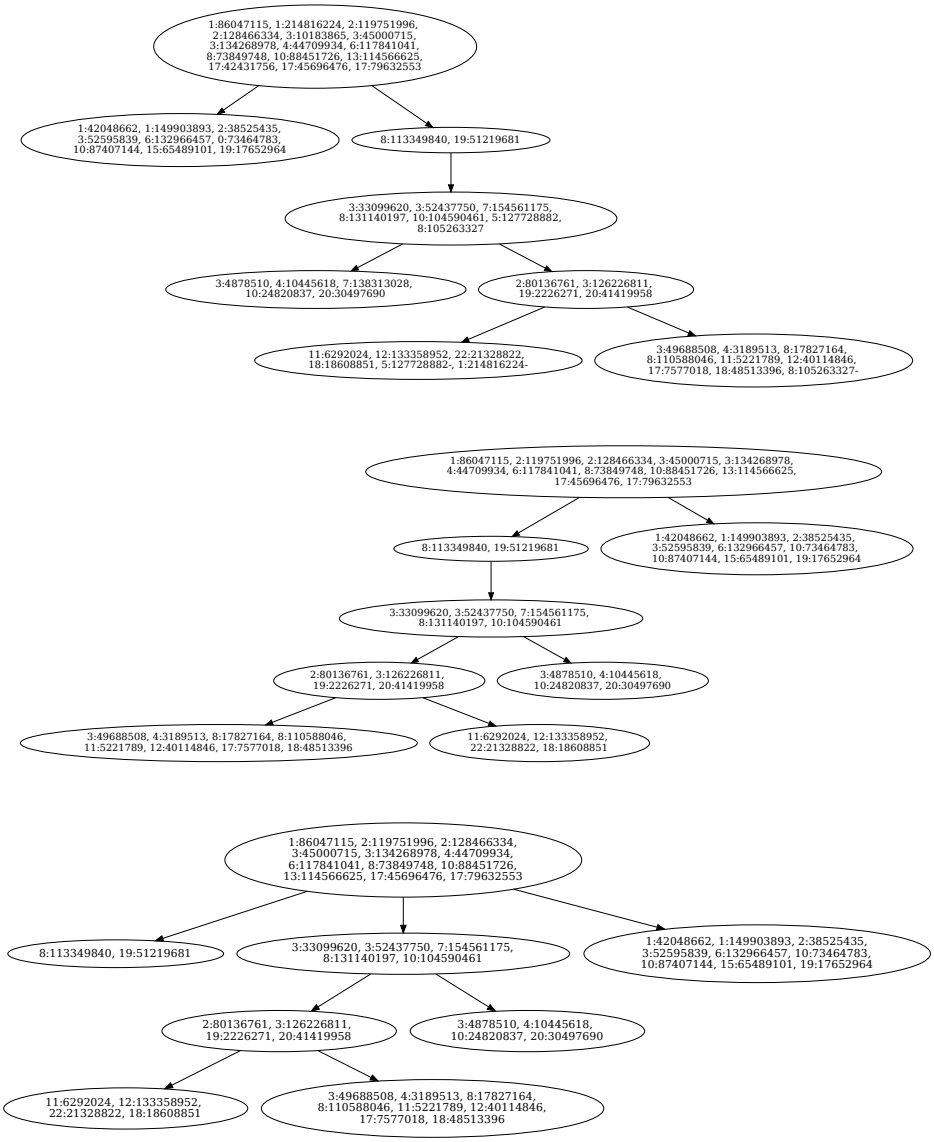


Figure B.7: Trees used in the experiment on real data from [134]. The upper tree is the base tree proposed in [134] for case SA501, the second one is the tree inferred by LICHeE, and the last one is the tree inferred by MIPUP. We obtained these trees from the supplementary material of [126]

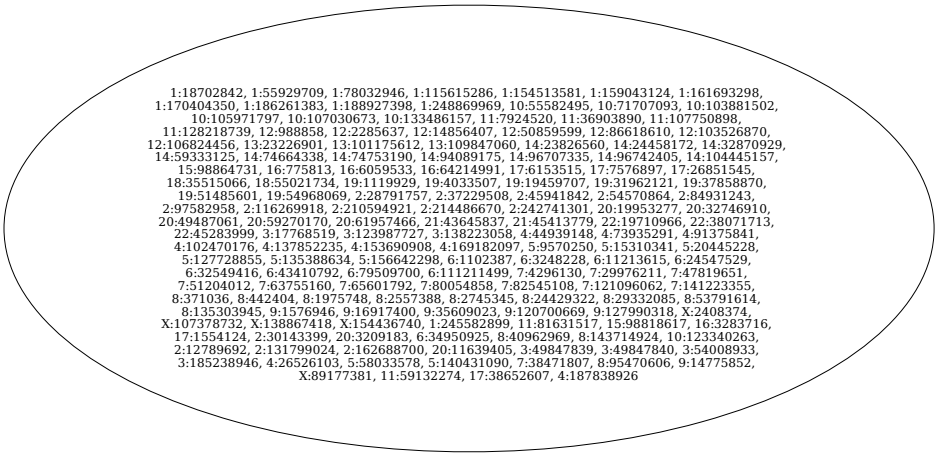


Figure B.8: Edge case tree used in the experiment on real data from [134]. We obtained such a tree by collapsing all nodes of the MIPUP's tree from Figure B.7 in a single node.

B.6 Example of computation of MP3

To better understand how to compute MP3, in this section we report a detailed example on trees with repeated labels.

In Figure B.9 we present two trees: the tree on the left, that we will refer to as Tree A, is composed of six nodes and does not include any repeated label, whereas the tree on the right, that we will refer to as Tree B, is composed of six nodes and only five distinct labels.

In Figure B.10 we report the 20 possible MTTs of Tree A and in Figure B.11 we report the 20 possible MTTs of Tree B.

Note that the MTTs of Tree A are unique, whereas the MTTs of Tree B are not. Indeed, since the label c appears twice, two MTTs of Tree B have the same labels (a, c, and f) and, by chance, the same shape (top right of Figure B.11). Notice that the same MTT is in the multiset of MTTs of Tree A (top right of Figure B.10) but appears only once.

In Figure B.12 we report the set of MTTs shared between Tree A and Tree B. This set is composed of seven MTTs and note that the MTT of the triplet (a, c, f) appears only once. This means that the MTT of that triplet will appear twice in $\mathtt{M}_B(a, c, f)$, once in the $\mathtt{M}_A(a, c, f) \sqcap \mathtt{M}_B(a, c, f)$, and once $M_B(a, c, f) \setminus M_A(a, c, f)$.

Overall, we have the following (we refer the reader to Section 2.2 of the main document for the notation):

- $|\lambda(A)| = 6$
- $|\lambda(B)| = 5$
- $|I| = 10$, i.e., the number of the triplets over the multiset $\{a, b, c, e, f\}$

- $|J| = 35$, i.e., the number of the triplets over the multiset $\{a, b, c, c, d, e, f\}$
- $\texttt{MP3}_{\cap} = 7/10 = 0.7$
- $\texttt{MP3}_{\cup} = 7/35 = 0.2$
- $\texttt{MP3}_{\sigma} = 0.2 + \sigma(0.7) \times \min\{0.7 - 0.2, 0.2\} \simeq 0.22384$

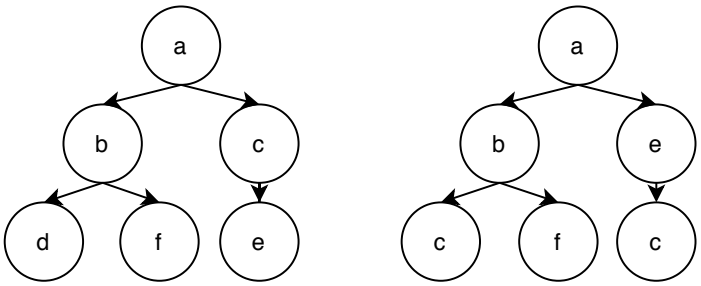


Figure B.9: Two tumor progression trees. Tree A (left) is composed of six nodes and six different labels, whereas Tree B (right) is composed of 6 nodes and five distinct labels.

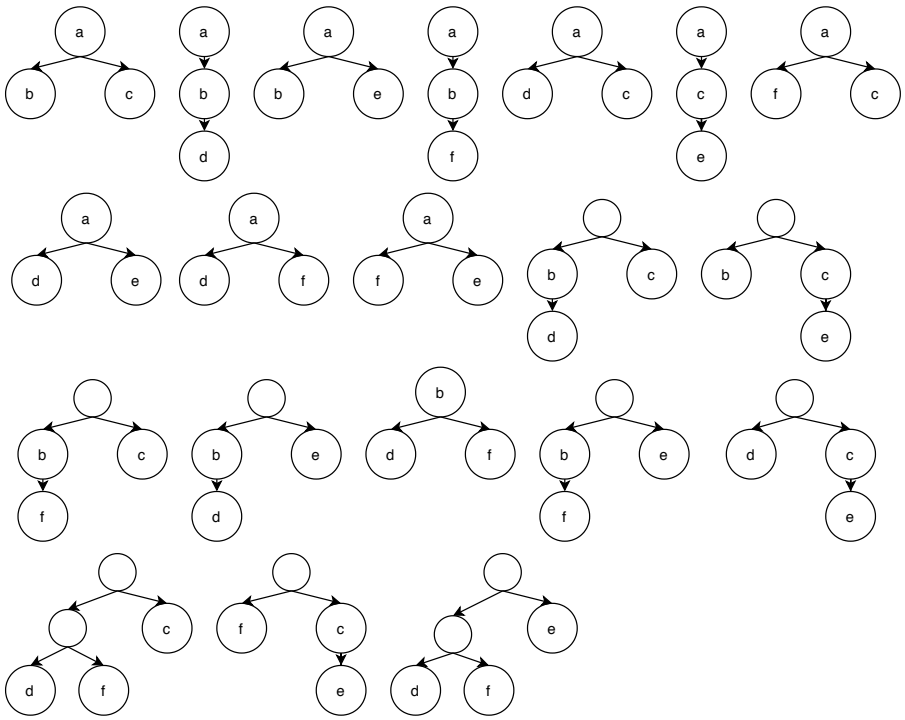


Figure B.10: Minimal tree topology of each triplet of nodes in Tree A.

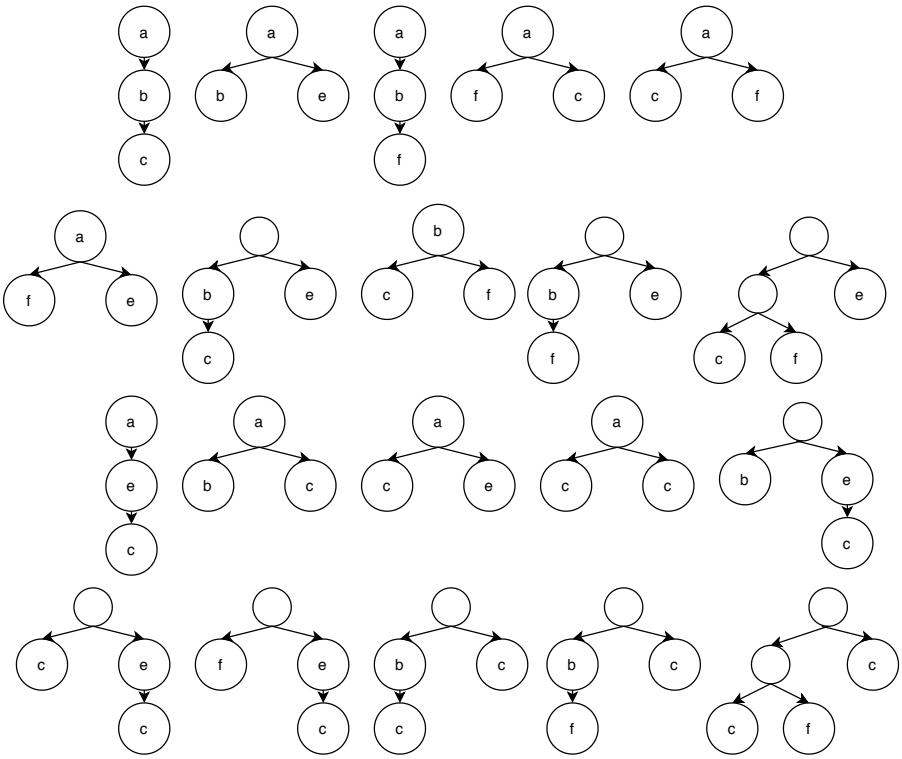


Figure B.11: Minimal tree topology of each triplet of nodes in Tree B.

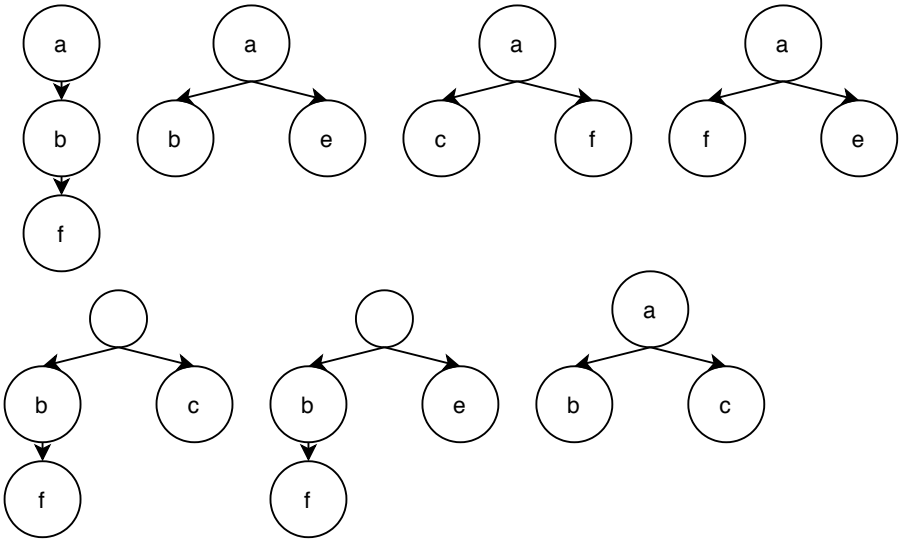


Figure B.12: Minimal tree topologies shared between Tree A and Tree B. Note that the MTT of the triplet (a, c, f) appears only once.

Appendix C

Combinatorial String Dissemination

Key Points

Problem. String data are often disseminated to support applications such as location-based service provision or DNA sequence analysis. This dissemination is likely to expose sensitive patterns that model confidential knowledge. We consider the problem of sanitizing a string by concealing the occurrences of sensitive patterns, while maintaining data utility.

Model. We seek for a string, to be disseminated in place of the original one, that preserves the order of appearance and frequency of all non-sensitive patterns, while sensitive patterns are concealed with the aid of an extra alphabet letter. We consider two settings. In the first one, we construct a minimal-length string that has the desired properties. In the second setting, we construct a string that is at minimal edit distance from the original string, in addition to preserving the order of appearance and frequency of all non-sensitive patterns. Since the extra alphabet letter may reveal the positions where some sensitive patterns originally were, we also tackle the problem of replacing such new letters with carefully selected letters of the original alphabet, so that sensitive patterns are not reinstated, implausible patterns are not introduced, and occurrences of spurious patterns are prevented.

Included Works

This chapter presents the paper **Combinatorial Algorithms for String Sanitization** [54], accepted for publication in *ACM Transactions on Knowledge Discovery from Data (TKDD)*. This paper is the journal extension of **String Sanitization: A Combinatorial Approach** [53], which has been presented at the *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases(ECML/PKDD, 2019)*.

C.1 Introduction

A large number of applications, in domains ranging from transportation to web analytics and bioinformatics feature data modeled as strings. For instance, a string may represent the history of visited locations of one or more individuals, with each letter corresponding to a location. Similarly, it may represent the history of search query terms of one or more web users, with letters corresponding to query terms, or a medically important part of the DNA sequence of a patient, with letters corresponding to DNA bases. Analyzing these data is key in applications including location-based service provision, product recommendation, and DNA sequence analysis. Therefore, such strings are often disseminated beyond the party that has collected them. For example, location-based service providers often outsource their data to data analytics companies who perform tasks such as similarity evaluation between strings [253], and retailers outsource their data to marketing agencies who perform tasks such as mining frequent patterns from the strings [256].

However, disseminating a string intact may result in the exposure of confidential knowledge, such as trips to mental health clinics in transportation data [346], query terms revealing political beliefs or sexual orientation of individuals in web data [284], or diseases associated with certain parts of DNA data [263]. Thus, it may be necessary to sanitize a string prior to its dissemination, so that confidential knowledge is not exposed. At the same time, it is important to preserve the utility of the sanitized string, so that data protection does not outweigh the benefits of disseminating the string to the party that disseminates or analyzes the string, or to the society at large. For example, a retailer should still be able to obtain actionable knowledge in the form of frequent patterns from the marketing agency who analyzed their outsourced data; and researchers should still be able to perform analyses such as identifying significant patterns in DNA sequences.

C.1.1 Model and Settings

Motivated by the discussion above, we introduce the following model which we call *Combinatorial String Dissemination* (CSD). In CSD, a party has a string W that it seeks to disseminate, while satisfying a set of *constraints* and a set of desirable *properties*. For instance, the constraints aim to capture privacy requirements and the properties aim to capture data utility considerations (e.g., posed by some other party based on applications). To satisfy both, W must be transformed to a string by applying a sequence of edit operations. The computational task is to determine this sequence of edit operations so that the transformed string satisfies the desirable properties subject to the constraints. Clearly, the constraints and the properties must be specified based on the application.

Under the CSD model, we consider two specific settings addressing practical considerations in common string processing applications; the *Minimal String Length* (MSL) setting, in which the goal is to produce a shortest string that satisfies the set of constraints and the set of desirable properties, and the *Minimal Edit Distance* (MED) setting, in which the goal is to produce a string that satisfies the set of constraints and the set of desirable properties and is at minimal edit distance from W. In the following,

we discuss each setting in more detail.

MSL Setting In this setting, the sanitized string X must satisfy the following constraint **C1**: for an integer $k > 0$, no given length-k substring (also called pattern) modeling confidential knowledge should occur in X. We call each such length-k substring a *sensitive pattern*. We aim at finding the shortest possible string X satisfying the following desired properties: (**P1**) the order of appearance of all other length-k substrings (*non-sensitive patterns*) is the same in W and in X; and (**P2**) the frequency of these length-k substrings is the same in W and in X. The problem of constructing X in this setting is referred to as TFS (Total order, Frequency, Sanitization). Note that it is straightforward to hide substrings of *arbitrary* lengths from X, by setting k equal to the length of the shortest substring we wish to hide, and then setting, for each of these substrings, any length-k substring as sensitive.

The MSL setting is motivated by real-world applications involving string dissemination. In these applications, a *data custodian* disseminates the sanitized version X of a string W to a *data recipient*, for the purpose of analysis (e.g., mining). W contains confidential information that the data custodian needs to hide, so that it does not occur in X. Such information is specified by the data custodian based on domain expertise, as in [13, 72, 172, 256]. At the same time, the data recipient specifies **P1** and **P2** that X must satisfy in order to be useful. These properties map directly to common data utility considerations in string analysis. By satisfying **P1**, X allows tasks based on the sequential nature of the string, such as blockwise q-gram distance computation [179], to be performed accurately. By satisfying **P2**, X allows computing the frequency of length-k substrings and hence mining frequent length-k substrings [297] with no utility loss. We require that X has minimal length so that it does not contain redundant information. For instance, the string which is constructed by concatenating all non-sensitive length-k substrings in W and separating them with a special letter that does not occur in W satisfies **P1** and **P2** but is not the shortest possible. Such a string X would have a negative impact on the efficiency of any subsequent analysis tasks to be performed on it.

MED Setting In this setting, the sanitized version X_{ED} of string W must satisfy the properties **P1** and **P2**, subject to the constraint **C1**, and also be at minimal edit distance from string W. Constructing such a string X_{ED} allows many tasks that are based on edit distance to be performed accurately. Examples of such tasks are frequent pattern mining [321], clustering [218], entity extraction [358] and range query answering [257], which are important in domains such as bioinformatics [321], text mining [358], and speech recognition [130].

Note, existing works for sequential data sanitization (e.g., [72, 172, 187, 256, 354]) or anonymization (e.g., [16, 73, 98]) cannot be applied to our settings (see Section C.2 for details).

C.1.2 Our Contributions

We define the TFS problem for string sanitization and a variant of it, referred to as PFS (Partial order, Frequency, Sanitization), which aims at producing an even

shorter string Y by relaxing **P1** of TFS. We also develop algorithms for TFS and PFS. Our algorithms construct strings X and Y using a separator letter #, which is not contained in the alphabet of W, ensuring that sensitive patterns do not occur in X or Y. The algorithms repeat proper substrings of sensitive patterns so that the frequency of non-sensitive patterns overlapping with sensitive ones does not change. For X, we give a deterministic construction which may be easily reversible (i.e., it may enable a data recipient to construct W from X), because the occurrences of # reveal the exact location of sensitive patterns. For Y, we give a construction which breaks several ties arbitrarily, thus being less easily reversible. We further address the reversibility issue by defining the MCSR (Minimum-Cost Separators Replacement) problem and designing an algorithm for dealing with it. In MCSR, we seek to replace all separators, so that the location of sensitive patterns is not revealed, while preserving data utility. In addition, we define the problem of constructing X_{ED} in the MED setting, which is referred to as ETFS (Edit-distance, Total order, Frequency, Sanitization), and design an algorithm framework to solve it.

Our work makes the following specific contributions:

1. We design an algorithm, TFS-ALGO, for solving the TFS problem in $\mathcal{O}(kn)$ time, where n is the length of W. In fact, we prove that $\mathcal{O}(kn)$ time is worst-case optimal by showing that the length of X is in $\Theta(kn)$ in the worst case. The output of TFS-ALGO is a string X consisting of a sequence of substrings over the alphabet of W separated by # (see Example 23 below). An important feature of our algorithm, which is useful in the efficient construction of Y discussed next, is that it can be implemented to produce an $\mathcal{O}(n)$-sized representation of X with respect to W in $\mathcal{O}(n)$ time. See Section C.4.

Example 23. *Let* $W =$ `aabaaacbcbbbaabbacaab`*,* $k = 4$*, and the set of sensitive patterns be* {`baaa`, `bbaa`}*. The string* $X =$ `aabaa#aaacbcbbba#baabbacaab` *consists of three substrings over the alphabet* {`a`, `b`, `c`} *separated by #. Note that no sensitive pattern occurs in* X*, while all non-sensitive substrings of length* $k = 4$ *have the same frequency in* W *and in* X *(e.g.,* `aaba` *appears once), and they appear in the same order in* W *and in* X *(e.g.,* `aaba` *precedes* `abaa`*). Also, note that any string shorter than* X *would either create sensitive patterns or change the frequencies of the non-sensitive ones (e.g., removing the last letter of* X *creates a string in which* `caab` *no longer appears).* □

2. We define the PFS problem relaxing **P1** of TFS to produce shorter strings that are more efficient to analyze. Instead of a *total order* (**P1**), we require a *partial order* (**Π1**) that preserves the order of appearance only for sequences of consecutive non-sensitive length-k substrings that overlap by $k-1$ letters. In other words, **Π1** requires preserving the order of appearance of any two non-sensitive length-k substrings U, V for which two conditions hold: (I) U and V occur consecutively in W, and (II) the length-$(k-1)$ suffix of U is the same as the length-$(k-1)$ prefix of V. This makes sense because the order of two consecutive non-sensitive length-k substrings with no length-$(k-1)$ overlap has anyway been "interrupted" (by one or more sensitive patterns). We exploit this observation to shorten the string further. Specifically, we design an algorithm that solves PFS in the optimal $\mathcal{O}(n + |Y|)$ time, where $|Y|$ is the length of Y, using the $\mathcal{O}(n)$-sized representation of X. See Section C.5.

Example 24. (Cont'd from Example 23) *Recall that* $W =$ `aabaaacbcbbbaabbacaab`*. A string* Y *is* `aaacbcbbba#aabaabbacaab`*. The order of* `aaba` *and* `abaa` *is preserved in*

Y as they are consecutive, non-sensitive, and the length-3 suffix of `aaba` *is the same as the length-3 prefix of* `abaa` *(i.e., they have an overlap of $k-1=3$ letters). The order of* `abaa` *and* `aaac`*, which are consecutive non-sensitive, is not preserved since they do not have an overlap of $k-1=3$ letters.* □

3. We define the MCSR problem, which seeks to produce a string Z, by deleting or replacing all separators in Y with letters from the alphabet of W so that: no sensitive patterns are reinstated in Z; occurrences of spurious patterns that may not be mined from W but can be mined from Z, at a given support threshold τ, are prevented; and the distortion incurred by the replacements in Z is bounded. The first requirement is to preserve privacy and the next two to preserve data utility. We show that MCSR is NP-hard and propose a heuristic to attack it. We also show how to apply the heuristic, so that letter replacements do not result in *implausible* patterns that may reveal the location of sensitive patterns. An implausible pattern is a string which is unlikely to occur in Z as a substring. For example, such a pattern may correspond to an impossible or unlikely trip in a sanitized movement dataset Z. When an occurrence of an implausible pattern is identified in Z, it becomes easier to identify the letter that replaced a # in the implausible pattern, and thus recover the sensitive pattern. To prevent this, we first define an implausible pattern as a statistically unexpected string. Our definition is based on a statistical significance measure computed over a reference dataset [77, 308, 25]. Specifically, an implausible pattern is a substring whose frequency in W is significantly smaller than its expected frequency in W. Then, we modify MCSR-ALGO, so that it does not replace any occurrence of # with letters that create implausible patterns. See Section C.6.

Example 25. (Cont'd from Example 24) *Recall that the output of* PFS *is $Y =$* `aaacbcbbba#aabaabbacaab`*. Let $\tau = 1$. A string $Z =$* `aaacbcbbbacaabaabbacaab` *is produced by replacing letter # with letter* `c`*. Note that Z contains no sensitive pattern, nor a non-sensitive pattern of length-4 substring that could not be mined from W at a support threshold τ (i.e., a pattern that does not occur in W). In addition, Z contains no implausible pattern, such as* `bbab`*, which is not expected to occur in W, according to an established statistical significance measure for strings [77, 308, 25].* □

4. We design an algorithm for solving the ETFS problem. The algorithm, called ETFS-ALGO, is based on a connection between ETFS and the approximate regular expression matching problem [279]. Given a string W and a regular expression E, the latter problem seeks to find a string T that matches E and is at minimal edit distance from W. ETFS-ALGO solves the ETFS problem in $\mathcal{O}(k|\Sigma|n^2)$ time, where $|\Sigma|$ is the size of the alphabet of W. See Section C.7.

Example 26. *Let $W =$* `aaaaaab`*, $k = 4$, and the set of sensitive patterns be $\{$*`aaaa`*,* `aaab`*$\}$.* TFS-ALGO *constructs string $X = \varepsilon$, where ε is the empty string, with $d_E(W, X) = 7$. On the contrary,* ETFS-ALGO *constructs string $X_{ED} =$* `aaa#aab` *with $d_E(W, X_{ED}) = 1 < 7$. Clearly, string X_{ED} is more suitable for applications based on measuring sequence similarity.* □

5. For the MSL setting, we implemented our combinatorial approach for sanitizing a string W (i.e., the aforementioned algorithms implementing the pipeline $W \to X \to$

$Y \rightarrow Z$) and show its effectiveness and efficiency on real and synthetic data. We also show that it possible to produce a string Z that does not contain implausible patterns, while incurring insignificant additional utility loss. The experiments are reported in Section C.8.

6. For the MED setting, we implemented ETFS-ALGO and experimentally compared it with TFS-ALGO. Interestingly, we demonstrate that TFS-ALGO constructs optimal or near-optimal solutions to the ETFS problem in practice. This is particularly encouraging because TFS-ALGO is linear in the length of the input string n, whereas ETFS-ALGO is quadratic in n. See Section C.8.

C.2 Related Work

We review related work in data sanitization (*a.k.a. knowledge hiding*) and data anonymization, two of the main topics in the area of privacy-preserving data mining [17, 68]. Data sanitization aims at concealing confidential knowledge, so that it is not easily discovered by mining a disseminated dataset [351, 13, 172]. For example, data sanitization may be used by a business to prevent a recipient of a dataset from inferring that a specific set of products (e.g., baking powder and flour) is purchased by many customers of the business [351]. This set of products needs to be concealed, as it provides competitive advantage to the business.

On the other hand, data anonymization [16, 275, 133] aims at preventing a data recipient from inferring information about individuals whose information is contained in the input dataset [153]. This includes inferences about the identity of an individual (identity disclosure), about whether or not an individual's information is contained in the output dataset (membership disclosure), as well as inferences that generally depend on an individual's information (inferential disclosure). For example, data anonymization works are used to prevent a data recipient from inferring the identity of an individual based on the products purchased by the individual, or from inferring that the individual has purchased a sensitive product (e.g., a medicine revealing their health condition) [369].

C.2.1 Data Sanitization

Existing data sanitization approaches can be classified, based on the type of data they are applied to, into those applied to a collection of records and others applied to a single sequence.

We first discuss data sanitization approaches that are applied to a collection of records. A record can be a set of values (itemset) [351, 337, 285], a trajectory [13], or a sequence [13, 172, 187]. In set-valued (transaction) datasets, the confidential knowledge to be hidden is typically modeled as a set of itemsets [337], association rules [351], or classification rules [285]. In trajectory datasets, the confidential knowledge is modeled as a set of subtrajectories [13]. Last, in sequential datasets, the confidential knowledge is modeled as a set of sequential patterns occurring in the dataset [13, 172, 187].

In what follows, we review three data sanitization approaches [13, 172, 187], which are applied to a collection of sequences, since they are the most relevant to our work.

The key difference of these approaches from our work is that they aim to hide sensitive patterns occurring as *subsequences* (not only as substrings) in the input *collection* (not in a single, long string). Moreover, they aim to hide sensitive patterns when these are *sufficiently frequent*; i.e., when a sensitive pattern occurs as a subsequence of least τ records, where τ is a given minimum frequency threshold. The hiding of a sensitive pattern is then performed by modifying some of the records in the collection (e.g., by letter deletion [13]), so that fewer than τ records contain the sensitive pattern as a subsequence. In our work, **C1** implies that no occurrence of a sensitive pattern exists in the sanitized sequence.

The problem of sanitizing a collection of sequences was first proposed by Abul et al. [13]. The authors developed a heuristic that applies deletion of letters contained in sensitive patterns. The heuristic aims to minimize the number of deleted letters in the collection. However, it does not focus on minimizing changes to the set of non-sensitive frequent sequential patterns that are incurred by deletions. In response, Gkoulalas-Divanis et al. [172] developed a heuristic that avoids such changes, hence improving data utility for frequent sequential pattern mining and tasks based on it. The heuristic of [172] first selects a sufficiently large subset of records to sanitize, favoring records that can be sanitized with few deletions. Then, it sanitizes each selected record by constructing a graph that represents the matchings between the record and sensitive patterns, and searching for graph nodes corresponding to good letters to delete. However, due to the fact that graph search is computationally inefficient, the heuristic searches only a small part of the graph.

Gwadera et al. [187] proposed a heuristic, called Permutation Hiding (PH). PH addresses the limitation of [13], as it aims to minimize changes to the set of non-sensitive frequent sequential patterns. Also, it addresses the limitation of [172], as it avoids the expensive graph search. Furthermore, PH employs both letter permutation and deletion to hide sensitive patterns. Permuting the letters of a sensitive pattern hides the pattern but may change the set of non-sensitive frequent sequential patterns. Thus, PH explores the space of possible permutations of the letters of a sensitive pattern to find a permutation that minimizes the number of such changes. When this is not possible, PH resorts to letter deletion.

Thus, in summary, our approach differs from existing approaches that are applied to a collection of sequences [13, 172, 187], in terms of: (I) input dataset (a collection of strings vs. a single string); (II) occurrences of a sensitive pattern that must be hidden (occurrences as a subsequence vs. occurrences as a substring); (III) data modification strategy (deletion and/or permutation vs. copying of non-sensitive substrings and letter replacement); (IV) utility considerations (no guarantees on minimizing changes to non-confidential frequent sequential patterns vs. guarantees on utility properties). Although these data sanitization methods were designed for the general case of a collection of sequences, they could in principle be applied to a single string. Through the following example, we illustrate this point and also highlight the difference with respect to the goals of our methods.

Example 27. *Let* $W =$ `aabaaacbcbbbaabbacaab`*,* $k = 4$*, and the set of sensitive patterns be* $\{$`baaa`, `bbaa`$\}$*. Consider applying the* PH *heuristic [187] using a minimum frequency threshold* $\tau = 1$*.* PH *constructs a string* $I =$ `aaba**cbcbbb**bb*ca*b`*, deleting six letters of* W *that are represented by the special letter* `*` *for the sake of clarity.*

PH *also creates non-sensitive length-k substrings that can be mined from* W *but cannot be mined from* Z *at frequency threshold* τ *(e.g.,* `abaa`*), as well as non-sensitive length-k substrings that cannot be mined from* W *but can be mined from* Z *at frequency threshold* τ *(e.g.,* `bacb`*). These substrings are referred to as* τ-lost *and* τ-ghost *patterns, respectively. Specifically, as shown in Table C.1,* PH *created* 11 τ*-lost and* 6 τ*-ghost patterns. On the other hand, applying our approach (i.e., the pipeline* TFS-ALGO→ PFS-ALGO→ MCSR-ALGO*) with* $\tau = 1$ *produces a string* $Z =$ `aaacbcbbbacaabaabbacaab` *with neither* τ*-lost nor* τ*-ghost patterns, as mentioned in Example 25. The reader can perhaps share the intuition that string* Z *is more useful than string* I*, as* Z *preserves the set of non-sensitive frequent sequential patterns that can be mined at* $\tau = 1$*.*

The main reason PH *incurs substantially more* τ*-lost and* τ*-ghost patterns than our method is because it hides the sensitive patterns when they occur as subsequences of the input string. That is, it hides all occurrences of each sensitive pattern in the string, albeit only occurrences comprised of consecutive letters (i.e., substrings) need to be hidden in our setting. For instance, two occurrences of the letter* `a` *have been deleted from the suffix* `bbacaab` *of* W *to prevent the sensitive pattern* `bbaa` *from occurring as a subsequence (the subsequence is comprised of the underlined letters in* W*). Note, however, that pattern* `bbaa` *does not occur as a substring in this suffix of* W*.* □

	τ-lost	τ-ghost
PH [187]	{`abaa`, `aaac`, `aacb`, `bbba`, `baab`, `aabb`, `abba`, `bbac`, `baca`, `acaa`, `caab`}	{`abac`, `bacb`, `bbbb`, `bbbc`,`bbca`, `bcab`}
Our method	∅	∅

Table C.1: The τ-lost and τ-ghost patterns, for $\tau = 1$, created by applying the PH heuristic [187] and our method on the string of Example 27.

In what follows, we review three data sanitization approaches [256, 73, 354], which are applied to a single sequence.

The work of Loukides et al. [256] is applied to a single event-sequence, in which each event is a multi-set of letters associated with a timestamp. Their work aims to hide sensitive patterns comprised of *a single letter*. Each such pattern is considered hidden when its relative frequency in any prefix of the event-sequence is sufficiently low. The hiding is performed by a dynamic-programming algorithm that applies letter deletion, while preserving the distribution of events across the sequence. The approach of [256] cannot be readily extended to hide sensitive patterns of length $k > 1$, which is our privacy objective. Moreover, it has a different utility criterion than our work, and it does not guarantee the satisfaction of the utility properties we consider here.

The work of Bonomi et al. [72] is applied to a single sequence and aims to prevent an attacker, who has background knowledge about the frequency distribution of sensitive patterns in the input sequence, from gaining additional knowledge about the frequency distribution of sensitive patterns by observing the sanitized sequence. This is performed by limiting the mutual information between the frequency distribution of sensitive patterns in the original and sanitized sequence. In other words, sensitive patterns are protected when their frequencies are similar in the input and in the sanitized sequence.

On the other hand, in our work, we consider a setting where sensitive patterns are unknown to the attacker and aim to prevent the attacker from observing their presence in the sanitized sequence. The hiding of sensitive patterns in [72] is performed by heuristics which aim to apply a small amount of generalization [315]. Generalization replaces a letter with an aggregate letter that is not part of the sequence alphabet, thereby introducing uncertainty. Thus, the work of [72] aims to produce sanitized data with a low level of uncertainty and does not focus on guaranteeing the accuracy of mining frequent substrings comprised of the letters of the alphabet.

The work of Wang et al. [354] is applied to an event-sequence, in which each event is a single letter associated with a timestamp. Their work considers the problem of deleting events in a given sequence, so as to reduce the ability of an attacker to detect sensitive patterns, while maximizing the detection of non-sensitive patterns. A pattern is detected when it occurs as a subsequence within a specified time window of the sequence. To solve this problem, the approach of [354] deletes events from the sequence in order to maximize a weighted utility function expressed as a sum of terms. An occurrence of a non-sensitive (respectively, sensitive) pattern in the sequence contributes a positive (respectively, negative) term to this function. Thus, [354] considers protecting sensitive patterns that occur as subsequences rather than as substrings, and it aims to achieve a good balance between matching non-sensitive patterns and preventing the matching of sensitive patterns.

C.2.2 Data Anonymization

Data anonymization is a different direction in privacy-preserving data mining than data sanitization [17, 12]. Data anonymization has been the focus of many research works (see [17, 156] for surveys). This includes works for anonymizing string data [16, 15, 73, 98]. The works of Aggarwal and Yu [16, 15] aim to enforce k-anonymity [315] on a collection of strings. This is performed by first grouping strings, so that each group contains at least k similar strings, and then replacing the strings in each group with a carefully constructed synthetic string. The work of [73] aims to release differentially private [133] top-k frequent substrings from a collection of strings, where k denotes the number of frequent substrings required. This is performed by building a noisy summary data structure that represents the collection and then mining the top-k frequent substrings from the data structure. The work of [98] aims to release a differentially private collection of strings. This is performed by exploiting the variable-length n-gram model [266] and calibrating the noise needed to enforce differential privacy based on the model.

The aforementioned anonymization methods aim to prevent privacy threats other than eliminating sensitive substrings from a string to prevent their mining. The threats they are dealing with, following the terminology of [153], are: identity disclosure for [16, 15] and membership as well as inferential disclosure for [73, 98]. Thus, our work is related to anonymization approaches in that it shares the general objective of protecting string data with [16, 15] and that of protecting data while supporting string mining with the work of [73].

C.3 Preliminaries

In this section, we start with providing some preliminary definitions. Then, we define our problems and introduce our main results. A summary of the acronyms introduced in the chapter is in Table C.2.

Acronym	Meaning
CSD	Combinatorial String Dissemination model
MSL	Minimal String Length setting
MED	Minimal Edit Distance setting
TFS	Total order, Frequency, Sanitization problem
PFS	Partial order, Frequency, Sanitization problem
MCSR	Minimum-Cost Separators Replacement problem
ETFS	Edit-distance, Total order, Frequency, Sanitization problem
PH	Permutation Hiding heuristic [187]
MCK	Multiple Choice Knapsack problem [227]
FO-SSM	Fixed-Overlap Shortest String with Multiplicities problem
SCS	Shortest Common Superstring problem [160]

Table C.2: Acronyms used throughout

By $\mathrm{Freq}_V(U)$ we denote the number of occurrences of string U in string V. Given two strings U and V we say that U has a *suffix-prefix overlap* of length $\ell > 0$ with V if and only if the length-ℓ suffix of U is equal to the length-ℓ prefix of V, i.e., $U[|U|-\ell\,..\,|U|-1] = V[0\,..\,\ell-1]$.

We fix a string W of length n over an alphabet $\Sigma = \{1,\ldots,n^{\mathcal{O}(1)}\}$ and an integer $0 < k < n$. We refer to a length-k string or a *pattern* interchangeably. An occurrence of a pattern is uniquely represented by its starting position. Let $\mathcal{S}$ be a set of positions over $\{0,\ldots,n-k\}$ with the following closure property: for every $i \in \mathcal{S}$, if there exists j such that $W[j\,..\,j+k-1] = W[i\,..\,i+k-1]$, then $j \in \mathcal{S}$. That is, if an occurrence of a pattern is in $\mathcal{S}$, then all its occurrences are in $\mathcal{S}$. A substring $W[i\,..\,i+k-1]$ of W is called *sensitive* if and only if $i \in \mathcal{S}$. $\mathcal{S}$ is thus the set of occurrences of sensitive patterns. The difference set $\mathcal{I} = \{0,\ldots,n-k\} \setminus \mathcal{S}$ is the set of occurrences of *non-sensitive* patterns.

For any string U, we denote by $\mathcal{I}_U$ the set of occurrences of non-sensitive length-k strings over Σ in U (we have that $\mathcal{I}_W = \mathcal{I}$). We call an occurrence i the *t-predecessor* of another occurrence j in $\mathcal{I}_U$ if and only if i is the largest element in $\mathcal{I}_U$ that is less than j. This relation induces a *strict total order* on the occurrences in $\mathcal{I}_U$. We call i the *p-predecessor* of j in $\mathcal{I}_U$ if and only if i is the t-predecessor of j in $\mathcal{I}_U$ *and* $U[i\,..\,i+k-1]$ has a suffix-prefix overlap of length $k-1$ with $U[j\,..\,j+k-1]$. This relation induces a *strict partial order* on the occurrences in $\mathcal{I}_U$. We call a subset $\mathcal{J}$ of $\mathcal{I}_U$ a *t-chain* (resp., *p-chain*) if for all elements in $\mathcal{J}$ except the minimum one, their t-predecessor (resp., p-predecessor) is also in $\mathcal{J}$. For two strings U and V, chains $\mathcal{J}_U$ and $\mathcal{J}_V$ are *equivalent*, denoted by $\mathcal{J}_U \equiv \mathcal{J}_V$, if and only if $|\mathcal{J}_U| = |\mathcal{J}_V|$ and $U[u\,..\,u+k-1] = V[v\,..\,v+k-1]$, where u is the jth smallest element of $\mathcal{J}_U$ and v is the jth smallest of $\mathcal{J}_V$, for all $j \le |\mathcal{J}_U|$.

Problem Statements and Main Results We define the following problem for the MSL setting.

Problem 1 (TFS). *Given W, k, $\mathcal{S}$ and $\mathcal{I}_W$, construct the* shortest *string X:*

C1 *X does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_X$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_X$ are equivalent.*
P2 *$Freq_X(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i\,..\,i+k-1] : i \in \mathcal{S}\}$.*

TFS requires constructing the shortest string X in which all sensitive patterns from W are concealed (**C1**), while preserving the order (**P1**) and the frequency (**P2**) of all non-sensitive patterns. Our first result is the following.

Theorem 29. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. Given $k < n$ and $\mathcal{S}$,* TFS-ALGO *solves Problem 1 in $\mathcal{O}(kn)$ time, which is worst-case optimal. An $\mathcal{O}(n)$-sized representation of X can be built in $\mathcal{O}(n)$ time.*

P1 implies **P2**, but **P1** is a strong assumption that may result in long output strings that are inefficient to analyze. We thus relax **P1** to require that the order of appearance remains the same only for sequences of consecutive non-sensitive length-k substrings that also overlap by $k-1$ letters (p-chains). This leads to the following problem for the MSL setting.

Problem 2 (PFS). *Given W, k, $\mathcal{S}$, and $\mathcal{I}_W$ construct a* shortest *string Y:*

C1 *Y does not contain any sensitive pattern.*
Π1 *There exists an injective function f from the p-chains of $\mathcal{I}_W$ to the p-chains of $\mathcal{I}_Y$ such that $f(\mathcal{J}_W) \equiv \mathcal{J}_W$ for any p-chain $\mathcal{J}_W$ of $\mathcal{I}_W$.*
P2 *$Freq_Y(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i\,..\,i+k-1] : i \in \mathcal{S}\}$.*

Our second result, which builds on Theorem 29, is the following.

Theorem 30. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. Given $k < n$ and $\mathcal{S}$,* PFS-ALGO *solves Problem 2 in the optimal $\mathcal{O}(n + |Y|)$ time.*

To arrive at Theorems 29 and 30, we use a special letter (separator) $\# \notin \Sigma$ when required. However, the occurrences of # may reveal the locations of sensitive patterns. We thus seek to delete or replace the occurrences of # in Y with letters from Σ. The new string Z should not reinstate sensitive patterns or create implausible patterns. Given an integer threshold $\tau > 0$, we call a pattern $U \in \Sigma^k$ a *τ-ghost* in Z if and only if $\mathrm{Freq}_W(U) < \tau$ but $\mathrm{Freq}_Z(U) \geq \tau$. Moreover, we seek to prevent *τ-ghost occurrences* in Z by also bounding the total *weight* of the *letter choices* we make to replace the occurrences of #. This is the MCSR problem. We show that already a restricted version of the MCSR problem, namely, the version when $k = 1$, is NP-hard via the *Multiple Choice Knapsack* (MCK) problem [295].

Theorem 31. *The* MCSR *problem is NP-hard.*

Based on this connection, we propose a non-trivial heuristic algorithm to attack the MCSR problem for the general case of an arbitrary k.

We define the following problem for the MED setting.

Problem 3 (ETFS). *Given W, k, $\mathcal{S}$, and $\mathcal{I}$, construct a string X_{ED} which is at minimal edit distance from W and satisfies the following:*

C1 *X_{ED} does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_{X_{ED}}$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_{X_{ED}}$ are equivalent.*
P2 *$Freq_{X_{ED}}(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i\,..\,i+k-1] : i \in \mathcal{S}\}$.*

We show how to reduce any instance of the ETFS problem to some instance of the approximate regular expression matching problem. In particular, the latter instance consists of a string of length n (string W) and a regular expression E of length $\mathcal{O}(k|\Sigma|n)$. We thus prove the claim of Theorem 32 by employing the $\mathcal{O}(|W| \cdot |E|)$-time algorithm of [279].

Theorem 32. *Let W be a string of length n over an alphabet Σ. Given $k < n$ and $\mathcal{S}$,* ETFS-ALGO *solves Problem 3 in $\mathcal{O}(k|\Sigma|n^2)$ time.*

C.4 TFS-ALGO

We convert the input string W into a string X over alphabet $\Sigma \cup \{\#\}$, $\# \notin \Sigma$, by reading the letters of W, from left to right, and appending them to X while enforcing the following two rules:

R1: When the last letter of a sensitive substring U is read from W, we append $\#$ to X (essentially replacing this last letter of U with $\#$). Then, we append the succeeding non-sensitive substring (in the t-predecessor order) after $\#$.
R2: When the $k-1$ letters before $\#$ are the same as the $k-1$ letters after $\#$, we remove $\#$ and the $k-1$ succeeding letters (inspect Fig. C.1).

R1 prevents U from occurring in X, and **R2** reduces the length of X (i.e., allows to hide sensitive patterns with fewer extra letters). Both rules leave unchanged the order and frequencies of non-sensitive patterns. It is crucial to observe that applying the idea behind **R2** on more than $k-1$ letters would decrease the frequency of some pattern, while applying it on fewer than $k-1$ letters would create new patterns. Thus, we need to consider just **R2** *as-is*.

$$W = \texttt{aabaaaababbbaab}$$
$$\tilde{X} = \texttt{aabaaa\#aaaba\#babb\#bbbaab}$$
$$X = \texttt{aabaaaba\#babb\#bbbaab}$$

Figure C.1: Sensitive patterns are underlined in red; non-sensitive patterns are overlined in blue; $\tilde{X}$ is obtained by applying **R1**; and X by applying **R1** and **R2**. In green we highlight an overlap of $k-1=3$ letters.

Let C be an array of size n that stores the occurrences of sensitive and non-sensitive patterns: $C[i] = 1$ if $i \in \mathcal{S}$ and $C[i] = 0$ if $i \in \mathcal{I}$. For technical reasons we set the last $k-1$ values in C equal to $C[n-k]$; i.e., $C[n-k+1] := \ldots := C[n-1] := C[n-k]$. Note that C is constructible from $\mathcal{S}$ in $\mathcal{O}(n)$ time. Given C and $k < n$, TFS-ALGO efficiently

constructs X by implementing **R1** and **R2** concurrently as opposed to implementing **R1** and then **R2** (see the proof of Lemma 51 for details of the workings of TFS-ALGO and Fig. C.1 for an example). We next show that string X enjoys several properties.

Lemma 51. *Let W be a string of length n over Σ. Given $k < n$ and array C,* TFS-ALGO *constructs the shortest string X such that the following hold:*

*(I) There exists no $W[i\,..\,i+k-1]$ with $C[i] = 1$ occurring in X (**C1**).*

*(II) $\mathcal{I}_W \equiv \mathcal{I}_X$, i.e., the order of substrings $W[i\,..\,i+k-1]$, for all i such that $C[i] = 0$, is the same in W and in X; conversely, the order of all substrings $U \in \Sigma^k$ of X is the same in X and in W (**P1**).*

*(III) $Freq_X(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i\,..\,i+k-1] : C[i] = 1\}$ (**P2**).*

*(IV) The occurrences of letter $\#$ in X are at most $\lfloor \frac{n-k+1}{2} \rfloor$ and they are at least k positions apart (**P3**).*

*(V) $0 \le |X| \le \lceil \frac{n-k+1}{2} \rceil \cdot k + \lfloor \frac{n-k+1}{2} \rfloor$ and these bounds are tight (**P4**).*

TFS-ALGO$(W \in \Sigma^n, C, k, \# \notin \Sigma)$

$X \leftarrow \varepsilon$; $j \leftarrow |W|$; $\ell \leftarrow 0$;
$j \leftarrow \min\{i | C[i] = 0\}$; /* j is the leftmost pos of a non-sens. pattern */
if $j + k - 1 < |W|$ **then** /* Append the first non-sens. pattern to X */
  $X[0\,..\,k-1] \leftarrow W[j\,..\,j+k-1]$; $j \leftarrow j + k$; $\ell \leftarrow \ell + k$;
while $j < |W|$ **do** /* Examine two consecutive patterns */
  $p \leftarrow j - k$; $c \leftarrow p + 1$;
  if $C[p] = C[c] = 0$ **then** /* If both are non-sens., append the last letter of the rightmost one to X */
    $X[\ell] \leftarrow W[j]$; $\ell \leftarrow \ell + 1$; $j \leftarrow j + 1$;
  if $C[p] = 0 \wedge C[c] = 1$ **then** /* If the rightmost is sens., mark it and advance j */
    $f \leftarrow c$; $j \leftarrow j + 1$;
  if $C[p] = C[c] = 1$ **then** $j \leftarrow j + 1$; /* If both are sens., advance j */
  if $C[p] = 1 \wedge C[c] = 0$ **then** /* If the leftmost is sens. and the rightmost is not */
    if $W[c\,..\,c+k-2] = W[f\,..\,f+k-2]$ **then** /* If the last marked sens. pattern and the current non-sens. overlap by $k-1$, append the last letter of the latter to X */
      $X[\ell] \leftarrow W[j]$; $\ell \leftarrow \ell + 1$; $j \leftarrow j + 1$;
    else /* Else append $\#$ and the current non-sens. pattern to X */
      $X[\ell] \leftarrow \#$; $\ell \leftarrow \ell + 1$;
      $X[\ell\,..\,\ell+k-1] \leftarrow W[j-k+1\,..\,j]$; $\ell \leftarrow \ell + k$; $j \leftarrow j + 1$;
report X

Proof. **C1**: Index j in TFS-ALGO runs over the positions of string W; at any moment it indicates the ending position of the currently considered length-k substring of W. When $C[j-k+1]=1$ (Lines 18-22 of the pseudocode) TFS-ALGO never appends $W[j]$, i.e., the last letter of a sensitive length-k substring, implying that, by construction of C, no $W[i\,..\,i+k-1]$ with $C[i]=1$ occurs in X.

P1: When $C[j-k]=C[j-k+1]=0$ (Lines 14-16) TFS-ALGO appends $W[j]$ to X, thus the order of $W[j-k\,..\,j-1]$ and $W[j-k+1\,..\,j]$ is clearly preserved. When $C[j-k]=0$ and $C[j-k+1]=1$, index f stores the starting position on W of the $(k-1)$-length suffix of the last non-sensitive substring appended to X (see also Fig. C.1). **C1** ensures that no sensitive substring is added to X in this case, nor when $C[j-k]=C[j-k+1]=1$. The next letter will thus be appended to X when $C[j-k]=1$ and $C[j-k+1]=0$ (Lines 24-34). The condition on Line 26 is satisfied if and only if the last non-sensitive length-k substring appended to X overlaps with the immediately succeeding non-sensitive one by $k-1$ letters: in this case, the last letter of the latter is appended to X by Line 28, clearly maintaining the order of the two. Otherwise, Line 34 will append $W[j-k+1\,..\,j]$ to X, once again maintaining the length-k substrings' order. Conversely, by construction, any $U\in\Sigma^k$ occurs in X only if it equals a length-k non-sensitive substring of W. The only occasion when a letter from W is appended to X more then once is when Line 34 is executed: it is easy to see that in this case, because of the occurrence of #, each of the $k-1$ repeated letters creates exactly one $U\notin\Sigma^k$, without introducing any new length-k string over Σ nor increasing the occurrences of a previous one. Finally, Line 28 does not introduce any new $U\in\Sigma^k$ except for the one present in W, nor any extra occurrence of the latter, because it is only executed when two consecutive non-sensitive length-k substrings of W overlap exactly by $k-1$ letters.

P2: It follows from the proof for **C1** and **P1**.

P3: Letter # is added only by Line 32, which is executed only when $C[j-k]=1$ and $C[j-k+1]=0$. This can be the case up to $\lceil\frac{n-k+1}{2}\rceil$ times as array C can have alternate values only in the first $n-k+1$ positions. By construction, X cannot start with # (Lines 4-8), and thus the maximal number of occurrences of # is $\lfloor\frac{n-k+1}{2}\rfloor$. By construction, letter # in X is followed by at least k letters (Line 34): the leftmost non-sensitive substring following a sequence of one or more occurrences of sensitive substrings in W.

P4: *Upper bound.* TFS-ALGO increases the length of string X by more than one letter only when letter # is added to X (Line 32). Every time Lines 32-34 are executed, the length of X increases by $k+1$ letters. Thus the length of X is maximized when the maximal number of occurrences of # is attained. This length is thus bounded by $\lceil\frac{n-k+1}{2}\rceil\cdot k+\lfloor\frac{n-k+1}{2}\rfloor$.

Tightness. For the lower bound, let $W=a^n$ and a^k be sensitive. The condition at Line 6 is not satisfied because no element in C is set to 0: $j=n$. Then the condition on Line 10 is also not satisfied because $j=n$, and thus TFS-ALGO outputs the empty string. A *de Bruijn sequence* of order k over an alphabet Σ is a string in which every possible length-k string over Σ occurs exactly once as a substring. For the upper bound, let W be the order-$(k-1)$ de Bruijn sequence over alphabet Σ, $n-k$ be even, and $\mathcal{S}=\{1,3,5,\ldots,n-k-1\}$. $C[0]=0$ and so Line 8 will add the first k letters of W to X. Then observe that $C[1]=1, C[2]=0; C[3]=1, C[4]=0,\ldots$, and so on; this

sequence of values corresponds to satisfying Lines 24 and 18 alternately. Line 18 does not add any letter to X. The *if* statement on Line 26 will always fail because of the de Bruijn sequence property. We thus have a sequence of the non-sensitive length-k substrings of W interleaved by occurrences of # appended to X. TFS-ALGO thus outputs a string of length $\lceil \frac{n-k+1}{2} \rceil \cdot k + \lfloor \frac{n-k+1}{2} \rfloor$ (see Example 28).

We finally prove that X has minimal length. Let X_j be the prefix of string X obtained by processing $W[0\,..\,j]$. Let $j_{\min} = \min\{i | C[i] = 0\} + k - 1$. We will proceed by induction on j, claiming that X_j is the shortest string such that **C1** and **P1**-**P4** hold for $W[0\,..\,j]$, $\forall\, j_{\min} \leq j \leq |W| - 1$. We call such a string *optimal*.

Base case: $j = j_{\min}$. By Lines 6-8 of TFS-ALGO, X_j is equal to the first non-sensitive length-k substring of W, and it is clearly the shortest string such that **C1** and **P1**-**P4** hold for $W[0\,..\,j]$.

Inductive hypothesis and step: X_{j-1} is optimal for $j > j_{\min}$. If $C[j-k] = C[j-k+1] = 0$, $X_j = X_{j-1}W[j]$ and this is clearly optimal. If $C[j-k+1] = 1$, $X_j = X_{j-1}$ thus still optimal. Finally, if $C[j-k] = 1$ and $C[j-k+1] = 0$ we have two subcases: if $W[f\,..\,f+k-2] = W[j-k+1\,..\,j-1]$ then $X_j = X_{j-1}W[j]$, and once again X_j is evidently optimal. Otherwise, $X_j = X_{j-1}\#W[j-k+1\,..\,j]$. Suppose by contradiction that there exists a shorter X'_j such that **C1** and **P1**-**P4** still hold: either drop # or append less than k letters after #. If we appended less than k letters after #, since TFS-ALGO will not read $W[j]$ ever again, **P2**-**P3** would be violated, as an occurrence of $W[j-k+1\,..\,j]$ would be missed. Without #, the last k letters of $X_{j-1}W[j-k+1]$ would violate either **C1** or **P1** and **P2** (since we suppose $W[f\,..\,f+k-2] \neq W[j-k+1\,..\,j-1]$). Then X_j is optimal. □

Example 28 (Illustration of **P3**). *Let* $k = 4$. *We construct the order-*3 *de Bruijn sequence* $W = \texttt{baaabbbaba}$ *of length* $n = 10$ *over alphabet* $\Sigma = \{\texttt{a}, \texttt{b}\}$, *and choose* $\mathcal{S} = \{1, 3, 5\}$. TFS-ALGO *constructs:*

$$X = \texttt{baaa\#aabb\#bbba\#baba}.$$

The upper bound of $\lceil \frac{n-k+1}{2} \rceil \cdot k + \lfloor \frac{n-k+1}{2} \rfloor = 19$ *on the length of* X *is attained.* □

Let us now show the main result of this section.

Theorem 29. *Let* W *be a string of length* n *over* $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. *Given* $k < n$ *and* $\mathcal{S}$, TFS-ALGO *solves Problem 1 in* $\mathcal{O}(kn)$ *time, which is worst-case optimal. An* $\mathcal{O}(n)$*-sized representation of* X *can be built in* $\mathcal{O}(n)$ *time.*

Proof. For the first part inspect TFS-ALGO. Lines 4-8 can be realized in $\mathcal{O}(n)$ time. The *while* loop in Line 10 is executed no more than n times, and every operation inside the loop takes $\mathcal{O}(1)$ time except for Line 26 and Line 34 which take $\mathcal{O}(k)$ time. Correctness and optimality follow directly from Lemma 51 (**P4**).

For the second part, we assume that X is represented by W and a sequence of pointers $[i, j]$ to W interleaved (if necessary) by occurrences of #. In Line 34, we can use an interval $[i, j]$ to represent the length-k substring of W added to X. In all other lines (Lines 8, 16 and 28) we can use $[i, i]$ as one letter is added to X per one letter of W. By Lemma 51 we can have at most $\lfloor \frac{n-k+1}{2} \rfloor$ occurrences of letter #. The check at Line 26 can be implemented in constant time after linear-time pre-processing of W

for longest common extension queries [113]. All other operations take in total linear time in n. Thus there exists an $\mathcal{O}(n)$-sized representation of X and it is constructible in $\mathcal{O}(n)$ time. □

C.5 PFS-ALGO

Lemma 51 tells us that X is the shortest string satisfying constraint **C1** and properties **P1**-**P4**. If we were to drop **P1** and employ the partial order **Π1** (see Problem 2), the length of $X = X_1\#\ldots\#X_N$ would not always be minimal: if a *permutation* of the strings $X_1, \ldots, X_N$ contains pairs X_i, X_j with a suffix-prefix overlap of length $\ell = k-1$, we may further apply **R2**, obtaining a shorter string.

To find such a permutation efficiently and construct from W a string Y shorter than X, we propose PFS-ALGO. The crux of our algorithm is an efficient method to solve a variant of the classic NP-complete *Shortest Common Superstring* (SCS) problem [160]. Specifically, our algorithm: (I) Computes the string X using Theorem 29. (II) Constructs a collection $\mathcal{B}'$ of strings, each of two letters (two ranks); the first (resp., second) letter is the lexicographic rank of the length-ℓ prefix (resp., suffix) of each string in the collection $\mathcal{B} = \{X_1, \ldots, X_N\}$. (III) Computes a shortest string containing every element in $\mathcal{B}'$ as a distinct substring. (IV) Constructs Y by mapping back each element to its distinct substring in $\mathcal{B}$. If there are multiple possible shortest strings, one is selected arbitrarily.

Example 29 (Illustration of the workings of PFS-ALGO)**.** *Let* $\ell = k - 1 = 3$ *and*

$$X = \texttt{aabaa}\#\texttt{aaacbcbbba}\#\texttt{baabbacaab}.$$

The collection $\mathcal{B}$ *is comprised of the following substrings:* $X_1 =$ `aabaa`*,* $X_2 =$ `aaacbcbbba`*, and* $X_3 =$ `baabbacaab`*. The collection* $\mathcal{B}'$ *is comprised of the following two-letter strings:* `23`, `14`, `32`*. To construct* B'*, we first find the length-*3 *prefix and the length-*3 *suffix of each* X_i*,* $i \in [1, 3]$*, which leads to a collection* $\{$`aab`, `baa`, `aaa`, `bba`$\}$*. Then, we sort the collection lexicographically to obtain* $\{$`aaa`, `aab`, `baa`, `bba`$\}$*, and last we replace each* X_i*,* $i \in [1, 3]$*, with the lexicographic ranks of its length-*3 *prefix and length-*3 *suffix. For instance,* X_1 *is replaced by* 23*. After that, a shortest string containing all elements of* $\mathcal{B}'$ *as distinct substrings is computed as:* `14` $\cdot$ `232`*. This shortest string is mapped back to the solution* $Y =$ `aaacbcbbba#aabaabbacaab`*. Note,* Y *contains one occurrence of* # *and has length* 23*, while* X *contains* 2 *occurrences of* # *and has length* 27*.* □

We now present the details of PFS-ALGO. We first introduce the *Fixed-Overlap Shortest String with Multiplicities* (FO-SSM) problem: Given a *collection* $\mathcal{B}$ of strings $B_1, \ldots, B_{|\mathcal{B}|}$ and an integer ℓ, with $|B_i| > \ell$, for all $1 \le i \le |\mathcal{B}|$, FO-SSM seeks to find a shortest string containing each element of $\mathcal{B}$ as a distinct substring using the following operations on any pair of strings B_i, B_j:

(I) $\texttt{concat}(B_i, B_j) = B_i \cdot B_j$;

(II) $\ell\text{-}\texttt{merge}(B_i, B_j) = B_i[0\,..\,|B_i| - 1 - \ell]B_j[0\,..\,|B_j| - 1] = B_i[0\,..\,|B_i| - 1 - \ell] \cdot B_j$.

Any solution to FO-SSM with $\ell := k-1$ and $\mathcal{B} := X_1, \ldots, X_N$ implies a solution to the PFS problem, because $|X_i| > k-1$ for all i's (see Lemma 51, **P3**)

The FO-SSM problem is a variant of the SCS problem. In the SCS problem, we are given a *set* of strings and we are asked to compute the shortest common superstring of the elements of this set. The SCS problem is known to be NP-complete, even for binary strings [160]. However, if all strings are of length two, the SCS problem admits a linear-time solution [160]. We exploit this crucial detail positively to show a linear-time solution to the FO-SSM problem in Lemma 53. In order to arrive to this result, we first adapt the SCS linear-time solution of [160] to our needs (see Lemma 52) and plug this solution into Lemma 53.

Lemma 52. *Let $\mathcal{Q}$ be a collection of q strings, each of length two, over an alphabet $\Sigma = \{1, \ldots, (2q)^{\mathcal{O}(1)}\}$. We can compute a shortest string containing every element of $\mathcal{Q}$ as a distinct substring in $\mathcal{O}(q)$ time.*

Proof. We sort the elements of $\mathcal{Q}$ lexicographically in $\mathcal{O}(q)$ time using radixsort. We also replace every letter in these strings with their *lexicographic rank* from $\{1, \ldots, 2q\}$ in $\mathcal{O}(q)$ time using radixsort. In $\mathcal{O}(q)$ time we construct the de Bruijn multigraph G of these strings [89]. Within the same time complexity, we find all nodes v in G with in-degree, denoted by $\mathrm{IN}(v)$, smaller than out-degree, denoted by $\mathrm{OUT}(v)$. We perform the following two steps:

Step 1 While there exists a node v in G with $\mathrm{IN}(v) < \mathrm{OUT}(v)$, we start an arbitrary path (with possibly repeated nodes) from v, traverse consecutive edges and delete them. Each time we delete an edge, we update the in- and out-degree of the affected nodes. We stop traversing edges when a node v' with $\mathrm{OUT}(v') = 0$ is reached: whenever $\mathrm{IN}(v') = \mathrm{OUT}(v') = 0$, we also delete v' from G. Then, we add the traversed path $p = v \ldots v'$ to a set $\mathcal{P}$ of paths. The path can contain the same node v more than once. If G is empty we halt. Proceeding this way, there are no two elements p_1 and p_2 in $\mathcal{P}$ such that p_1 starts with v and p_2 ends with v; thus this path decomposition is minimal. If G is not empty at the end, by construction, it consists of only cycles.

Step 2 While G is not empty, we perform the following. If there exists a cycle c that *intersects* with any path p in $\mathcal{P}$, we splice c into p, update p with the result of splicing, and delete c from G. This operation can be efficiently implemented by maintaining an array A of size $2q$ of linked lists over the paths in $\mathcal{P}$: $A[\alpha]$ stores a list of pointers to all occurrences of letter α in the elements of $\mathcal{P}$. Thus in constant time per node of c we check if any such path p exists in $\mathcal{P}$ and splice the two in this case. If no such path exists in $\mathcal{P}$, we add to $\mathcal{P}$ any of the path-linearizations of the cycle, and delete the cycle from G. After each change to $\mathcal{P}$, we update A and delete every node u with $\mathrm{IN}(u) = \mathrm{OUT}(u) = 0$ from G.

The correctness of this algorithm follows from the fact that $\mathcal{P}$ is a minimal path decomposition of G. Thus any concatenation of paths in $\mathcal{P}$ represents a shortest string containing all elements in $\mathcal{Q}$ as distinct substrings. □

Lemma 53. *Let $\mathcal{B}$ be a collection of strings over an alphabet $\Sigma = \{1, \ldots, ||\mathcal{B}||^{\mathcal{O}(1)}\}$. Given an integer ℓ, the FO-SSM problem for $\mathcal{B}$ can be solved in $\mathcal{O}(||\mathcal{B}||)$ time.*

Proof. Consider the following renaming technique. Each length-ℓ substring of the collection is assigned a *lexicographic rank* from the range $\{1, \ldots, ||\mathcal{B}||\}$. Each string in $\mathcal{B}$ is converted to a two-letter string as follows. The first letter is the lexicographic rank of its length-ℓ prefix and the second letter is the lexicographic rank of its length-ℓ suffix. We thus obtain a new *collection* $\mathcal{B}'$ of two-letter strings. Computing the ranks for all length-ℓ substrings in $\mathcal{B}$ can be implemented in $\mathcal{O}(||\mathcal{B}||)$ time by employing radixsort to sort Σ and then the well-known LCP data structure over the concatenation of strings in $\mathcal{B}$ [113]. The FO-SSM problem is thus solved by finding a shortest string containing every element of $\mathcal{B}'$ as a distinct substring. Since $\mathcal{B}'$ consists of two-letter strings only, we can solve the problem in $\mathcal{O}(|\mathcal{B}'|)$ time by applying Lemma 52. The statement follows. □

Thus, PFS-ALGO applies Lemma 53 on $\mathcal{B} := X_1, \ldots, X_N$ with $\ell := k - 1$ (recall that $X_1 \# \ldots \# X_N = X$). Note that each time the `concat` operation is performed, it also places the letter # in between the two strings.

Lemma 54. *Let W be a string of length n over an alphabet Σ. Given $k < n$ and array C,* PFS-ALGO *constructs a shortest string Y with* ***C1****,* **Π1***, and* ***P2-P4****.*

Proof. **C1** and **P2** hold trivially for Y as no length-k substring over Σ is added or removed from X. Let $X = X_1 \# \ldots \# X_N$. The order of non-sensitive length-k substrings within X_i, for all $i \in [1, N]$, is preserved in Y. Thus there exists an injective function f from the p-chains of $\mathcal{I}_W$ to the p-chains of $\mathcal{I}_Y$ such that $f(\mathcal{J}_W) \equiv \mathcal{J}_W$ for any p-chain $\mathcal{J}_W$ of $\mathcal{I}_W$ (**Π1** is preserved). **P3** also holds trivially for Y as no occurrence of # is added. Since $|Y| \leq |X|$, for **P4**, it suffices to note that the construction of W in the proof of tightness in Lemma 51 (see also Example 28) ensures that there is no suffix-prefix overlap of length $k-1$ between *any* pair of length-k substrings of Y over Σ due to the property of the order-$(k-1)$ de Bruijn sequence. Thus the upper bound of $\lceil \frac{n-k+1}{2} \rceil \cdot k + \lfloor \frac{n-k+1}{2} \rfloor$ on the length of X is also tight for Y.

The minimality on the length of Y follows from the minimality of $|X|$ and the correctness of Lemma 53 that computes a shortest such string. □

Let us now show the main result of this section.

Theorem 30. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. Given $k < n$ and $\mathcal{S}$,* PFS-ALGO *solves Problem 2 in the optimal $\mathcal{O}(n + |Y|)$ time.*

Proof. We compute the $\mathcal{O}(n)$-sized representation of string X with respect to W described in the proof of Theorem 29. This can be done in $\mathcal{O}(n)$ time. If $X \in \Sigma^*$, then we construct and return $Y := X$ in time $\mathcal{O}(|Y|)$ from the representation. If $X \in (\Sigma \cup \{\#\})^*$, implying $|Y| \leq |X|$, we compute the LCP data structure of string W in $\mathcal{O}(n)$ time [113]; and implement Lemma 53 in $\mathcal{O}(n)$ time by avoiding to read string X explicitly: we rather rename $X_1, \ldots, X_N$ to a collection of two-letter strings by employing the LCP information of W directly. We then construct and report Y in time $\mathcal{O}(|Y|)$. Correctness follows directly from Lemma 54. □

C.6 MCSR Problem, MCSR-ALGO, and Implausible Pattern Elimination

In the following, we introduce the MCSR problem and prove that it is NP-hard (see Section C.6.1). Then, we introduce MCSR-ALGO, a heuristic to address this problem (see Section C.6.2). Finally, we discuss how to configure MCSR-ALGO in order to eliminate implausible patterns (see Section C.6.3).

C.6.1 The MCSR Problem

The strings X and Y, constructed by TFS-ALGO and PFS-ALGO, respectively, may contain the separator #, which reveals information about the location of the sensitive patterns in W. Specifically, a malicious data recipient can go to the position of a # in X and "undo" Rule **R1** that has been applied by TFS-ALGO, removing # and the $k-1$ letters after # from X. The result could be an occurrence of the sensitive pattern. For example, applying this process to the first # in X shown in Figure C.1 results in recovering the sensitive pattern `abab`. A similar attack is possible on the string Y produced by PFS-ALGO, although it is hampered by the fact that substrings within two consecutive #s in X often swap places in Y.

To address this issue, we seek to construct a new string Z, in which #s are either deleted or replaced by letters from Σ. To preserve data utility, we favor separator replacements that have a small cost in terms of occurrences of τ-ghosts (patterns with frequency less than τ in W and at least τ in Z) and incur a level of distortion bounded by a parameter θ in Z. The cost of an occurrence of a τ-ghost at a certain position is given by function *Ghost*, while function *Sub* assigns a distortion weight to each letter that could replace a #. Both functions will be described in further detail below.

To preserve privacy, we require separator replacements not to reinstate sensitive patterns. This is the MCSR problem, a restricted version of which is presented in Problem 4. The restricted version is referred to as $\text{MCSR}_{k=1}$ and differs from MCSR in that it uses $k=1$ for the pattern length instead of an arbitrary value $k>0$. $\text{MCSR}_{k=1}$ is presented next for simplicity and because it is used in the proof of Lemma 55. Lemma 55 implies Theorem 31.

Problem 4 ($\text{MCSR}_{k=1}$)**.** *Given a string Y over an alphabet $\Sigma \cup \{\#\}$ with $\delta > 0$ occurrences of letter #, and parameters τ and θ, construct a new string Z by substituting the δ occurrences of # in Y with letters from Σ, such that:*

$$\text{(I)} \sum_{\substack{i:Y[i]=\#,\ Freq_Y(Z[i])<\tau \\ Freq_Z(Z[i])\geq\tau}} \text{Ghost}(i, Z[i]) \text{ is minimum, and (II)} \sum_{i:Y[i]=\#} \text{Sub}(i, Z[i]) \leq \theta.$$

Lemma 55. *The $\text{MCSR}_{k=1}$ problem is NP-hard.*

Proof. We reduce the NP-hard *Multiple Choice Knapsack* (MCK) problem [326] to $\text{MCSR}_{k=1}$ in polynomial time. In MCK, we are given a set of elements subdivided into δ mutually exclusive classes, $C_1, \ldots, C_\delta$, and a knapsack. Each class C_i has $|C_i|$ elements. Each element $j \in C_i$ has an arbitrary cost $c_{ij} \geq 0$ and an arbitrary weight

w_{ij}. The goal is to minimize the total cost (Eq. C.1) by filling the knapsack with one element from each class (constraint II), such that the weights of the elements in the knapsack satisfy constraint I, where constant $b \geq 0$ represents the minimum allowable total weight of the elements in the knapsack:

$$\min \sum_{i \in [1,\delta]} \sum_{j \in C_i} c_{ij} \cdot x_{ij} \tag{C.1}$$

subject to the constraints: (I) $\sum_{i \in [1,\delta]} \sum_{j \in C_i} w_{ij} \cdot x_{ij} \geq b$, (II) $\sum_{j \in C_i} x_{ij} = 1, \quad i = 1, \ldots \delta$, and (III) $x_{ij} \in \{0, 1\}, \quad i = 1, \ldots, \delta, \quad j \in C_i$.

The variable x_{ij} takes value 1 if the element j is chosen from class C_i, 0 otherwise (constraint III). We reduce any instance $\mathtt{I}_{\mathrm{MCK}}$ to an instance $\mathtt{I}_{\mathrm{MCSR}_{k=1}}$ in polynomial time, as follows:

(I) Alphabet Σ consists of letters α_{ij}, for each $j \in C_i$ and each class C_i, $i \in [1, \delta]$.

(II) We set $Y = \alpha_{11}\alpha_{12} \ldots \alpha_{1|C_1|} \# \ldots \# \alpha_{\delta 1}\alpha_{\delta 2} \ldots \alpha_{\delta|C_\delta|} \#$. Every element of Σ occurs exactly once: $\mathrm{Freq}_Y(\alpha_{ij}) = 1$. Letter $\#$ occurs δ times in Y. For convenience, let us denote by $\mu(i)$ the ith occurrence of $\#$ in Y.

(III) We set $\tau = 2$ and $\theta = \delta - b$.

(IV) $\mathrm{Ghost}(\mu(i), \alpha_{ij}) = c_{ij}$ and $\mathrm{Sub}(\mu(i), \alpha_{ij}) = 1 - w_{ij}$. The functions are otherwise *not defined*.

This is clearly a polynomial-time reduction. We now prove the correspondence between a solution $S_{\mathtt{I}_{\mathrm{MCK}}}$ to the given instance $\mathtt{I}_{\mathrm{MCK}}$ and a solution $S_{\mathtt{I}_{\mathrm{MCSR}_{k=1}}}$ to the instance $\mathtt{I}_{\mathrm{MCSR}_{k=1}}$.

We first show that if $S_{\mathtt{I}_{\mathrm{MCK}}}$ is a solution to $\mathtt{I}_{\mathrm{MCK}}$, then $S_{\mathtt{I}_{\mathrm{MCSR}_{k=1}}}$ is a solution to $\mathtt{I}_{\mathrm{MCSR}_{k=1}}$. Since the elements in $S_{\mathtt{I}_{\mathrm{MCK}}}$ have minimum $\sum_{i \in [1,\delta]} \sum_{j \in C_i} c_{ij} \cdot x_{ij}$, $\mathrm{Freq}_Y(\alpha_{ij}) = 1$, and $\tau = 2$, the letters $\alpha_1, \ldots, \alpha_\delta$ corresponding to the selected elements lead to a Z that incurs a minimum

$$\sum_{i \in [1,\delta]} \sum_{\substack{j = \mu(i) : \mathrm{Freq}_Y(Z[j]) < \tau \\ \mathrm{Freq}_Z(Z[j]) \geq \tau}} \mathrm{Ghost}(j, Z[j]). \tag{C.2}$$

In addition, each letter $Z[j]$ that is considered by the inner sum of Eq. C.2 corresponds to a single occurrence of $\#$, and these are all the occurrences of $\#$. Thus we obtain that

$$\sum_{i \in [1,\delta]} \sum_{\substack{j = \mu(i) : \mathrm{Freq}_Y(Z[j]) < \tau \\ \mathrm{Freq}_Z(Z[j]) \geq \tau}} \mathrm{Ghost}(j, Z[j]) = \sum_{\substack{i : Y[i] = \#, \ \mathrm{Freq}_Y(Z[i]) < \tau \\ \mathrm{Freq}_Z(Z[i]) \geq \tau}} \mathrm{Ghost}(i, Z[i]) \tag{C.3}$$

(i.e., condition I in Problem 4 is satisfied). Since the elements in $S_{\mathtt{I}_{\mathrm{MCK}}}$ have total weight $\sum_{i \in [1,\delta]} \sum_{j \in C_i} w_{ij} \cdot x_{ij} \geq b$, the letters $\alpha_1, \ldots, \alpha_\delta$, they map to, lead to a Z with $\sum_{i \in [1,\delta]} \sum_{j \in C_i} (1 - \mathrm{Sub}(\mu(i), \alpha_i)) \cdot x_{ij} \geq \delta - \theta$, which implies

$$\sum_{i\in[1,\delta]}\sum_{j\in C_i}\mathrm{Sub}(\mu(i),\alpha_{ij})\cdot x_{ij}=\sum_{i:Y[i]=\#}\mathrm{Sub}(i,Z[i])\leq\theta \tag{C.4}$$

(i.e., condition II in Problem 4 is satisfied). $S_{\mathtt{I}_{\mathrm{MCSR}_{k=1}}}$ is thus a solution to $\mathtt{I}_{\mathrm{MCSR}_{k=1}}$.

We finally show that, if $S_{\mathtt{I}_{\mathrm{MCSR}_{k=1}}}$ is a solution to $\mathtt{I}_{\mathrm{MCSR}_{k=1}}$, then $S_{\mathtt{I}_{\mathrm{MCK}}}$ is a solution to $\mathtt{I}_{\mathrm{MCK}}$. Since each $\#_i$, $i\in[1,\delta]$, is replaced by a single letter α_i in $S_{\mathtt{I}_{\mathrm{MCSR}_{k=1}}}$, exactly one element will be selected from each class C_i (i.e., conditions II-III of MCK are satisfied). Since the letters in $S_{\mathtt{I}_{\mathrm{MCSR}_{k=1}}}$ satisfy condition I of Problem 4, every element of Σ occurs exactly once in Y, and $\tau=2$, their corresponding selected elements $j_1\in C_1,\ldots,j_\delta\in C_\delta$ will have a minimum total cost. Since $S_{\mathtt{I}_{\mathrm{MCSR}_{k=1}}}$ satisfies $\sum_{i:Y[i]=\#}\mathrm{Sub}(i,Z[i])=\sum_{i\in[1,\delta]}\sum_{j\in C_i}\mathrm{Sub}(\mu(i),\alpha_{ij})\cdot x_{ij}\leq\theta$, the selected elements $j_1\in C_1,\ldots,j_\delta\in C_\delta$ that correspond to $\alpha_1\ldots,\alpha_\delta$ will satisfy $\sum_{i\in[1,\delta]}\sum_{j\in C_i}(1-w_{ij})\cdot x_{ij}\leq\delta-b$, which implies $\sum_{i\in[1,\delta]}\sum_{j\in C_i}w_{ij}\cdot x_{ij}\geq b$ (i.e., condition I of MCK is satisfied). Therefore, $S_{\mathtt{I}_{\mathrm{MCK}}}$ is a solution to $\mathtt{I}_{\mathrm{MCK}}$. The statement follows. □

Lemma 55 implies the main result of this section.

Theorem 31. *The* MCSR *problem is NP-hard.*

The cost of τ-ghosts is captured by a function Ghost. This function assigns a cost to an occurrence of a τ-ghost, which is caused by a separator replacement at position i, and is specified based on domain knowledge. For example, with a cost equal to 1 for each gained occurrence of each τ-ghost, we penalize more heavily a τ-ghost with frequency much below τ in Y and the penalty increases with the number of gained occurrences. Moreover, we may want to penalize positions towards the end of a temporally ordered string, to avoid spurious patterns that would be deemed important in applications based on time-decaying models [111].

The replacement distortion is captured by a function Sub which assigns a weight to a letter that could replace a # and is specified based on domain knowledge. The maximum allowable replacement distortion is θ. Small weights favor the replacement of separators with desirable letters (e.g., letters that reinstate non-sensitive frequent patterns) and letters that reinstate sensitive patterns are assigned a weight larger than θ that prohibits them from replacing a #. As will be explained in Section C.6.3, weights larger than θ are also assigned to letters which would lead to implausible substrings [187] if they replaced #s.

C.6.2 MCSR-ALGO

We next present MCSR-ALGO, a non-trivial heuristic that exploits the connection of the MCSR and MCK [295] problems. We start with a high-level description of MCSR-ALGO:

(I) Construct the set of all candidate τ-ghost patterns (i.e., length-k strings over Σ with frequency below τ in Y that can have frequency at least τ in Z).

(II) Create an instance of MCK from an instance of MCSR. For this, we map the ith occurrence of # to a class C_i in MCK and each possible replacement of the occurrence with a letter j to a different item in C_i. Specifically, we consider all possible replacements with letters in Σ and also a replacement with the empty string, which models deleting (instead of replacing) the ith occurrence of #. In addition, we set the costs and weights that are input to MCK as follows. The cost for replacing the ith occurrence of # with the letter j is set to the sum of the Ghost function for all candidate τ-ghost patterns when the ith occurrence of # is replaced by j. That is, we make the worst-case assumption that the replacement forces all candidate τ-ghosts to become τ-ghosts in Z. The weight for replacing the ith occurrence of # with letter j is set to $\mathrm{Sub}(i, j)$.

(III) Solve the instance of MCK and translate the solution back to a (possibly suboptimal) solution of the MCSR problem. For this, we replace the ith occurrence of # with the letter corresponding to the element chosen by the MCK algorithm from class C_i, and similarly for each other occurrence of #. If the instance has no solution (i.e., no possible replacement can hide the sensitive patterns), MCSR-ALGO reports that Z cannot be constructed and terminates.

Lemma 56 below states the running time of an efficient implementation of MCSR-ALGO.

Lemma 56. MCSR-ALGO *runs in* $\mathcal{O}(|Y| + k\delta\sigma + \mathcal{T}(\delta, \sigma))$ *time, where* $\mathcal{T}(\delta, \sigma)$ *is the running time of the MCK algorithm for* δ *classes with* $\sigma + 1$ *elements each.*

Proof. It should be clear that if we conceptually extend Σ with the empty string, our approach takes into account the possibility of deleting (instead of replacing) an occurrence of #. To ease comprehension though we only describe the case of letter replacements.

Step 1 Given Y, Σ, k, δ, and τ, we construct a set $\mathcal{C}$ of *candidate τ-ghosts* as follows. The candidates are at most $(|Y| - k + 1 - k\delta) + (k\delta\sigma) = \mathcal{O}(|Y| + k\sigma\delta)$ distinct strings of length k. The first term corresponds to all substrings of length k over Σ occurring in Y (i.e., if Y did not contain #, we would have $|Y| - k + 1$ such substrings; each of the δ # causes the loss of k such substrings). The second term corresponds to all possible substrings of length k that may be introduced in Z but do not occur in Y. For any string U from the set of these $\mathcal{O}(|Y| + k\delta\sigma)$ strings, we want to compute $\mathrm{Freq}_Y(U)$ and its *maximal frequency* in Z, denoted by $\max \mathrm{Freq}_Z(U)$, i.e., the largest possible frequency that U can have in Z, to construct set $\mathcal{C}$. Let S_{ij} denote the string of length $2k - 1$, containing the k consecutive length-k substrings, obtained after replacing the ith occurrence of # with letter j in Y.

(I) If $\mathrm{Freq}_Y(U) \geq \tau$, U by definition can never become τ-ghost in Z, and we thus exclude it from $\mathcal{C}$. $\mathrm{Freq}_Y(U)$, for all U occurring in Y, can be computed in $\mathcal{O}(|Y|)$ total time using the suffix tree of Y [113].

(II) If $\max \mathrm{Freq}_Z(U) < \tau$, U by definition can never become τ-ghost in Z, and we thus exclude it from $\mathcal{C}$. $\max \mathrm{Freq}_Z(U)$ can be computed by adding to $\mathrm{Freq}_Y(U)$ the

maximum additional number of occurrences of U caused by a letter replacement among all possible letter replacements. We sum up this quantity for each U and for all replacements of occurrences of # to obtain $\max \mathrm{Freq}_Z(U)$. To do this, we first build the generalized suffix tree of $Y, S_{11}, \ldots, S_{\delta\sigma}$ in $\mathcal{O}(|Y| + k\delta\sigma)$ time [113]. We then spell $S_{i1}, \ldots, S_{i\sigma}$, for all i, in the generalized suffix tree in $\mathcal{O}(k\sigma)$ time per i. We exploit suffix links to spell the length-k substrings of S_{ij} in $\mathcal{O}(k)$ time and memorize the maximum number of occurrences of U caused by replacing the ith occurrence of # among all j. We represent set $\mathcal{C}$ on the generalized suffix tree by marking the corresponding nodes, and we denote this representation by $T(\mathcal{C})$. The total size of this representation is $\mathcal{O}(|Y| + k\sigma\delta)$.

Step 2 We now want to construct an instance of the MCK problem using $T(\mathcal{C})$. We first set letter j as element α_{ij} of class C_i. We then set c_{ij} equal to the sum of the Ghost function cost incurred by replacing the ith occurrence of # by letter j for all (at most k) affected length-k substrings that are marked in $T(\mathcal{C})$. The main assumption of our heuristic is precisely the fact that we assume that this letter replacement will force all of these affected length-k substrings becoming τ-ghosts in Z. The computation of c_{ij} is done as follows. For each (i, j), $i \in [1, \delta]$ and $j \in [1, \sigma]$, we have k substrings whose frequency changes, each of length k. Let U be one such pattern occurring at position t of Z, where $\mu(i) - k + 1 \leq t \leq \mu(i)$ and $\mu(i)$ is the ith occurrence of # in Y. We check if U is marked in $T(\mathcal{C})$ or not. If U is not marked we add nothing to c_{ij}. If U is marked, we increment c_{ij} by $\mathrm{Ghost}(t, U)$. We also set $w_{ij} = \mathrm{Sub}(i, j)$ (as stated above, any letter that reinstates a sensitive pattern is assigned a weight $\mathrm{Sub} > \theta$, so that it cannot be selected to replace an occurrence of # in Step 3). Similar to Step 1, the total time required for this computation is $\mathcal{O}(|Y| + k\delta\sigma)$.

Step 3 In Step 2, we have computed c_{ij} and w_{ij}, for all i, j, $i \in [1, \delta]$ and $j \in [1, \sigma]$. We thus have an instance of the MCK problem. We solve it and translate the solution back to a (suboptimal) solution of the MCSR problem: the element α_{ij} chosen by the MCK algorithm from class C_i corresponds to letter j and it is used to replace the ith occurrence of #, for all $i \in [1, \delta]$. The cost of solving MCK depends on the chosen algorithm and is given by a function $\mathcal{T}(\delta, \sigma)$.

Thus, the total cost of MCSR-ALGO is $\mathcal{O}(|Y| + k\delta\sigma + \mathcal{T}(\delta, \sigma))$. □

C.6.3 Eliminating Implausible Patterns

We present the notion of implausible substring and explain how we can ensure that implausible patterns do not occur in Z, as a result of applying the MCSR-ALGO algorithm to string Y.

Consider, for instance, an input string $Y = \ldots$ `a#c` $\ldots$ that models the movement of an individual, and the string `abc`, which is created as a substring of Z when we replace # with `b`. Consider further that an individual can, generally, not go from `a` to `c` through `b`, or that it is highly unlikely for them to do so. We call a substring such as `abc` *implausible*. Clearly, if `abc` occurs in Z, it may be possible for an attacker to infer that `b` replaced #, and then infer a sensitive pattern by "undoing" **R1** as explained in Section C.6.1. In order to effectively model this scenario, we define implausible patterns

based on a statistical significance measure for strings [77, 308, 25]. The measure is defined as follows [77]:

$$z_W(U) = \frac{\mathrm{Freq}_W(U) - \mathbb{E}_W[U]}{\max(\sqrt{\mathbb{E}_W[U]}, 1)},$$

where U is a string with $|U| > 2$, W is the reference string, and

$$\mathbb{E}_W[U] = \begin{cases} \frac{\mathrm{Freq}_W(U[0..|U|-2]) \cdot \mathrm{Freq}_W(U[1..|U|-1])}{\mathrm{Freq}_W(U[1..|U|-2])}, & \mathrm{Freq}_W(U[1..|U|-2]) > 0 \\ 0, \text{otherwise} \end{cases}$$

is the expected frequency of U in W, computed based on an independence assumption between the event "$U[0..|U|-1]$ occurs in W" and "$U[1..|U|-1]$ occurs in W". The measure z_W is a normalized version of the standard score of U, based on the fact that the variance $\mathrm{Var}_W[U] \approx \sqrt{\mathbb{E}_W[U]}$ [308]. A small $z_W(U)$ indicates that U occurs less likely than expected, and hence it can naturally be considered as an artefact of sanitization.

Given a user-defined threshold $\rho < 0$, we define a string U as *ρ-implausible* if $z_W(U) < \rho$. The set of ρ-implausible substrings of W can be computed in the optimal $\mathcal{O}(|\Sigma| \cdot |W|)$ time [25]. We use W as the reference string, assuming that it is a good representation of the domain; e.g., a trip (substring) that is ρ-implausible in W is also implausible in general. Alternatively, one could use any other string as reference, impose length constraints on implausible patterns [255, 342], or even directly specify substrings that should not occur in Z based on domain knowledge.

Given the set $\mathcal{U}$ of (ρ-)implausible patterns, we ensure that no # replacement creates $U = U_1 \alpha U_2 \in \mathcal{U}$ in Z, where α is the letter that replaces #, by assigning a weight $\mathrm{Sub}(i, Z[i]) > \theta$, for each $Z[i]$ such that $Y[i] = \#$ and $U_1 \cdot Z[i] \cdot U_2 \in \mathcal{U}$. This guarantees that no replacement leading to an artefact occurrence of an element of $\mathcal{U}$ is performed by MCSR-ALGO. Note, however, that a ρ-implausible pattern may occur in Z as a substring, either because it occurred in a part of W that was copied to Z (e.g., a non-sensitive pattern), or due to the change of frequency of some substrings that are created in Z after the replacement of a #. However, since such ρ-implausible patterns did not contain a # in the first place, they cannot be exploited by an attacker seeking to reverse the construction of Z.

C.7 ETFS-ALGO

Let U and V be two non-sensitive length-k substrings of W such that U is the t-predecessor of V. Since U and V must occur in the same order in the solution string X_{ED}, the main choice we have to make in order to solve the ETFS problem is whether to:

(I) "merge" U and V when the length-$(k-1)$ suffix of U and the length-$(k-1)$ prefix of V match; or

(II) "interleave" U and V with a carefully selected string over $\Sigma \cup \{\#\}$.

Among operations I and II, for every such pair U and V, we must select the operation that *globally* results in the smallest number of edit operations. Operations I and II can naturally be expressed by means of a regular expression E. In particular, this implies that any instance of the ETFS problem can be reduced to an instance of approximate regular expression matching and thus an algorithm for approximate regular expression matching between E and W [279] can be employed. More formally, given a string W and a regular expression E, the *approximate regular expression matching* problem is to find a string T that matches E with minimal $d_E(W,T)$. The following result is known.

Theorem 33 ([279]). *Given a string W and a regular expression E, the approximate regular expression matching problem can be solved in $\mathcal{O}(|W|\cdot|E|)$ time.*

In the following, we define a specific type of a regular expression E. Let us first define the following regular expression:

$$\Sigma^{<k} = (\underbrace{(a_1|a_2|\ldots|a_{|\Sigma|}|\varepsilon)\ldots(a_1|a_2|\ldots|a_{|\Sigma|}|\varepsilon)}_{k-1 \text{ times}}),$$

where $\Sigma = \{a_1, a_2, \ldots, a_{|\Sigma|}\}$ is the alphabet of W and $k > 1$. We also define the following regular-expression gadgets, for a letter $\# \notin \Sigma$:

$$\oplus = \#(\Sigma^{<k}\#)^*, \quad \ominus = (\Sigma^{<k}\#)^*, \quad \otimes = (\#\Sigma^{<k})^*.$$

Intuitively, the gadget $\oplus$ represents a string we may choose to include in the output in an effort to minimize the edit distance between W and the solution string X_{ED}. It should be clear that the length of $\oplus$ is in $\mathcal{O}(k|\Sigma|)$ and that $\oplus$ cannot generate any length-k substring over Σ. Furthermore, inserting $\oplus$ in E cannot create any sensitive or non-sensitive pattern due to the occurrences of $\#$ on both ends of $\oplus$. The gadgets $\ominus$ and $\otimes$ are similar to $\oplus$. They are added in the beginning and at the end of E, respectively. This is because E should not start or end with $\#$ as this would only increase the edit distance to W. As it will be explained later, to construct E, we also make use of the $|$ operator. Intuitively, the $|$ operator represents the choice we make between operation "merge" or "interleave".

We are now in a position to describe ETFS-ALGO, an algorithm for solving the ETFS problem. ETFS-ALGO starts by constructing E. Let $(N_1, N_2 \ldots, N_{|\mathcal{I}|})$ be the sequence of non-sensitive length-k substrings as they occur in W from left to right. We first set $E = \ominus N_1$ and then process the pairs of non-sensitive length-k substrings N_i and N_{i+1}, for all $i \in \{1, |\mathcal{I}| - 1\}$. At the ith step, we examine whether or not N_i and N_{i+1} can be merged. If they can, we append to E a regular expression $(A| \oplus N_{i+1})$, where A is obtained by chopping-off the length-$(k-1)$ prefix of N_{i+1} (that is, the remainder of N_{i+1} after merging it with N_i). Otherwise, we append $\oplus N_{i+1}$ to E. Intuitively, using A corresponds to choosing "merge" and $\oplus N_{i+1}$ to choosing "interleave". After examining each pair N_i and N_{i+1}, we append $\otimes$ to E. This concludes the construction of E. Note how, for any combination of choices, N_{i+1} will always appear in the string obtained.

Next, ETFS-ALGO employs Theorem 33 to construct X_{ED}. In particular, it finds a string T that matches E with minimal $d_E(W,T)$. Last, it sets $X_{\mathrm{ED}} = T$. We arrive at the main result of this section.

Theorem 32. *Let W be a string of length n over an alphabet Σ. Given $k < n$ and $\mathcal{S}$,* ETFS-ALGO *solves Problem 3 in $\mathcal{O}(k|\Sigma|n^2)$ time.*

Proof. Constructing E can be done in $\mathcal{O}(n + kn + |E|) = \mathcal{O}(k|\Sigma|n)$ time, since: (I) The non-sensitive length-k substrings of W can be obtained in $\mathcal{O}(n)$ time, by reading W from left to right and checking $\mathcal{S}$. (II) Checking whether N_i and N_{i+1} are mergeable takes $\mathcal{O}(k)$ time via letter comparisons, and it is performed in each of the $\mathcal{O}(n)$ steps. (III) The length is $|E| = \mathcal{O}(kn + k|\Sigma|n) = \mathcal{O}(k|\Sigma|n)$. This is because E contains at most n occurrences of non-sensitive length-k substrings, at most n occurrences of $\oplus$, and one occurrence of each of $\ominus$ and $\otimes$ and because the lengths of $\oplus$, $\ominus$ and $\otimes$ are $\mathcal{O}(k|\Sigma|)$.

Computing T from W and E can be performed in $\mathcal{O}(|W| \cdot |E|) = \mathcal{O}(n \cdot |E|)$ time using Theorem 33. Thus ETFS-ALGO takes $\mathcal{O}(k|\Sigma|n^2)$ time in total.

The correctness of ETFS-ALGO follows from the fact that by construction: (I) T does not contain any sensitive pattern, so **C1** is satisfied; (II) T satisfies **P1** and **P2** as no length-k substring over Σ (other than the non-sensitive ones) is inserted in E; (III) All strings satisfying **C1**, **P1** and **P2** can be obtained by E, since they must have the same t-chain of non-sensitive patterns over Σ^* as W, interleaved by length-k substrings that are on $(\Sigma \cup \#)^*$ but *not* on Σ^*; and (IV) the minimality on edit distance is guaranteed by Theorem 33. The statement follows. □

A factor of $|\Sigma|$ can be shaved from $\mathcal{O}(k|\Sigma|n^2)$ via dynamic programming [56], albeit it seems unlikely to yield a strongly subquadratic time bound [38]. In any case, as our experiments show, TFS-ALGO, which runs in $\mathcal{O}(kn)$ time, outputs optimal or near-optimal solutions in practice.

Example 30 (Illustration of the workings of ETFS-ALGO)**.** *Let $W = \mathtt{aaabbaabaccbbb}$, $k = 4$, and the set of sensitive patterns be $\{\mathtt{aabb}, \mathtt{abba}, \mathtt{bbaa}, \mathtt{baab}, \mathtt{ccbb}\}$. The sequence of non-sensitive patterns is thus $(N_1, \ldots, N_6) = (\mathtt{aaab}, \mathtt{aaba}, \mathtt{abac}, \mathtt{bacc}, \mathtt{accb}, \mathtt{cbbb})$. Given that $k = 4$ and $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$,* ETFS-ALGO *constructs the following gadgets,*

$$\oplus = \#(\Sigma^{<4}\#)^* = \#(((\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon)(\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon)(\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon))\#)^*$$

$$\ominus = (\Sigma^{<4}\#)^* = (((\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon)(\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon)(\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon))\#)^*$$

$$\otimes = (\#\Sigma^{<4})^* = (\#((\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon)(\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon)(\mathtt{a}|\mathtt{b}|\mathtt{c}|\varepsilon)))^*$$

and sets $E = \ominus N_1 = \ominus\mathtt{aaab}$. Then, it iterates over each pair of consecutive non-sensitive length-k substrings in the order they appear in W (i.e., pair (N_i, N_{i+1}) is considered in Step $i \in [1, 5]$) and the regular expression E is updated, as detailed below.

In Step 1, ETFS-ALGO *considers the pair $(N_1, N_2) = (\mathtt{aaab}, \mathtt{aaba})$. Observe that in this case N_1 and N_2 can be merged, since the length-3 suffix of N_1 and the length-3 prefix of N_2 match. Thus, $(A|N_2) = (\mathtt{a}| \oplus \mathtt{aaba})$ is appended to E. Recall that when merging, we chop off the length-$(k-1)$ prefix of $N_{i+1} = N_2$ (because we have merged it already) and write down what is left of N_2 ($\mathtt{a}$ in this case) before $|$. Thus, $E = \ominus\mathtt{aaab}(\mathtt{a}| \oplus \mathtt{aaba})$.*

In Step 2, ETFS-ALGO *considers $(N_2, N_3) = (\mathtt{aaba}, \mathtt{abac})$. Again, N_2 and N_3 can be merged. Thus, $(\mathtt{c}| \oplus \mathtt{abac})$ is appended into E, which leads to $E = \ominus\mathtt{aaab}(\mathtt{a}| \oplus \mathtt{aaba})(\mathtt{c}| \oplus \mathtt{abac})$.*

In Steps 3 and 4, ETFS-ALGO *considers the pairs* $(N_3, N_4) = (\mathtt{abac}, \mathtt{bacc})$ *and* $(N_4, N_5) = (\mathtt{bacc}, \mathtt{accb})$*, respectively. Since the patterns in each pair can be merged, the algorithm appends into* E *the regular expression* $(\mathtt{c}| \oplus \mathtt{bacc})$ *and* $(\mathtt{b}| \oplus \mathtt{accb})$*, for the first and second pair, respectively. This leads to* $E = \ominus\mathtt{aaab}(\mathtt{a}| \oplus \mathtt{aaba})(\mathtt{c}| \oplus \mathtt{abac})(\mathtt{c}| \oplus \mathtt{bacc})(\mathtt{b}| \oplus \mathtt{accb})$.

In Step 5, ETFS-ALGO *considers the last pair* $(N_5, N_6) = (\mathtt{accb}, \mathtt{cbbb})$*, which cannot be merged, and appends* $\oplus\mathtt{cccb}$ *to* E*. Since there is no other pair to be considered,* $\otimes$ *is also appended to* E*, leading to:*

$$E = \ominus\underline{\mathtt{aaab}}(\mathtt{a}|\underline{\oplus\mathtt{aaba}})(\underline{\mathtt{c}}| \oplus \mathtt{abac})(\underline{\mathtt{c}}| \oplus \mathtt{bacc})(\underline{\mathtt{b}}| \oplus \mathtt{accb})\underline{\oplus\mathtt{cbbb}} \otimes .$$

At this point, ETFS-ALGO *employs Theorem 33 to find the following string* T *that matches* E *(the choices that were made in the construction of* T *are underlined in* E *and* $\ominus$, $\oplus$, $\otimes$ *are matched by the empty string):*

$$T = \mathtt{aaab\#aabaccb\#cbbb},$$

with minimal $d_E(T, W) = 4$*. Last,* ETFS-ALGO *returns* $X_{ED} = T$. □

Note that $X_{\mathrm{ED}} = T$ in Example 30 does not contain any sensitive pattern and that all non-sensitive patterns of W appear in T in the same order and with the same frequency as they appear in W. Note also that, for the same instance, TFS-ALGO would return string $X =$ `aaabaccb#cbbb` with $d_E(W, X) = 5 > d_E(W, X_{\mathrm{ED}}) = 4$ and $|X| = 13 < |X_{\mathrm{ED}}| = 17$.

C.8 Experimental Evaluation

We evaluate our algorithms in terms of *effectiveness* and *efficiency*. Effectiveness is measured based on data utility and number of implausible patterns. Efficiency is measured based on runtime.

Evaluated Algorithms First, we consider the pipeline TFS-ALGO→ PFS-ALGO→ MCSR-ALGO, referred to as TPM. Given a string W over Σ, TPM sanitizes W by applying TFS-ALGO, PFS-ALGO, and then MCSR-ALGO. MCSR-ALGO uses the $\mathcal{O}(\delta\sigma\theta)$-time algorithm of [295] for solving the MCK instances. The final output is a string Z over Σ. MCSR-ALGO is configured with an empty set $\mathcal{U}$ (i.e., it may lead to implausible patterns that are created in Z after the replacement of a #).

Among the related works discussed in Section C.2.1, we compared TPM against the PH heuristic [187]. This is because we found PH to be the closest to our setting, and, moreover, because it outperforms other related sequence sanitization methods [13, 172] (see Section C.2.1 for details). We also compared TPM against a greedy baseline referred to as BA, in terms of data utility and efficiency. BA initializes its output string Z_{BA} to W and then considers each sensitive pattern R in Z_{BA}, from left to right. For each R, BA replaces the letter r of R that has the largest frequency in Z_{BA} with another letter r' that is not contained in R and has the smallest frequency in Z_{BA}, breaking all ties arbitrarily. Note that this letter replacement should not introduce any other sensitive pattern in Z_{BA}. If no such r' exists, r is replaced by # to ensure that a

solution is produced (even if it may reveal the location of a sensitive pattern). Each replacement removes the occurrence of R and aims to prevent τ-ghost occurrences by selecting an r' that will not substantially increase the frequency of patterns overlapping with R. Note that BA does not preserve the frequency of non-sensitive patterns, and thus, unlike TPM, it can incur τ-lost patterns. We also implemented a similar baseline that replaces the letter in R that has the smallest frequency in Z_{BA} with another letter that is not contained in R and has the largest frequency in Z_{BA}, but omit its results as it was worse than BA.

In addition, we consider the pipelines TFS-ALGO→MCSR-ALGO and TFS-ALGO→MCSRI-ALGO, referred to as TM and TMI, respectively. With MCSRI-ALGO we refer to the configuration of MCSR in which there is a non-empty set $\mathcal{U}$ of ρ-implausible patterns that must not occur in the output string Z. We omit PFS-ALGO from the TM and TMI pipelines to avoid the elimination of some implausible patterns due to re-ordering of blocks of non-sensitive patterns that is performed by PFS-ALGO.

Last, we consider ETFS-ALGO, which we compare to TFS-ALGO, to demonstrate that the latter is a very effective heuristic for the ETFS problem.

Experimental Data We considered the following publicly available datasets used in [13, 172, 187, 256, 55]: Oldenburg (OLD), Trucks (TRU), MSNBC (MSN), the complete genome of *Escherichia coli* (DNA), and synthetic data (uniformly random strings, the largest of which is referred to as SYN). See Table F.1 for the characteristics of these datasets and the parameter values used in experiments, unless stated otherwise.

Dataset	**Domain**	**Length** n	$\lvert\Sigma\rvert$	**# sensitive patterns**	**# sensitive positions** $\lvert\mathcal{S}\rvert$	**Pattern length** k	**Implaus. pat. threshold** ρ
OLD	Movement	85,563	100	$[30, 240]$ (**60**)	$[600, 6103]$	$[3, 7]$ (**4**)	$[-2, -0.1]$ (**−1**)
TRU	Transport.	5,763	100	$[30, 120]$ (**10**)	$[324, 2410]$	$[2, 5]$ (**4**)	$[-3, -0.1]$ (**−4**)
MSN	Web	4,698,764	17	$[30, 120]$ (**60**)	$[6030, 320480]$	$[3, 8]$ (**4**)	$[-6, -3]$ (**−1**)
DNA	Genomic	4,641,652	4	$[25, 500]$ (**100**)	$[163, 3488]$	$[5, 15]$ (**13**)	$[-4.5, -2.5]$ (**−2.5**)
SYN	Synthetic	20,000,000	10	$[10, 1000]$ (**1000**)	$[10724, 20171]$	$[3, 6]$ (**6**)	
$\mathrm{SYN}_{\mathrm{BIN}}$	Synthetic	1,000	2	$[4, 32]$ (**16**)	$[16, 128]$	$[4, 7]$ (**4**)	-

Table C.3: Characteristics of datasets and values used (default values are in bold).

Experimental Setup The sensitive patterns were selected randomly among the frequent length-k substrings at minimum support τ following [172, 187, 256]. We used the fairly low values ($\tau = 10$ for TRU, SYN, and $\mathrm{SYN}_{\mathrm{BIN}}$; $\tau = 20$ for OLD and DNA; and $\tau = 200$ for MSN), to have a wider selection of sensitive patterns. In MCSR-ALGO, we used a uniform cost of 1 for every occurrence of each τ-ghost, a weight of 1 (resp., ∞) for each letter replacement that does not (resp., does) create a sensitive pattern, and we further set $\theta = \delta$. This setup treats all candidate τ-ghost patterns and all candidate letters for replacement uniformly, to facilitate a fair comparison with BA which cannot distinguish between τ-ghost candidates or favor specific letters. In MCSRI-ALGO, we instead set a weight ∞ for each letter replacement that does not create a sensitive pattern or an implausible pattern of length k.

In PH, we used a minimum frequency threshold of $\tau = 1$ to ensure that sensitive patterns will not occur as subsequences (and hence nor as substrings) in the output. We

also transformed the input string into a collection of strings and provided the collection as input to PH. This is because, although in principle PH can be applied to a single string, as in Example 27, this was not possible for any of the datasets of Table F.1. In fact, as it will be shown later, PH did not terminate within 12 hours, even for very short strings of length 25 that took milliseconds to be sanitized by our algorithms. The reason is that PH requires finding all occurrences of every sensitive pattern in the string and computing changes to the set of non-sensitive frequent sequential patterns incurred by permutation and deletion. When $\tau = 1$ and for reasonably long strings, this is a very computationally intensive task. This observation agrees with the findings in [187] and similar findings were reported for other sanitization algorithms [13, 172].

Therefore, to be able to compare with PH, we converted a long string to a collection of short strings (i.e., the type of dataset that PH was designed for). Specifically, we created a collection of strings $W_1, W_2, \ldots, W_m$ from a string W, such that $W = W_1 \cdot W_2 \cdot \ldots \cdot W_m$ and $|W_i| = r$, for $i \in [1, m]$, and then we applied PH to the collection. In our experiments, we varied r in $[5, 25]$ and used $r = 15$ as the default value. The smallest value $r = 5$ was selected to enable the hiding of sensitive patterns of length $k = 5$ that we used; the largest value $r = 25$ was selected empirically. PH took much longer as we increased r and did not terminate within 12 hours for $r = 25$. After applying PH, we obtained a sanitized collection of strings $W'_1, W'_2, \ldots, W'_m$ and constructed a final string $I = W'_1 \cdot W'_2 \cdot \ldots \cdot W'_m$ by concatenating the strings in the sanitized collection. Note that we favor PH by neglecting the possibility that sensitive patterns may be created when concatenating the strings in the sanitized collection.

To capture the utility of sanitized data, we used the *(frequency) distortion* measure

$$\sum_{U} (\mathrm{Freq}_W(U) - \mathrm{Freq}_Z(U))^2,$$

where $U \in \Sigma^k$ is a non-sensitive pattern. The distortion measure quantifies changes in the frequency of non-sensitive patterns with low values suggesting that Z remains useful for tasks based on pattern frequency (e.g., identifying motifs corresponding to functional or conserved DNA [297]).

We also measured the number of τ-ghost and τ-lost patterns in Z following [172, 187, 256], where a pattern U is *τ-lost* in Z if and only if $\mathrm{Freq}_W(U) \geq \tau$ but $\mathrm{Freq}_Z(U) < \tau$. That is, τ-lost patterns model knowledge that can no longer be mined from Z but could be mined from W, whereas τ-ghost patterns model knowledge that can be mined from Z but not from W. A small number of τ-lost/ghost patterns suggests that frequent pattern mining can be accurately performed on Z [172, 187, 256]. Unlike BA, by design TPM *does not* incur any τ-lost pattern, as TFS-ALGO and PFS-ALGO preserve frequencies of non-sensitive patterns, and MCSR-ALGO can only increase pattern frequencies.

To examine the benefit of using MCSRI-ALGO instead of MCSR-ALGO when implausible patterns need to be eliminated, we measured the percentage of ρ-implausible patterns of length k that may occur in Z, when a letter replaces a #. Clearly, the percentage is 0 when MCSRI-ALGO is used, and a large percentage for MCSR-ALGO implies that it is beneficial to use MCSRI-ALGO instead.

To capture the effectiveness of TFS-ALGO in terms of constructing a string X that is at small edit distance from W (see the ETFS problem), we used the *Edit Distance*

Relative Error, defined as

$$\frac{d_E(W, X) - d_E(W, X_{\mathrm{ED}})}{d_E(W, X_{\mathrm{ED}})}.$$

All experiments ran on a Desktop PC with an Intel Xeon E5-2640 at 2.66GHz and 16GB RAM. Our source code is written in `C++` and is accessible from

`https://bitbucket.org/stringsanitization/stringsanitizationtkdd/`

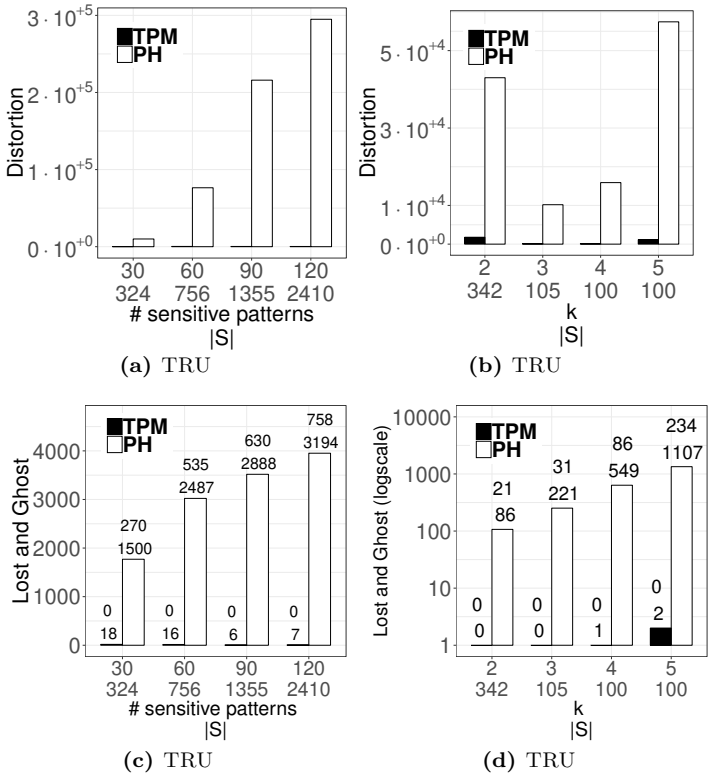
The code for PH is also written in `C++` and was provided by the authors of [187]. The results presented below have been averaged over 10 runs.

C.8.1 TPM vs. PH

Data Utility. We first demonstrate that TPM substantially outperformed PH in terms of distortion. This suggests that TPM is a much better method for preserving utility in tasks based on the frequency of substrings (e.g., [297]). Fig. C.2a shows that, for varying number of sensitive patterns, TPM incurred on average 477 (and up to 1045) times lower distortion than PH did. These results are expected because PH applies permutation and/or deletion to eliminate all occurrences of a sensitive pattern as a subsequence from the sanitized output, whereas only the occurrences in which the pattern is comprised of consecutive letters (i.e., the sensitive pattern occurs as a substring) should be eliminated. This "overprotection" incurs distortion unnecessarily and severely harms utility, particularly when there are more sensitive patterns. Indeed, Fig. C.2a shows that PH becomes less effective as the number of sensitive patterns increases. In addition, TPM incurred substantially less distortion than PH for all tested values of k. Fig. C.2b shows that TPM incurred on average 78 (and up to 169) times lower distortion than PH. This is again because our setting calls for hiding occurrences of sensitive patterns as substrings and, in this setting, PH overprotects data unnecessarily.

We now demonstrate that TPM allows substantially more accurate frequent substring mining than PH. Fig. C.2c shows that, for varying number of sensitive patterns, the number of τ-lost and τ-ghost patterns for TPM was on average 376 (and up to 586 times) lower compared to that of PH. Quantitatively similar results were obtained for varying k, as can be seen in Fig. C.2d. Specifically, the number of τ-lost and τ-ghost patterns for TPM was at least 21 (and up to 234) times lower than that of PH. Note that TPM creates no τ-lost patterns by design and it created no more than 2 τ-ghost patterns in the experiments of Fig. C.2d, while PH created up to 234 τ-lost and 1107 τ-ghost patterns.

Impact of r on Efficiency We demonstrate the runtime of PH as a function of r, the length of records in the collection of records $W_1, W_2, \ldots, W_m$ that was created from a string dataset W and given as input to PH. As can be seen in Fig. C.3a, the runtime of PH increased from 4 seconds when $r = 5$ to 2.5 hours when $r = 20$. Also, PH did not terminate within 12 hours for $r = 25$. This shows why it was not feasible to apply PH directly to an entire string dataset of Table F.1, and we needed to construct a collection of sequences instead. As mentioned in "Experimental setup" above, the

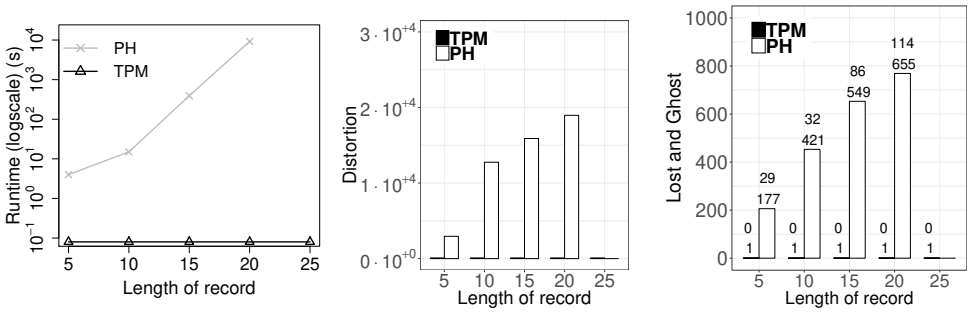


(a) TRU **(b)** TRU **(c)** TRU **(d)** TRU

Figure C.2: Distortion vs. (a) number of sensitive patterns and their total number $|\mathcal{S}|$ of occurrences in W (first two lines on the X axis), and (b) length of sensitive patterns k (and $|\mathcal{S}|$). Total number of τ-lost and τ-ghost patterns vs. (c) length of sensitive patterns k, and (d) length of sensitive patterns k (and $|\mathcal{S}|$). $\frac{x}{y}$ on the top of each bar denotes x τ-lost and y τ-ghost patterns.

reason is that PH needs much time to hide all occurrences of sensitive patterns as subsequences for large strings, particularly when $\tau = 1$, which is needed to reduce the frequency of sensitive patterns (substrings) to zero. On the other hand, TPM required less than a second to process W. Note that the results reported for TPM are the same for all values of r, because r is not an input parameter to TPM.

Impact of r on Data Utility We demonstrate that TPM substantially outperforms PH, for all tested values of r, both in terms of distortion and number of τ-lost and τ-ghost patterns. Specifically, TPM incurred on average 169 (and up to 201) times lower distortion than PH. Also, it created only 1 τ-lost pattern, while PH created at least 29 τ-lost and 421 τ-ghost patterns. The reason that PH gets worse when r increases is because a longer record implies that there are generally more occurrences of sensitive patterns (as subsequences) that PH needs to hide, and this requires more substantial changes to the input data. Note that the results reported for TPM are the same for all values of r, because r is not an input parameter to TPM.



(a) TRU **(b)** TRU **(c)** TRU

Figure C.3: (a) Runtime, (b) distortion, and (c) total number of τ-lost and τ-ghost patterns vs. length of the records of the input dataset to PH. Note that PH did not terminate within 12 hours when $r = 25$.

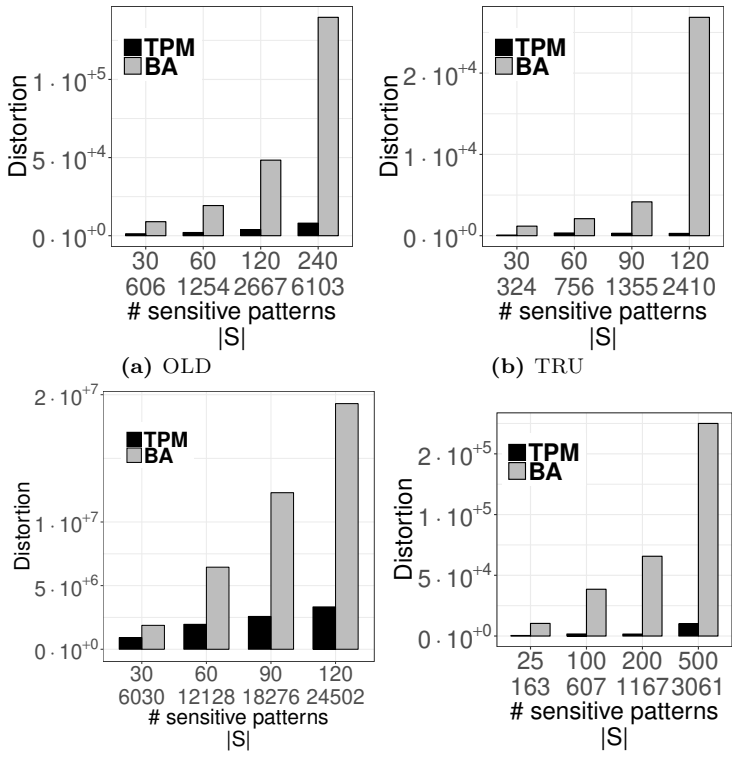
C.8.2 TPM vs. BA

Data Utility We first demonstrate that TPM incurs *very low distortion*. Fig. C.4 shows that, for varying number of sensitive patterns, TPM incurred on average 18.4 (and up to 95) times lower distortion than BA over all experiments. Also, Fig. C.4 shows that TPM remains effective even in challenging settings, with many sensitive patterns (e.g., the last point in Fig. C.4b where about 42% of the positions in W are sensitive). Fig. C.5 shows that, for varying k, TPM caused on average 7.6 (and up to 14) times lower distortion than BA over all experiments.

Next, we demonstrate that TPM permits *accurate frequent pattern mining*: Fig. C.6 shows that TPM led to no τ-lost or τ-ghost patterns for the TRU and MSN datasets. This implies no utility loss for mining frequent length-k substrings with threshold τ. In all other cases, the number of τ-ghosts was on average 6 (and up to 12) times smaller than the total number of τ-lost and τ-ghost patterns for BA. BA performed poorly (e.g., up to 44% of frequent patterns became τ-lost for TRU and 27% for DNA). Fig. C.7 shows that, for varying k, TPM led to on average 5.8 (and up to 19) times fewer τ-lost/ghost patterns than BA. BA performed poorly (e.g., up to 98% of frequent patterns became τ-lost for DNA).

We also demonstrate that PFS-ALGO reduces the length of the output string X of TFS-ALGO substantially, creating a string Y that contains *less redundant information* and allows for more efficient analysis. Fig. C.8a shows the length of X and of Y and their difference for $k = 5$. Y was much shorter than X and its length decreased with the number of sensitive patterns, since more substrings had a suffix-prefix overlap of length $k - 1 = 4$ and were removed (see Section C.5). Interestingly, the length of Y was close to that of W (the string before sanitization). A larger k led to less substantial length reduction as shown in Fig. C.8b (but still few thousand letters were removed), since it is less likely for long substrings of sensitive patterns to have an overlap and be removed.

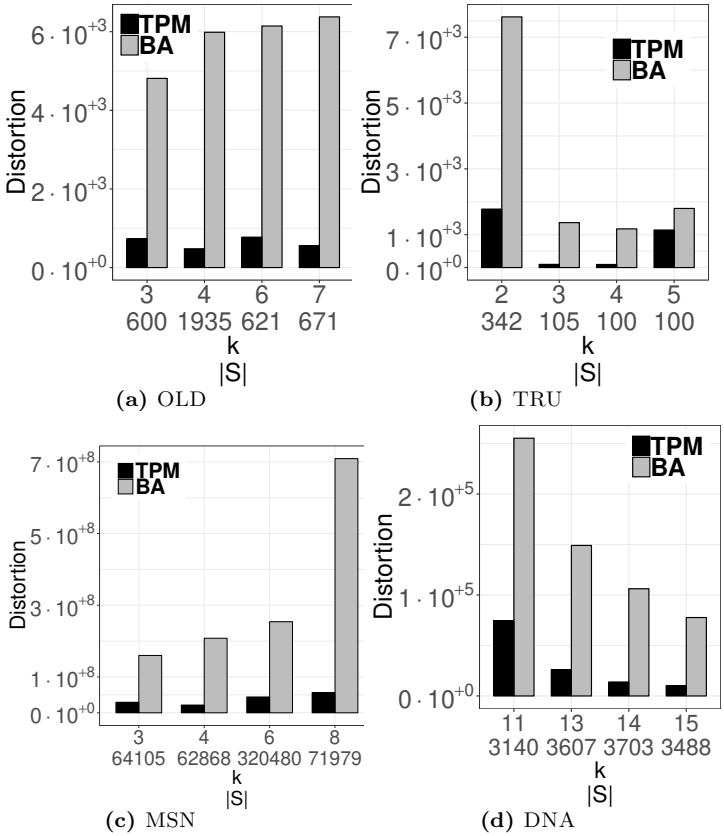
Efficiency We finally measured the runtime of TPM using prefixes of the synthetic string SYN whose length n is 20 million letters. Fig. C.8c (resp., Fig. C.8d) shows



(a) OLD **(b)** TRU

(c) MSN **(d)** DNA

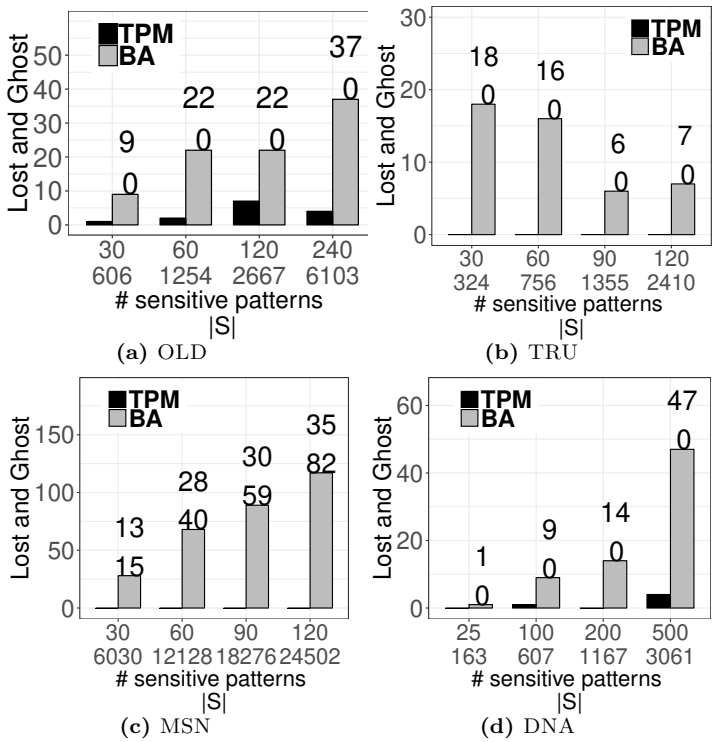
Figure C.4: Distortion vs. number of sensitive patterns and their total number $|\mathcal{S}|$ of occurrences in W (first two lines on the X axis).



(a) OLD **(b)** TRU

(c) MSN **(d)** DNA

Figure C.5: Distortion vs. length of sensitive patterns k (and $|\mathcal{S}|$).



(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Figure C.6: Total number of τ-lost and τ-ghost patterns vs. number of sensitive patterns (and $|\mathcal{S}|$). $\frac{x}{y}$ on the top of each bar for BA denotes x τ-lost and y τ-ghost patterns.

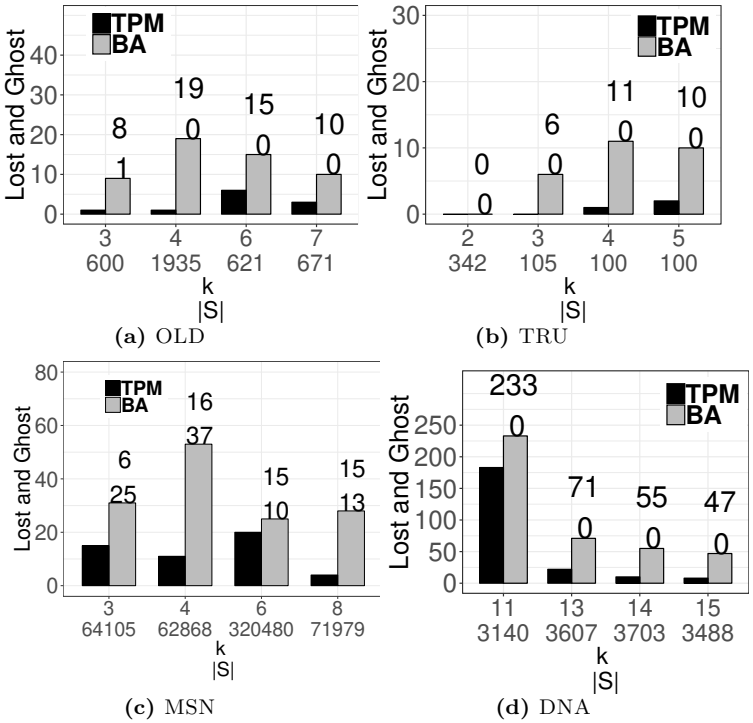


Figure C.7: Total number of τ-lost and τ-ghost patterns vs. length of sensitive patterns k (and $|\mathcal{S}|$). $\frac{x}{y}$ on the top of each bar for BA denotes x τ-lost and y τ-ghost patterns.

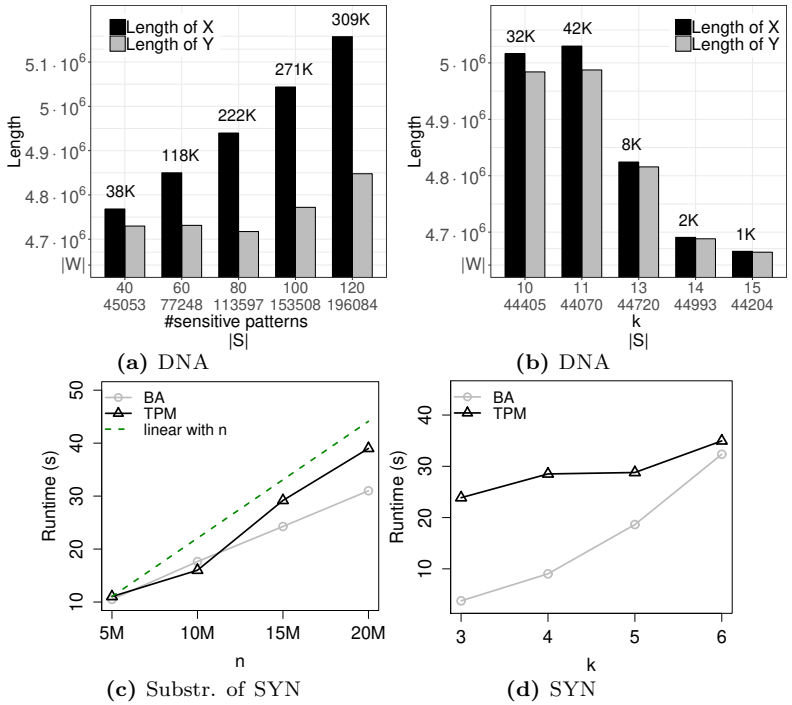
that TPM scaled linearly with n (resp., k), as predicted by our analysis in Section C.6 (TPM takes $\mathcal{O}(n + |Y| + k\delta\sigma + \delta\sigma\theta) = \mathcal{O}(kn + k\delta\sigma + \delta\sigma\theta)$ time, since the algorithm of [295] was used for MCK instances). In addition, TPM is efficient, with a runtime similar to that of BA and less than 40 seconds for SYN.

C.8.3 TM vs. TMI

We compare TM with TMI based on data utility and the number of implausible patterns incurred. The objective of these experiments is to show that TMI is able to produce a string Z that does not contain implausible patterns, while being comparable to TM in terms of the amount of distortion and number of ghost patterns incurred.

We do not report the results of comparing TM with TMI in terms of efficiency, because the runtime of TMI was almost identical to that of TM.

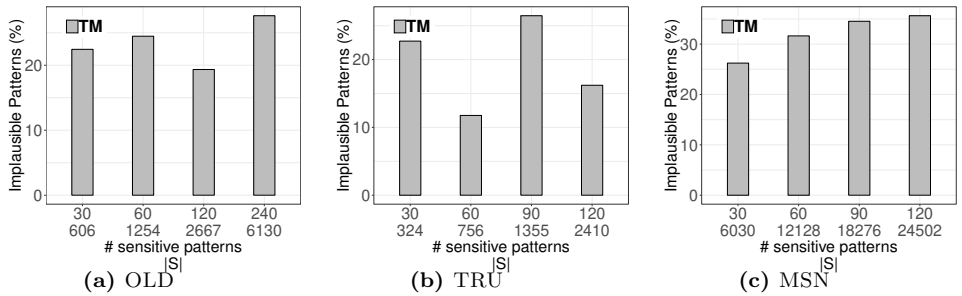
Impact of $|\mathcal{S}|$ We first demonstrate that many implausible patterns may occur as a result of replacing #s with letters, when MCSR is used. This can be seen from Figs. C.9a, C.9b, and C.9c, which show the percentage of implausible patterns incurred by TM, for varying $|\mathcal{S}|$ in OLD, TRU, and MSN, respectively. The percentage is on average 33.08% (and up to 35.63%). The percentage for DNA is 0% (omitted), because



(a) DNA **(b)** DNA

(c) Substr. of SYN **(d)** SYN

Figure C.8: Length of X and Y (output of TFS-ALGO and PFS-ALGO, resp.) for varying: (a) number of sensitive patterns (and $|\mathcal{S}|$), (b) length of sensitive patterns k (and $|\mathcal{S}|$). On the top of each pair of bars we plot $|X| - |Y|$. Runtime on synthetic data for varying: (c) length n of string and (d) length k of sensitive patterns. Note that $|Y| = |Z|$.

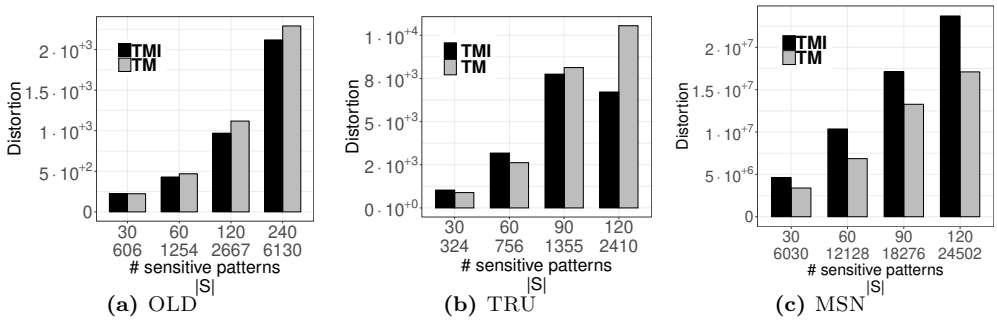


(a) OLD **(b)** TRU **(c)** MSN

Figure C.9: Percentage of implausible patterns vs. number of sensitive patterns (and $|\mathcal{S}|$). The percentages of implausible patterns for DNA are all 0%.

this dataset has a very small alphabet size. Thus, in this experiment, MCSR-ALGO and MCSRI-ALGO are essentially the same algorithm. Since TMI is guaranteed to eliminate implausible patterns, its corresponding percentages are zero (omitted).

We then demonstrate that TMI eliminates implausible patterns without incurring substantial utility loss compared to TM. Figs. C.10 and C.11 show that TMI incurred a comparable amount of distortion to TM. Specifically, TMI incurred 8% and 1% less distortion in the case of OLD and TRU datasets and 37% more distortion in the case of MSN. TMI also incurred a similar number of ghosts than TM. Specifically, TMI incurred 7.1% fewer ghosts in the case of TRU and 54% more ghosts in the case of MSN. Note that no τ-ghost patterns were incurred in the case of OLD (for both TM and TMI). The worse performance of TMI in the case of the MSN dataset is attributed to its relatively small alphabet size, which makes it more difficult to select a letter replacement that does not incur implausible patterns.

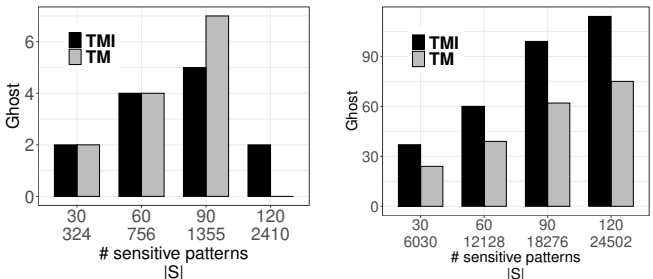


(a) OLD **(b)** TRU **(c)** MSN

Figure C.10: Distortion vs. number of sensitive patterns and their total number $|\mathcal{S}|$ of occurrences in W (first two lines on the X axis).

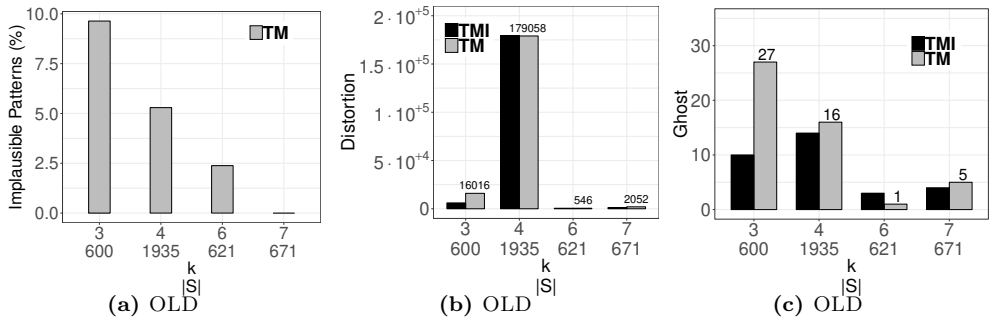
Impact of k Fig. C.12a shows that the percentage of implausible patterns incurred by TM for the OLD dataset was on average 4.3% (and up to 9.6%). Again, this confirms the need to eliminate implausible patterns in practice. The results for TRU, MSN, and DNA are qualitatively similar and omitted from all remaining experiments.

We now demonstrate that TMI eliminates implausible patterns, while incurring a comparable amount of distortion and ghosts (on average) compared to TM. Specifically,



(a) TRU **(b)** MSN

Figure C.11: Number of τ-ghost patterns (the number of τ-lost patterns is zero by design) vs. number of sensitive patterns (and $|\mathcal{S}|$). The number of τ-ghost patterns for OLD is 0.



(a) OLD **(b)** OLD **(c)** OLD

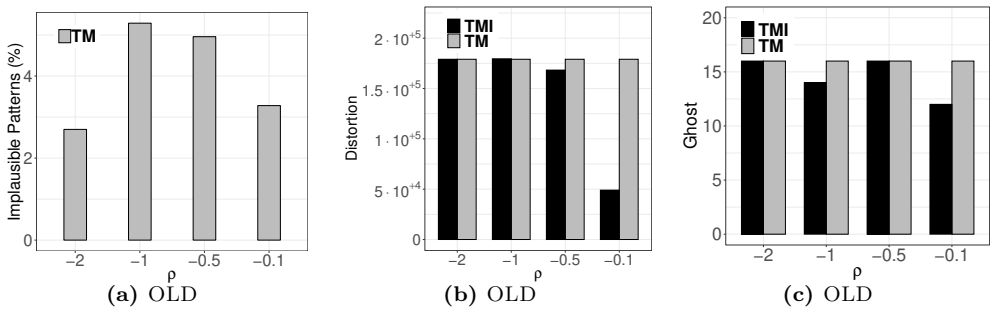
Figure C.12: (a) Percentage of implausible patterns vs. k (and $|\mathcal{S}|$). (b) Distortion vs. k (and $|\mathcal{S}|$). (c) Number of τ-ghost patterns vs. k (and $|\mathcal{S}|$).

the distortion for TMI was 17% lower than TM on average (see Fig. C.12b), and the number of τ-ghost patterns for TMI was 16.2% lower on average (see Fig. C.12c).

Impact of ρ We demonstrate that TMI can eliminate implausible patterns, while preserving data utility as well as TM does. This can be seen from Fig. C.13a, which shows that the percentage of implausible patterns incurred by TM was 4.1% on average (and up to 5.3%), and from Figs. C.13b and C.13c, which show that TMI caused on average 19.5% lower distortion and 9.4% fewer τ-ghosts, respectively, compared to TM.

C.8.4 TFS-ALGO vs. ETFS-ALGO

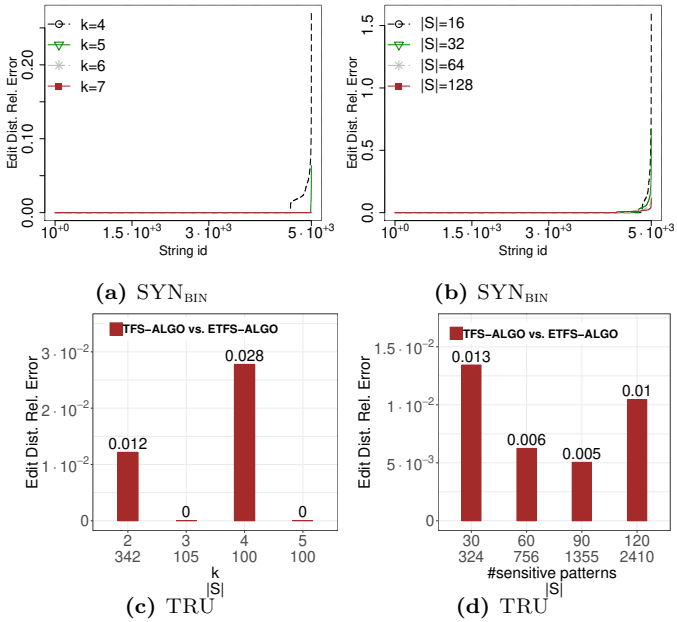
We demonstrate that TFS-ALGO is a very effective heuristic for the ETFS problem. Specifically, it constructs a string X that is either an optimal solution to the problem or it is at slightly larger edit distance from W compared to the exact solution string X_{ED} that is constructed by ETFS-ALGO. This can be seen from Fig. C.14a (resp., C.14b), which shows that TFS-ALGO constructed optimal solutions (i.e., Edit Distance Relative Error was 0) in 98% (resp., 93%) of the tested strings, on average. These strings are uniformly random and have the same length and alphabet as SYN_{BIN}. Qualitatively similar results were obtained for uniformly random strings of different lengths and alphabet sizes (omitted). In addition, the effectiveness of TFS-ALGO can be seen



(a) OLD **(b)** OLD **(c)** OLD

Figure C.13: (a) Distortion, (b) number of τ-ghost patterns, and (c) percentage of implausible patterns vs. ρ.

from Figs. C.14c and C.14d, which show that the Edit Distance Relative Error in TRU was no more than 2.8%. These results are encouraging because, unlike ETFS-ALGO, TFS-ALGO is applicable to large strings such as OLD, MSN, and DNA (recall that its time complexity is linear instead of quadratic in $|W|$).



(a) SYN_{BIN} **(b)** SYN_{BIN}

(c) TRU **(d)** TRU

Figure C.14: Edit Distance Relative Error vs. (a) k (and $|\mathcal{S}|$), and (b) number of sensitive patterns (and $|\mathcal{S}|$) for each of the 50,000 random strings. Edit Distance Relative Error vs. (c) k (and $|\mathcal{S}|$), and (d) number of sensitive patterns (and $|\mathcal{S}|$) for TRU.

Appendix D

String Sanitization Under Edit Distance

Key Points

Problem. We consider again the problem of sanitizing a string by concealing the occurrences of sensitive patterns, while maintaining data utility.

Model. Like in the previous chapter, we seek for a string, to be disseminated in place of the original one, that preserves the order of appearance and frequency of all non-sensitive patterns, while sensitive patterns are concealed with the aid of an extra alphabet letter. We consider the setting in which we aim to construct a string that is at minimal edit distance from the original string, in addition to preserving the order of appearance and frequency of all non-sensitive patterns. We manage to improve the algorithm presented in the previous chapter and we reduce the edit distance problem, which is known to admit the same conditional lower bound, to our problem.

Included Works

This chapter presents the work **String Sanitization Under Edit Distance** [56], that improves the results of the previous chapter. This paper has been presented at the *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*.

D.1 Introduction

In Chapter C we introduced the *Combinatorial String Dissemination* (CSD) model to provide privacy and utility guarantees. In CSD, we are given a string W and the aim is to apply a sequence of edit operations to W, so that the resulting counterpart X of W satisfies a set of *privacy constraints* and a set of *utility properties*. Specifically, in this chapter we consider the following CSD problem, referred to as ETFS (Edit distance,

Total order, Frequency, Sanitization). Given W of length n over an alphabet Σ, a positive integer k, and a set $\mathcal{S}$ of *sensitive* length-k substrings of W modeling confidential information, construct a string X such that: X does not contain any sensitive length-k substring (**C1**); the order (and thus the frequency) of all other length-k substrings over Σ in W is the same as in X (**P1**); and X is at *minimal edit distance* from W. The constraint **C1** ensures that no sensitive length-k substring occurs in X. The property **P1** ensures that X incurs minimal utility loss for tasks based on the sequential nature of length-k non-sensitive substrings of W, as well as on their frequency.

Strings constructed by means of solving ETFS can be used, with minimal utility loss, in tasks that are based on edit distance as a similarity measure. Examples of such tasks are frequent pattern mining [321], clustering [218], entity extraction [358] and range query answering [257]. To solve ETFS, in Chapter C we proposed an $\mathcal{O}(k|\Sigma|n^2)$-time algorithm. Such algorithm is based on solving a specific instance of approximate regular expression matching, essentially applying the algorithm of Myers and Miller [279] on an appropriate regular expression that models all strings satisfying **C1** and **P1** to finally pick the one that is at minimal edit distance to W.

Note that, to have a solution to ETFS, we may need to insert in W a letter $\# \notin \Sigma$. Indeed, inserting (or replacing letters of W with) any letter of Σ could violate **P1** and/or possibly create new occurrences of sensitive length-k substrings. We thus generally have that X_{ED} (a solution of ETFS) is over $\Sigma \cup \{\#\}$.

Example 31. *Let* $W = \mathtt{babaaaaabbbab}$, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $k = 3$, *and the set of sensitive substrings be* $\{\mathtt{aba}, \mathtt{baa}, \mathtt{aaa}, \mathtt{aab}, \mathtt{bba}\}$. *Then* $X_{\mathrm{ED}} = \mathtt{bab\#aa\#abbb\#bab}$. *Note that* X_{ED} *satisfies* ***C1*** *and* ***P1***. X_{ED} *is a string* closest *to* W *in terms of edit distance.*

In Section D.2, we show the following theorem improving the result of Chapter C by a factor of $|\Sigma|$.

Theorem 34. *The* ETFS *problem can be solved in* $\mathcal{O}(kn^2)$ *time.*

Our algorithm is based on a non-trivial modification of the classic dynamic programming algorithm for computing the edit distance between two given strings. In particular, the modification is based on the fact that in ETFS we are given a *single* string W, and we are asked to *construct* a string X_{ED} that satisfies **C1** and **P1** and that is closest to W. We thus actually fill in the dynamic programming matrix that computes the minimum edit distance between W and a regular expression that is a suitable abstraction of X_{ED}; our algorithm encodes in its recurrence formulae the choices that specify the instance of the regular expression that we eventually output.

In Section D.3, we also show that ETFS cannot be solved in strongly subquadratic time unless the Strong Exponential Time Hypothesis (SETH) [208, 207] is false. This is the most technically involved part of the chapter.

Theorem 35. *The* ETFS *problem cannot be solved in* $\mathcal{O}(n^{2-\delta})$ *time, for any* $\delta > 0$, *unless SETH is false.*

To arrive at this theorem, we reduce the weighted edit distance problem, which is known to admit the same conditional lower bound [38, 80], to the ETFS problem. In particular, given two strings P and Q of length $\Theta(n)$, we construct an instance of ETFS of length $\mathcal{O}(n)$ from the output of which we can infer the insertions corresponding to

some optimal alignment of P and Q with respect to the weighted edit distance. Using another suitable instance of ETFS, we can determine the corresponding deletions. That gives us an optimal alignment of P and Q, from which we can compute the weighted edit distance of P and Q in $\mathcal{O}(n)$ time.

D.2 ETFS-DP: An $\mathcal{O}(kn^2)$-time Algorithm for ETFS

Refer to Section C.3 for the definitions of this chapter. We report here for convenience the formal definition of ETFS.

Problem 5 (ETFS). *Given a string W of length n, an integer $k > 1$, and a set $\mathcal{S}$ (and thus set $\mathcal{I}$), construct a string X_{ED} which is at minimal (weighted) edit distance from W and satisfies the following:*

C1 *X_{ED} does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_{X_{ED}}$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_{X_{ED}}$ are equivalent.*

In this section we describe ETFS-DP, a dynamic programming algorithm that solves ETFS faster than the algorithm proposed in Chapter C. We describe our algorithm for the unweighted edit distance model for simplicity, but it should be clear that it can be extended to the weighted edit distance model in a straightforward way and with no additional cost. Intuitively, since we are looking for a string X_{ED} that contains all the non-sensitive patterns of W, and in the same order, for each pair (U, V) of non-sensitive patterns of W such that U is the t-predecessor of V, we can (i) *merge* U and V into $U \cdot V[k-1]$ when U and V have a suffix-prefix overlap of length $k-1$; or (ii) *interleave* U and V constructing a string UYV, where Y is a carefully selected string over $\Sigma \cup \{\#\}$, where $\# \notin \Sigma$.

Let us start by reporting the definition of the regular expression gadget $\oplus$, which encodes all candidate strings that can be used to interleave two non-sensitive patterns while respecting **C1**, and the two similar gadgets $\ominus$ and $\otimes$. We will make use of the following regular expression:

$$\Sigma^{<k} = (\underbrace{(a_1|a_2|\ldots|a_{|\Sigma|}|\varepsilon)\ldots(a_1|a_2|\ldots|a_{|\Sigma|}|\varepsilon)}_{k-1 \text{ times}}),$$

where $\Sigma = \{a_1, a_2, \ldots, a_{|\Sigma|}\}$ is the alphabet of W. Given a letter $\# \notin \Sigma$, we define

$$\oplus = \#(\Sigma^{<k}\#)^*, \quad \ominus = (\Sigma^{<k}\#)^*, \quad \otimes = (\#\Sigma^{<k})^*.$$

Let N_0, N_1, …, $N_{|\mathcal{I}|-1}$ be the sequence of non-sensitive patterns as they occur in W from left to right. In the algorithm of Chapter C, X_{ED} was built by finding an optimal alignment between W and a regular expression R constructed as follows. First, set $R = \ominus N_0$ and then process pairs of non-sensitive patterns N_i and N_{i+1}, for all $i \in \{1, \ldots, |\mathcal{I}| - 2\}$: in the i-th step, if N_i and N_{i+1} can be merged, append $(N_{i+1}[k-1] \mid \oplus N_{i+1})$ to R. Otherwise, append $\oplus N_{i+1}$ to R. After processing all pairs, conclude by appending $\otimes$ to R. The length of R is $\mathcal{O}(k|\Sigma|n)$.

The general idea in Algorithm ETFS-DP is to simulate the alignment of W to R without constructing R explicitly. Instead, we use a string $T = \boxminus N_0 \boxplus N_1 \cdots \boxplus N_{|\mathcal{I}|-1} \boxtimes$,

where $\boxminus$, $\boxplus$ and $\boxtimes$ are length-1 *placeholders* for $\ominus$, $\oplus$ and $\otimes$, respectively. The length of T is thus only $(k+1)|\mathcal{I}|+1 = \mathcal{O}(kn)$, leading to an $\mathcal{O}(kn^2)$-time algorithm when aligned to W, $|W| = n$.

D.2.1 Dynamic Programming

In a preprocessing phase, we compute a binary array M of length $|\mathcal{I}|$ so that $M[\ell] = 1$ if the ℓ-th and the $(\ell-1)$-th non-sensitive patterns (in the order given by their occurrences in W) can be merged. We set $M[0] = 0$ for completeness. More formally, for all $0 < \ell \leq |\mathcal{I}| - 1$, $M[\ell] = 1$ if $N_{\ell-1}[1\,..\,k-1] = N_\ell[0\,..\,k-2]$, and $M[\ell] = 0$ otherwise.

We then solve ETFS in a dynamic programming fashion by filling in an $(|\mathcal{I}|(k+1)+1) \times (|W|+1)$ matrix E. The rows of E correspond to string T, and the columns to string W. We denote by $E[i][\cdot]$ and $E[\cdot][j]$ the i-th row and the j-th column of E, respectively.

Entry $E[i][j]$, for all $0 \leq i \leq |\mathcal{I}|(k+1)$ and $0 \leq j \leq |W|$, contains the edit distance between (the regular expression corresponding to) $T[0\,..\,i]$ and $W[0\,..\,j-1]$. Rows corresponding to $\boxplus$, i.e., rows with index $i = \ell(k+1)$ for some $\ell \in [1, |\mathcal{I}| - 1]$, implicitly represent a regular expression gadget and must be filled in with ad hoc rules; we will refer to them as *gadgets rows*. In turn, we will name *possibly mergeable* the rows with index $i = \ell(k+1) - 1$ for some $\ell \in [1, |\mathcal{I}| - 1]$, as they must be filled in taking into account the option of merging the corresponding pattern with the preceding one, should it be possible. All other rows of E will be called *ordinary*. In what follows, I is a function such that $I[T[i] \neq W[j-1]] = 1$ if $T[i] \neq W[j-1]$, and 0 otherwise. We give below the recursive formulae that constitute the core of our dynamic programming algorithm.

Initialization. Entry $E[0][j]$ contains the edit distance between $\ominus$ and $W[0\,..\,j-1]$ for $j \geq 1$, while $E[0][0] = 0$. Because of the definition of $\ominus$, it is only possible to match up to $k-1$ consecutive letters, after which a mismatch due to `#` occurs, and hence $E[0][j] = \lceil j/k \rceil$.

$E[i][0]$ stores the edit distance between $T[0\,..\,i]$ and the empty prefix ε of W. This distance is minimized by the shortest possible string in each regular expression prefix, obtained by always merging when allowed, and picking the shortest possible string encoded by $\oplus$ when not. This leads to the following formula, where $\ell \in [0, |\mathcal{I}| - 1]$.

$$E[i][0] = \begin{cases} E[i-k-1][0] + 1, & \text{if } i = (\ell+1)(k+1) - 1 \wedge M[\ell] = 1 \text{ (merge)} \\ E[i-1][0] + 1, & \text{otherwise (no merge)} \end{cases} \tag{D.1}$$

Ordinary Rows: $i \not\equiv 0 \mod (k+1)$ and $i \not\equiv -1 \mod (k+1)$. The formula is the same as in the standard algorithm for edit distance [245]: recall that $E[\cdot][j]$

correspond to $W[j-1]$.

$$E[i][j] = \min \begin{cases} E[i-1][j]+1, & \text{(insert)} \\ E[i][j-1]+1, & \text{(delete)} \\ E[i-1][j-1] + I[T[i] \neq W[j-1]], & \text{(match or substitute)} \end{cases} \tag{D.2}$$

Possibly Mergeable Rows: $i \equiv -1 \mod (k+1)$. These rows correspond to the last letter of a non-sensitive pattern. The first three options of Equation D.3 encode the case where we do not merge, regardless of the value of $M[\ell]$. The last two options, instead, require $M[\ell] = 1$, as a merge does take place. This means that the letters corresponding to the k rows above will not appear in the output string X_{ED}, and thus play no role in the edit distance computation. We thus read the values of row $i-k-1$, corresponding to the last letter of the previous non-sensitive pattern.

$$E[i][j] = \min \begin{cases} E[i-1][j]+1, & & \text{(insert)} \\ E[i][j-1]+1, & & \text{(delete)} \\ E[i-1][j-1] + I[T[i] \neq W[j-1]], & & \text{(match or substitute)} \\ E[i-k-1][j]+1, & \text{if } M[\ell]=1 & \text{(insert and merge)} \\ E[i-k-1][j-1] + I[T[i] \neq W[j-1]], & \text{if } M[\ell]=1 & \text{(match or sub. and merge)} \end{cases} \tag{D.3}$$

Gadget Rows: $i \equiv 0 \mod (k+1)$. A gadget row encodes the possibility of interleaving two non-sensitive patterns with a string that preserves **C1** and **P1** and minimizes the edit distance. Because of the form of the regular expression gadgets, a `#` can either be inserted or substituted directly after a non-sensitive pattern, or be preceded by another `#` no more than k positions earlier. This results in the following formula:

$$E[i][j] = \min \begin{cases} E[i-1][j]+1, & \text{(insert)} \\ E[i-1][j-1]+1, & \text{(substitute)} \\ E[i][j-1]+1, \ldots, E[i][\max\{0, j-k\}]+1, & \text{(delete or extend gadget)} \end{cases} \tag{D.4}$$

The following lemma states that the above formulae correctly compute the edit distance between prefixes of T and prefixes of W.

Lemma 57. *$E[i][j] = d_E(T[0\,..\,i], W[0\,..\,j-1])$, for all $0 \leq i < |\mathcal{I}|(k+1)$ and $0 < j \leq |W|$, and $E[i][0] = d_E(T[0\,..\,i], \varepsilon)$.*

Proof. The correctness of the equations that describe how to fill in entries $E[0][j]$ and $E[i][0]$ follows from the explanation in paragraph "Initialization", and the correctness of Equation D.2 follows from the standard dynamic programming algorithm for edit distance [245]. Let us focus on the case of possibly mergeable rows (Equation D.3): when merging is not possible, the equation is the same as in the standard algorithm, and therefore it is correct. When merging is possible, we must pick the minimum

value among all possible edit operations when we actually choose to merge and among all possible operations when we do not merge, even if we could. The first three rows of Equation D.3 correspond to the three possible operations when we do not merge, and are again the same possibilities as the standard algorithm for edit distance; the last two rows correspond to the case where we merge. When we merge, we append the letter corresponding to the possibly mergeable row to the previous non-sensitive pattern. If we were to run the standard algorithm for computing the edit distance between such string and W, the row above, where we had to read the values for insertion and match or substitution, would be the one corresponding to the last letter of the previous non-sensitive pattern. These are precisely the values of the last two rows of Equation D.3, that are therefore correct.

Consider now the gadget rows. An entry $E[i][j]$ on a gadget row should contain the value of an optimal alignment between $W[0\,..\,j-1]$ and a prefix of X_{ED} that ends with a #: since $\# \notin \Sigma$, it cannot match with any letter of W, therefore $I[T[i] \neq W[j-1]] = 1$ always holds. As previously observed, a # can either be inserted or substituted directly after a non-sensitive pattern, or be preceded by another # no more than k positions earlier. Moreover, it is easy to see that, if an optimal alignment between W and the regular expression R involves a local alignment between $W[i\,..\,j]$ and $\#S\#$ with $|S| = j - i - 1 < k$, then $S = W[i+1\,..\,j-1]$: this is because any alignment with $S \neq W[i+1\,..\,j-1]$ can be improved by replacing S with $W[i+1\,..\,j-1]$. Equation D.4 follows from the two observations above: the first two lines compute the cost of appending a # directly after a non-sensitive pattern, that always entails either an insertion or a substitution.

The third row of the equation considers the possibility of interleaving two non-sensitive patterns with a whole string encoded by $\oplus$, or deleting #. □

Note that Lemma 57 refers to rows $0 \leq i < |\mathcal{I}|(k+1)$. Let us now look at the last row: even if it was filled in like any other gadget row, since it corresponds to $\otimes$ instead of $\oplus$, its values need to be interpreted in a different way. Namely, the value stored in $E[|\mathcal{I}|(k+1)][j]$, for all $0 \leq j \leq |W|$, is the cost of an optimal alignment between $W[0\,..\,j+e_j-1]$ and a string in R whose length-(e_j+1) suffix is $\#W[j\,..\,j+e_j-1]$, where $e_j = \min\{k-1, |W|-j\}$.

Unlike in the standard edit distance algorithm [245], the edit distance between W and any string matching the regular expression R is not necessarily found in its bottom-right entry $E[|\mathcal{I}|(k+1)][|W|]$. Instead, it is found among the rightmost k entries of the last row (in case X_{ED} ends with a string in $\otimes$), and the rightmost entry of the second-last row (when X_{ED} ends with the last letter of the last non-sensitive pattern). We thus obtain the following.

Lemma 58. *Let X_{ED} be a solution to* ETFS. *Then*

$$d_E(X_{\mathrm{ED}}, W) = \min\big\{E\big[|\mathcal{I}|(k+1)-1\big]\big[|W|\big], E\big[|\mathcal{I}|(k+1)\big]\big[|W|\big], \\ E\big[|\mathcal{I}|(k+1)\big]\big[|W|-1\big], \ldots, E\big[|\mathcal{I}|(k+1)\big]\big[|W|-k+1\big]\big\}. \tag{D.5}$$

D.2.2 Construction of X_{ED}

Once we have computed the edit distance d according to Lemma 58, we need to construct a string X_{ED} that matches R and is at edit distance d from W. To do so,

when computing each entry $E[i][j]$ of the matrix for $i, j \geq 1$, we store, in an array A, a *pointer* $\langle i', j' \rangle$ to an entry from which the minimum value for $E[i][j]$ was obtained. We then build X_{ED}^R by following any path from an entry $E[\bar{\imath}][\bar{\jmath}]$ where the global optimum is stored to $E[0][0]$.

At any step of the construction, let $E[i'][j']$ be the endpoint of the pointer stored for $E[i][j]$ currently considered, i.e., $A[i][j] = \langle i', j' \rangle$. If $\bar{\imath} = |\mathcal{I}|(k+1)$, i.e., if the minimum is in the last row of E, we initialize X_{ED}^R with $W[\bar{\jmath}\,..\,|W|-1]^R$; otherwise, we just initialize it with the empty string ε. We then enforce the following rules:

If $\mathbf{i' < i}$, we append $T[i]$ to X_{ED}^R when i is not a gadget row and `#` otherwise. Indeed, the condition is fulfilled when the edge in the path is either diagonal (a match or a substitution in the alignment) or vertical (an insertion in W). Moreover, i' can either be equal to $i-1$ or to $i-k-1$ (when we merge two non-sensitive patterns).

If $\mathbf{i' = i}$ and $\mathbf{i \equiv 0 \mod (k+1)}$, we append `#` to X_{ED}^R followed by $W[j'\,..\,j-2]^R$. Because this happens when we follow a horizontal edge on a gadget row, the solution must include the corresponding substring, that is composed of `#` and $j - j' - 1$ letters of W.

If none of the two cases above happens, we do not write anything, because a horizontal edge in the path corresponds to a deletion in W. We denote the above procedure by Algorithm X_{ED}-construct. Lemma 59 guarantees that this construction produces a string that satisfies **C1** and **P1**.

Lemma 59. *X_{ED} returned by Algorithm X_{ED}-construct satisfies **C1** and **P1**.*

Proof. Let us start by proving that Algorithm X_{ED}-construct satisfies **C1**. X_{ED}^R (and thus X_{ED}) is obtained by appending either consecutive letters of T^R (case $i' < i$ for all but gadget rows) or a letter `#` (all cases for gadget rows) or a number of consecutive letters of W^R (case $i' = i$ for gadget rows and initialization of X_{ED}^R when the minimum is on the last row of E): since T does not contain any sensitive patterns by construction and $\texttt{\#} \notin \Sigma$, we only need to verify that no more than $k-1$ consecutive letters read directly from W^R can ever be appended to X_{ED}^R. Inspect case $i' = i$ for gadget rows: $j - j' - 1$ is the number of entries between entry $E[i][j]$ and the endpoint of the corresponding horizontal pointer $A[i][j]$. The last line of Equation D.4 exhibits the only possibilities for a pointer to point a non-adjacent entry on the same row, thus $j' \geq j - k$ and consequently $j - j' - 1 \leq k - 1$. Since both when the path leaves a gadget row and when it goes on on a gadget row a `#` is appended to X_{ED}^R, no sensitive patterns can be created and therefore X_{ED} satisfies **C1**.

Let us now show that **P1** is satisfied as well, i.e., N_0, N_1, $\ldots$ $N_{|\mathcal{I}|-1}$ occur in X_{ED} in the same order as they appear in W, and no other length-k string over Σ is a substring of X_{ED}. Consider a letter $N_\ell[h] = T[i]$. If $0 \leq h < k-1$, i is an ordinary row. Since any optimal path goes from the entry of E where the minimum is stored to $E[0][0]$, and by construction to leave a row the pointers can only point to an entry in the row directly above the current one (ordinary rows) or in the $(k+1)$-th row above (merge case in the possibly mergeable rows, see Equations D.2 and D.3), there are only two possibilities: either the path goes through row i, i.e., there exists j such that $A[i][j] = \langle i-1, j' \rangle$ is part of the optimal path, or row i is skipped by the path, and thus there exists j

such that $A[i+k-h-1][j] = \langle i-h-2, j' \rangle$. Let us observe that in the latter case, all of the rows from $i-h-1$ to $i+k-h-2$ are skipped by the path, while in the first case $A[i+k-h-1][j] = \langle i+k-h-2, j' \rangle$ and no rows are skipped up to $i-h-2$. In the first case, Algorithm X_{ED}-construct will append $N_\ell[h]$ to X^R_{ED} after $N_\ell[h+1]$ for all $0 \leq h \leq k-1$, then a # right after $N_\ell[0]$, that prevents the making of spurious length-k strings over Σ in X_{ED}. In the second case, $N_\ell[h]$ is not explicitly appended to the string: instead, after appending $N_\ell[k-1]$ to X^R_{ED}, the algorithm goes to row $i-h-2$, corresponding to $N_{\ell-1}[k-1]$. Nevertheless, this only happens when the merge condition is satisfied, i.e., when $N_{\ell-1}[1\,..\,k-1] = N_\ell[0\,..\,k-2]$, implying that $N_\ell[h] = N_{\ell-1}[h+1]$ will be appended next to $N_\ell[h+1] = N_{\ell-1}[h+2]$ after $k-h-1$ steps. The order in which N_0, N_1, $\ldots N_{|\mathcal{I}|-1}$ appear in X_{ED} is by construction the same as they appear in T, which in turn is the same as the order they appear in W. In no other parts of the algorithm a length-k string over Σ is created in X_{ED}. It follows that **P1** is preserved. □

D.2.3 Wrapping up

Lemma 60. *Algorithm* ETFS-DP *runs in* $\mathcal{O}(kn^2)$ *time.*

Proof. We first construct string T and array M in $\mathcal{O}(kn)$ time and initialize the first row and the first column of matrix E in $\mathcal{O}(kn)$ time. There are $\mathcal{O}(kn)$ "ordinary", $\mathcal{O}(n)$ "possibly mergeable" and $\mathcal{O}(n)$ "gadget rows", each of size $\mathcal{O}(n)$. Each entry (and its corresponding pointer) on the "ordinary" and "possibly mergeable" rows takes constant time to compute, while the entries (and pointers) on the gadget rows require $\mathcal{O}(k)$ time each. Thus, we can compute all entries and pointers in $\mathcal{O}(kn^2)$ time. Tracing back the pointers and constructing string X_{ED} takes again $\mathcal{O}(kn)$ time. This results in a total time complexity of $\mathcal{O}(kn^2)$. □

Lemmas 57-60 imply Theorem 34.

D.3 A Conditional Lower Bound for ETFS

We prove that, assuming SETH introduced in [208] and [207], ETFS cannot be solved in strongly subquadratic time. We do so by a reduction from the classical edit distance problem, and using the following known conditional lower bound for it: for all $\delta > 0$, the edit distance d_E between two strings of length $\Omega(n)$ cannot be computed in $\mathcal{O}(n^{2-\delta})$ time without violating SETH [38], and hence the well-known quadratic-time solution of [245] for computing the edit distance between two strings of length $\mathcal{O}(n)$ is optimal up to subpolynomial factors. Bringmann and Künnemann [80] proved that this is also true for weighted edit distance, where each operation (insertion, deletion, substitution and match) has a corresponding fixed non-negative cost (respectively $c_{\mathrm{i}}, c_{\mathrm{d}}, c_{\mathrm{s}}, c_{\mathrm{m}}$), and the following conditions, which we will call the BK conditions, hold: (i) $c_{\mathrm{i}} + c_{\mathrm{d}} > c_{\mathrm{m}}$, (ii) $c_{\mathrm{i}} + c_{\mathrm{d}} > c_{\mathrm{s}}$, and (iii) $c_{\mathrm{m}} \neq c_{\mathrm{s}}$.

Let P and Q be two arbitrary strings over Σ, both of length $\Theta(n)$, and without loss of generality $1 \leq |P| \leq |Q|$. We would like to compute the weighted edit distance between P and Q with the following associated costs: $c_{\mathrm{i}} = 2.5, c_{\mathrm{d}} = 2.5, c_{\mathrm{s}} = 1$, $c_{\mathrm{m}} = 0$.

These costs satisfy the BK conditions. Let $c = (c_{\mathrm{i}}, c_{\mathrm{d}}, c_{\mathrm{s}}, c_{\mathrm{m}})$ and d_c be the weighted edit distance with associated costs c. Assuming SETH is true, there is no algorithm for computing $d_{(2.5,2.5,1,0)}(P,Q)$ in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$ [80]. In order to prove that ETFS cannot be solved in strongly subquadratic time either, we will compute $d_{(2.5,2.5,1,0)}(P,Q)$, by solving two instances of ETFS on a string of length $\mathcal{O}(n)$ and using an additional $\mathcal{O}(n)$ number of operations. Thus if ETFS is solvable in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$, SETH is false.

Let us now show the first instance of the ETFS. We define a new alphabet $\Sigma' = \Sigma \sqcup \{\mathtt{a}, \mathtt{b}, \mathtt{c_1}, \mathtt{c_2}, \mathtt{c_3}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}\}$ and a new string $U(P,Q) = F_1F_2F_3F_4$ over Σ' as follows:

$$F_1 = (\mathtt{aab})^{2x+1}\mathtt{aae}, \quad F_2 = \prod_{i=0}^{|P|-1} \mathtt{c_1 d}P[i]\mathtt{c_2c_3}, \quad F_3 = (\mathtt{aae})^{2x-1}\mathtt{aa}, \quad F_4 = \prod_{i=0}^{|Q|-1} \mathtt{c_1 f}Q[i]\mathtt{c_2c_3}$$

where $x = 2|Q|$, and the product denotes the concatenation operation on strings. We also set $k = 5$ and define the set $\mathcal{I}$ of non-sensitive pattern occurrences over U as follows:

$$\mathcal{I} \;=\; \{0, 3, 6, 9, \ldots, 6x\} \;\cup\; \{6x+6, 6x+11, 6x+16, \ldots, 6x+1+5|P|\}.$$

In particular, $U(P,Q)$ is the string input to the first instance of ETFS. The construction above gives us the following sequence of *non-sensitive* patterns:

$$\begin{aligned} &\mathtt{aabaa}, \mathtt{aabaa}, \mathtt{aabaa}, \ldots, \mathtt{aabaa} && (2x+1 \text{ occurrences}) \\ &\mathtt{c_1 d}P[0]\mathtt{c_2c_3}, \mathtt{c_1 d}P[1]\mathtt{c_2c_3}, \mathtt{c_1 d}P[2]\mathtt{c_2c_3}, \ldots, \mathtt{c_1 d}P[|P|-1]\mathtt{c_2c_3} && (|P| \text{ occurrences}). \end{aligned}$$

It is easy to verify that the set $\mathcal{I}$ of occurrences of non-sensitive patterns (and thus the complementary set $\mathcal{S}$) has the closure property requested by ETFS. The resulting regular expression R is

$$R = \ominus\, \mathtt{aabaa} \oplus \mathtt{aabaa} \oplus \ldots \oplus \mathtt{aabaa} \oplus \mathtt{c_1 d}P[0]\mathtt{c_2c_3} \oplus \mathtt{c_1 d}P[1]\mathtt{c_2c_3} \oplus \ldots \oplus \mathtt{c_1 d}P[|P|-1]\mathtt{c_2c_3} \otimes.$$

We will prove that it is optimal to align the first $x+1$ patterns with F_1, a gadget $\oplus$ with F_2, the next x patterns with F_3 and the final $|P|$ patterns with F_4. Then, we will show that the alignment of those last patterns with F_4 corresponds to an alignment of P and Q.

We call the occurrences of $\mathtt{aabaa}$ and $\mathtt{c_1 d}P[i]\mathtt{c_2c_3}$ in the regular expression R, or in any string in the regular language corresponding to R, *AB-patterns* and *P-patterns*, respectively. Notice that these non-sensitive patterns are substrings of F_1 and F_2 and that we cannot merge any two consecutive non-sensitive patterns.

Recall that the output X_{ED} of ETFS is a string with minimal edit distance to U in that language. One alignment of U and R, which we denote by $\mathcal{A}_{U/R}$ and that we will later show to be optimal under unit cost for insertion, deletion and substitution and zero cost for match, is as follows:

- We align F_1 with the first $x+1$ AB-patterns interleaved by `#`'s as illustrated below. The cost of this alignment is $x+1$ substitutions.

  ```
  aabaabaabaabaabaabaabaab...aabaae
  aabaa#aabaa#aabaa#aabaa#...aabaa#
  ```

- We align F_2 with a single gadget $\oplus$ suitably expanded as shown below. The cost of this alignment is $|P|$ substitutions. Recall that we have to use a `#` after every $k-1=4$ letters, so as not to introduce any new length-k substrings that would violate property **P1**.

  $\mathtt{c_1d}P[0]\mathtt{c_2c_3c_1d}P[1]\mathtt{c_2c_3c_1d}P[2]\mathtt{c_2c_3c_1d}P[3]\mathtt{c_2c_3}\ldots\mathtt{c_1d}P[|P|-1]\mathtt{c_2c_3}$
  $\mathtt{c_1d}P[0]\mathtt{c_2\#}\ \mathtt{c_1d}P[1]\mathtt{c_2\#}\ \mathtt{c_1d}P[2]\mathtt{c_2\#}\ \mathtt{c_1d}P[3]\mathtt{c_2\#}\ \ldots\mathtt{c_1d}P[|P|-1]\mathtt{c_2\#}$

- We align F_3 with the remaining x AB-patterns interleaved by `#`'s as illustrated below. The cost of this alignment is $2x-1$ substitutions.

  `aaeaaeaaeaaeaaeaaeaaeaae...aaeaa`
  `aabaa#aabaa#aabaa#aabaa#...aabaa`

- We align F_4 with the final $|P|$ P-pattern occurrences according to an optimal alignment $\mathcal{A}_{P/Q}$ of P and Q with respect to cost c. Let `p` and `q` denote placeholders for letters of P and Q, respectively. For each edit operation in $\mathcal{A}_{P/Q}$ (insertion of `q`, deletion of `p`, substitution or match between `p` and `q`), we align in $\mathcal{A}_{U/R}$ the corresponding fragment of F_4 and the P-pattern of R as follows.

Insertion	**Deletion**	**Substitution or Match**
$\mathtt{c_1f\ q\ c_2c_3}$	`- - - - - -`	$\mathtt{-\ c_1f\ q\ c_2c_3}$
$\mathtt{\#\ f\ q\ c_2c_3}$	$\mathtt{\#\ c_1d\ p\ c_2c_3}$	$\mathtt{\#\ c_1d\ p\ c_2c_3}$

  When inserting a letter of Q, rather than paying 5 consecutive gaps opposite to fragment $\mathtt{c_1fqc_2c_3}$ of F_4, we extend the gadget $\oplus$ of R with $\mathtt{\#fqc_2c_3}$, to pay only one (unavoidable) substitution for `#`. Deleting a letter of P, instead, results in 6 gaps in $\mathcal{A}_{U/R}$. Finally, substitutions and matches in $\mathcal{A}_{P/Q}$ result in the same alignment in $\mathcal{A}_{U/R}$, with the cost being, respectively, 3 and 2 according to whether $\mathtt{q}=\mathtt{p}$ or not. Therefore, it turns out that the cost of this last fragment of alignment $\mathcal{A}_{U/R}$ equals $d_{(1,6,3,2)}(P,Q)$.

We next show that it is possible to express $d_{(1,6,3,2)}(P,Q)$ in terms of $d_{(2.5,2.5,1,0)}(P,Q)$, because symmetry will greatly simplify things later on, when we swap P and Q.

Lemma 61. *Let c and c' be two costs. We write $c \sim c'$ if for any alphabet Σ and for all $P, Q \in \Sigma^*$, the set of optimal alignments of P and Q with respect to cost c is equal to the set of optimal alignments of P and Q with respect to cost c'. Then*

1. *$c \sim \alpha c$ for all $\alpha \in \mathbb{R}_{>0}$.*
2. *$c \sim (c_i + \alpha, c_d, c_s + \alpha, c_m + \alpha)$ for all $\alpha \in \mathbb{R}$.*
3. *$c \sim (c_i, c_d + \alpha, c_s + \alpha, c_m + \alpha)$ for all $\alpha \in \mathbb{R}$.*

Proof. Let the number of insertions, deletions, substitutions and matches in some alignment of P and Q be n_i, n_d, n_s and n_m respectively. We know that $n_i+n_s+n_m = |Q|$ and $n_d + n_s + n_m = |P|$. So the transformations 1, 2, and 3 of c given in the lemma statement change the costs of alignments from d to αd, $d+\alpha|Q|$ and $d+\alpha|P|$ respectively. The costs of alignments are all strictly increasing in d, so the optimal alignments are preserved. □

By applying transformation 2 of Lemma 61 with $\alpha = 1.5$ and then transformation 3 of Lemma 61 with $\alpha = -3.5$, we obtain

$$d_{(1,6,3,2)}(P,Q) = d_{(2.5,2.5,1,0)}(P,Q) - 1.5|Q| + 3.5|P|. \tag{D.6}$$

By summing up the costs of the alignment $\mathcal{A}_{U/R}$ detailed above and using Equation D.6, we get

$$d_E(U, X_{\mathrm{ED}}) \leq 4.5(|P| + |Q|) + d_{(2.5,2.5,1,0)}(P,Q), \tag{D.7}$$

which we can bound by $3|P| + 7|Q|$, because $d_{(2.5,2.5,1,0)}(P,Q) \leq 2.5(|Q| - |P|) + |P|$, corresponding to the cost of deleting the $(|Q| - |P|)$ extra letters of Q (recall that $|P| \leq |Q|$) and substituting the remaining $|P|$ letters. In Lemma 62 we prove that alignment $\mathcal{A}_{U/R}$ is indeed optimal and equality holds in Equation D.7.

Lemma 62. *Alignment $\mathcal{A}_{U/R}$ is optimal. Moreover, from any output X_{ED} of* ETFS *on U we can obtain a supersequence P' of P in $\mathcal{O}(|Q|)$ time such that $d_c(P,Q) = |P'| - |P| + d_c(P',Q)$ and there exists an optimal alignment of P' and Q, which does not use any insertions.*

The reader can probably share the intuition that alignment $\mathcal{A}_{U/R}$ is optimal, at least for the part $F_1F_2F_3$ of string U. We prove that indeed no AB-pattern is aligned to any part of F_4 and that no P-pattern is aligned to $F_1F_2F_3$ (see Example 32). The proof of Lemma 62 consists of a case analysis combined with basic counting and bounding arguments.

Example 32. *Let $P =$ KITTEN and $Q =$ SITTING over $\Sigma = \{E, G, I, K, N, S, T\}$. We define a new alphabet $\Sigma' = \Sigma \sqcup \{a, b, c_1, c_2, c_3, d, e, f, g\}$ and a new string $U(P,Q) = F_1F_2F_3F_4$ over Σ' as follows (recall that $x = 2|Q|$, so $2x + 2 = 4Q + 2 = 30$):*

$$\begin{aligned} F_1 &= aabaabaabaabaabaabaabaab\ldots aabaae \\ F_2 &= c_1dKc_2c_3c_1dIc_2c_3c_1dTc_2c_3c_1dTc_2c_3c_1dEc_2c_3c_1dNc_2c_3 \\ F_3 &= aaeaaeaaeaaeaaeaaeaaeaaeaae\ldots aaeaa \\ F_4 &= c_1fSc_2c_3c_1fIc_2c_3c_1fTc_2c_3c_1fTc_2c_3c_1fIc_2c_3c_1fNc_2c_3c_1fGc_2c_3 \end{aligned}$$

We also set $k = 5$ and define the set $\mathcal{I}$ of non-sensitive pattern occurrences over U as follows:

$$\mathcal{I} \;=\; \{0, 3, 6, 9, \ldots, 6x\} \;\cup\; \{6x+6, 6x+11, 6x+16, \ldots, 6x+1+5|P|\}.$$

We thus have the following sequence of occurrences of non-sensitive patterns:

$$\begin{aligned} &aabaa, aabaa, aabaa, \ldots, aabaa && (29 \text{ occurrences}) \\ &c_1dKc_2c_3,\ c_1dIc_2c_3,\ c_1dTc_2c_3c_1dTc_2c_3,\ c_1dEc_2c_3,\ c_1dNc_2c_3 && (6 \text{ occurrences}). \end{aligned}$$

Therefore, the corresponding regular expression R is

$$R = \ominus\, aabaa \oplus \ldots \oplus aabaa \oplus c_1dKc_2c_3 \oplus c_1dIc_2c_3 \oplus c_1dTc_2c_3c_1dTc_2c_3 \oplus c_1dEc_2c_3 \oplus c_1dNc_2c_3 \otimes.$$

We now show the crucial fragment of alignment $\mathcal{A}_{U/R}$: how F_4 is aligned with the P-patterns.

$\texttt{-c}_1\texttt{fSc}_2\texttt{c}_3\texttt{-c}_1\texttt{fIc}_2\texttt{c}_3\texttt{-c}_1\texttt{fTc}_2\texttt{c}_3\texttt{-c}_1\texttt{fTc}_2\texttt{c}_3\texttt{-c}_1\texttt{fIc}_2\texttt{c}_3\texttt{-c}_1\texttt{fNc}_2\texttt{c}_3\texttt{c}_1\texttt{fGc}_2\texttt{c}_3$
$\texttt{\#c}_1\texttt{dKc}_2\texttt{c}_3\texttt{\#c}_1\texttt{dIc}_2\texttt{c}_3\texttt{\#c}_1\texttt{dTc}_2\texttt{c}_3\texttt{\#c}_1\texttt{dTc}_2\texttt{c}_3\texttt{\#c}_1\texttt{dEc}_2\texttt{c}_3\texttt{\#c}_1\texttt{dNc}_2\texttt{c}_3\texttt{\# fGc}_2\texttt{c}_3$

Observe that the cost of the above alignment under unit cost equals to 15: the cost of 4 P-pattern matches (8), plus the cost of 2 P-pattern substitutions (6), plus the cost of 1 gadget insertion (1). It can be readily verified that $d_{(1,6,3,2)}(\texttt{KITTEN}, \texttt{SITTING}) = 15$.

Lemma 62 implies the following result.

Corollary 36. $d_E(U, X_{\mathrm{ED}}) = 4.5(|P| + |Q|) + d_{(2.5,2.5,1,0)}(P, Q)$.

Given that constructing U takes $\mathcal{O}(n)$ time, Corollary 36 tells us that, if we can compute $d_E(U, X_{\mathrm{ED}})$ in strongly subquadratic time, then we can also compute d_c between any two strings in strongly subquadratic time contradicting SETH. In fact, to prove that the *output string* of ETFS cannot be computed in strongly subquadratic time either, we show that $d_{(2.5,2.5,1,0)}$ can be obtained by solving ETFS twice and $\mathcal{O}(n)$ additional operations.

By Lemma 62, from the output X_{ED} of the ETFS algorithm, we can obtain a supersequence P' of P in $\mathcal{O}(n)$ time such that $d_c(P, Q) = d_c(P, P') + d_c(P', Q)$ and no insertions are required to optimally align P' and Q. There also exists a supersequence Q' of Q such that $d_c(P, Q) = d_c(P, P') + d_c(Q', Q) + d_c(P', Q')$ and some optimal alignment of P' and Q' which aligns each $P'[i]$ with $Q'[i]$ through either a match or a substitution. One such Q' is the string obtained by taking the alignment of P' and Q given by ETFS and inserting aligned letters of P' into the gaps of Q. The edit distance of Q and P is

$$d_c(P, P') + d_c(Q, Q') + d(P', Q') = |P'| - |P| + |Q'| - |Q| + \sum_{i=0}^{|P'|-1} I[P'[i] \neq Q'[i]], \quad \text{(D.8)}$$

which can be computed in $\mathcal{O}(n)$ time once we know P' and Q'.

Note that by using ETFS on $U(Q, P')$, we could find a supersequence Q'' of Q such that $d_c(P, Q) = d_c(P, P') + d_c(Q, Q'') + d_c(P', Q'')$ and no deletions are required to optimally align P' and Q''. It is not necessarily the case that we do not need any more insertions, though, as optimal alignments are not unique. We now show that we can still compute an appropriate Q' by changing c.

Let $\mathcal{Q}_c$ be the set of supersequences Q'' of Q with minimal $d_c(Q'', Q) + d_c(P', Q'')$ and no deletions needed in the alignment of P' and Q''. Note that there exists some $Q' \in \mathcal{Q}_c$ such that $|P'| = |Q'|$. Increasing the cost of deletion by ϵ, $d_c(Q'', Q) + d_c(P', Q'')$ increases by at least $\epsilon(|P'| - |Q|)$ with equality if and only if $|Q''| = |P'|$. Since $|Q'| = |P'|$, no deletions implies no insertions. Therefore it suffices to find the insertions, when aligning P' and Q with weights $c' = (2.5, 2.5 + \epsilon, 1, 0)$ for some $\epsilon > 0$. We find these insertions by running the ETFS algorithm on $U(G(Q), G(P'))$ with $k = 5$, where $G(V) = \prod_{i=0}^{|V|-1}(V[i]\mathtt{g})$ for any string $V \in (\Sigma \sqcup \{\mathtt{a}, \mathtt{b}, \mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3, \mathtt{d}, \mathtt{e}, \mathtt{f}\})^*$, and with the set of non-sensitive pattern occurrences

$$\mathcal{I} \;=\; \{0, 3, 6, 9, \ldots, 6x'\} \;\cup\; \{6x' + 6, 6x' + 11, 6x' + 16, \ldots, 6x' + 1 + |Q|\},$$

where $x' = 2|G(P')|$. The solution to this new problem corresponds to an optimal alignment of $G(Q)$ and $G(P')$ with $c = (2.5, 2.5, 1, 0)$, which in its turn corresponds to

an optimal alignment of Q and P' with weights $c' = (5, 5, 1, 0) \sim (2.5, 2.5 + 5, 1, 0)$ by Lemma 61. We first carefully define what properties such a corresponding alignment should satisfy, and then prove that all optimal alignments of $G(Q)$ and $G(P')$ are indeed of this form.

Definition 20. *The alignment of $G(P')$ and $G(Q)$ corresponding to an alignment $\mathcal{A}_{P'/Q}$ of P' and Q is defined as follows:*

- *If $P'[i]$ is aligned with $Q[i]$ in $\mathcal{A}_{P'/Q}$, then $G(P')[2i]$ and $G(P')[2i+1]$ are aligned with $G(Q)[2i]$ and $G(Q)[2i+1]$, respectively.*
- *If $P'[i]$ is deleted in $\mathcal{A}_{P'/Q}$, then $G(P')[2i]$ and $G(P')[2i+1]$ are deleted.*
- *If $Q[i]$ is inserted in $\mathcal{A}_{P'/Q}$, then $G(Q)[2i]$ and $G(Q)[2i+1]$ are inserted.*

Lemma 63. *Let $P', Q \in \Sigma^*$ such that there exits an optimal alignment of P' and Q which does not include any insertions. Each optimal alignment of $G(P')$ and $G(Q)$ with respect to cost $c = (2.5, 2.5, 1, 0)$ corresponds to an optimal alignment of P' and Q with weights $c' = (5, 5, 1, 0)$.*

Proof. Let the number of insertions, deletions, substitutions and matches in some optimal alignment $\mathcal{A}_{P'/Q}$ of P' and Q be w, x, y and z respectively. The cost of $\mathcal{A}_{P'/Q}$ with respect to c' is $5w + 5x + y$. The corresponding alignment of $G(P')$ and $G(Q)$ has $2w$ insertions, $2x$ deletions, y substitutions and $2z + y$ matches, and its cost with respect to c is $(2w) \cdot 2.5 + (2x) \cdot 2.5 + y \cdot 1 + (2z + y) \cdot 0 = 5w + 5x + y$. Therefore $d_c(G(P'), G(Q)) \leq d_{c'}(P', Q)$. It remains to be shown that equality holds.

Consider an optimal alignment $\mathcal{A}_{G(P')/G(Q)}$ of $G(P')$ and $G(Q)$. We will show that we can transform $\mathcal{A}_{G(P')/G(Q)}$ into one corresponding to an alignment of P' and Q without increasing the edit distance. Consider the rightmost $P'[i]$ and $Q[j]$ where the corresponding alignment fails, and call them x and y. There are 13 possibilities for their alignment:

1	2	3	4	5	6	7	8	9	10	11	12	13
`--xg`	`-xg`	`-x-g`	`-xg`	`-xg-`	`x-g`	`xg`	`xg-`	`x--g`	`x-g`	`x-g-`	`xg-`	`xg--`
`ygxg`	`ygg`	`yxgg`	`yxg`	`yxgg`	`ygg`	`yg`	`ygg`	`.ygg`	`.yg`	`.ygg`	`.yg`	`..yg`

Blue letters are original letters of $G(Q)$, red letters are deleted letters from $G(P')$, dots are arbitrary strings and dashes denote gaps. Note that configurations 1, 7 and 13 are already properly aligned. Moreover, the cost can be reduced for configurations 2, 3, 4, 5, 6, 8, 9, 11 and 12 by deleting red letters and shifting blue ones. This only leaves configuration 10. Here there are 3 subcases:

- If `x` is aligned with an `x`, there must be a `g` between `x` and `y`. We can align this `g` with $G(P')[2i]$ and move to configuration 13 without increasing the cost nor changing letters.
- If `x` is aligned with an `x`, we move an adjacent inserted letter to this place and reduce the cost, which is a contradiction.

- Otherwise, `x` is aligned with a different letter. In this case we can realign it with `y` without increasing the cost or changing letters.

Since there is a corresponding alignment for the output string, equality holds. □

Therefore the output string is equal to $G(Q')$ for some $Q' \in \mathcal{Q}_c$. We can infer Q' in $\mathcal{O}(n)$ time and compute $d_c(P, Q)$ using Equation D.8. However, since $d_c(P, Q)$ could not be computed in strongly subquadratic time given SETH, we conclude that ETFS cannot be computed in strongly subquadratic time either, unless SETH is false, thus proving Theorem 35.

D.4 Final Remarks

The following questions remain unanswered. Can ETFS be solved in $\mathcal{O}(n^2)$ time? Can ETFS be solved in strongly subquadratic time when $|\mathcal{S}| = \mathcal{O}(1)$?

Appendix E

Reverse-Safe Text Indexing

Key Points

Problem. Data structures are the workhorse of many data analysis applications, that are often fueled by data collected from individuals and have led to justified privacy concerns. It is thus necessary to guarantee that using data structures does not lead to the reconstruction of the stored individuals' data.

Model. We consider a setting where a large group of users want to query a dataset directly via a data structure which prevents the reconstruction of the data. To this end, we introduce a novel encoding model that enables the construction of *reverse-safe data structures* (RSDSs). These are data structures that prevent the reconstruction of the data they encode (i.e., they cannot be easily reversed). The aim of an RSDS is to make the reconstruction of a dataset sufficiently unlikely, so that an adversary cannot infer the dataset based on the query answers, but at the same time the RSDS stores as many answers to useful queries as possible. In addition, the RSDS should be constructed efficiently and have size close to the size of the original dataset it encodes.

Included Works

This chapter presents the results of the paper **Reverse-Safe Text Indexing**, submitted to the *Journal of Experimental Algorithmics*. This paper is an extension of **Reverse-Safe Data Structures for Text Indexing** [55], which I presented at the *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX 2020)*.

E.1 Introduction

Data structures organize data allowing for their efficient access and modification. They are thus the workhorse of many data analysis applications, such as clustering and outlier detection (e.g., through indexes for k-nearest neighbors join queries [67]), frequent pattern mining (e.g., through FP-trees [190]), document retrieval (e.g., through inverted

indexes [265]), graph pattern matching (e.g., through graph indexes [356]), and range search in databases (e.g., through R-trees [186]).

These applications are often fueled by data collected from individuals, such as location, genomic, or customer data, and have led to justified privacy concerns [329]. To alleviate these concerns and comply with legislation such as HIPAA [125] in the US and GDPR [291] in the EU, it is necessary to guarantee that using data structures does not lead to the reconstruction of the stored individuals' data. This is a fundamentally different privacy goal than that of existing privacy-preserving techniques, such as anonymization [99, 98, 368, 191], sanitization [354, 172, 187, 72, 256, 53], query auditing [282], or access control [61]. Anonymization aims at preventing the disclosure of individuals' identities and/or sensitive information. Sanitization aims at preventing the mining of confidential knowledge. Query auditing aims at preventing answering aggregate queries that leak private information. Access control is the selective restriction of access to some parts of a database. Our privacy goal is also different from that of encryption techniques, such as searchable encryption [62, 247, 303], which aim to prevent unauthorized parties from accessing the data.

We consider a setting where a large group of users want to query a dataset directly via a data structure which prevents the reconstruction of the data. To this end, we introduce a novel encoding model that enables the construction of *reverse-safe data structures* (RSDSs). The ultimate aim of an RSDS is to make the reconstruction of a dataset sufficiently unlikely, so that an adversary cannot infer the dataset based on the query answers, but at the same time the RSDS stores as many answers to useful queries as possible in order to support applications. In addition, the RSDS should be constructed efficiently and have size close to the size of the original dataset it encodes. Our idea is inspired by *encoding data structures* (EDSs) [305]. The ultimate aim of an EDS is to break the information-theoretical lower bound, which is required to store a dataset, by storing only the answers to useful queries (e.g., range queries [149, 177] or nearest largest value queries [194]).

Given a data structure D, we denote by $\mathcal{A}_D$ its set of *consistent* datasets: all datasets with the same set of answers as the answers stored by D. Let us denote $\alpha_D = |\mathcal{A}_D|$. Given an integer threshold $z > 1$, which we call the *privacy threshold*, we say that D is z-RSDS if and only if $\alpha_D \geq z$. A large z implies strong data privacy because an adversary cannot distinguish between the $\alpha_D \geq z$ consistent datasets, which implies that it is less likely that the adversary infers the dataset used to construct D in the first place. Still, it could be the case that D stores answers to many useful queries.

The notion of z-RSDS is related to the privacy notion of *z-anonymity* [316]. This notion was introduced in the context of a relational database, where each record corresponds to a different individual. The notion of z-anonymity dictates that at least $z > 1$ records of the database must have the same values over a set of attributes that may lead to the disclosure the identity of individuals in the database. The privacy goal is to prevent an adversary from distinguishing an individual among at least z individuals in the database.

In this work, we consider string data (sometimes called text, word, or document depending on the context). A string is a sequence of letters from an alphabet. A string may represent various types of confidential information about individuals, including their movement history [346], diagnosed diseases [339], purchased products [342], or

DNA sequence [263]. Our goal is to construct a z-RSDS for string data which allows for decision and counting pattern matching queries to be accurately and efficiently answered. Decision queries are fundamental for intrusion detection [250], activity monitoring [354], as well as for cataloguing human genetic variation [48], while counting queries are fundamental for pattern mining that is central in application domains ranging from bioinformatics [321] to marketing [256] and to public health [41].

Pattern matching queries in strings are answered efficiently by means of indexing data structures. These structures enable fast access to the substrings of a string, which is important in many data analysis applications [184]. The main idea behind indexing a string S for efficient substring querying is that every substring of S is a prefix of some suffix of S. Indexing data structures thus arrange the suffixes of S lexicographically in an ordered tree data structure. One popular such data structure is the *suffix tree* [357]. The suffix tree of S is the compacted trie of all the suffixes of S. The term compacted refers to the fact that it reduces the number of nodes by replacing each maximal branchless path segment with a single edge, and it uses intervals over S to store the labels of these edges. This ensures that the suffix tree has size linear in $|S|$: it has no more than $2|S|$ nodes. Importantly, the suffix tree answers several types of pattern matching queries over S in optimal time; see [184] for a nice exposition.

However, the suffix tree of S, which provides (random) access to *all* substrings of S, is *not* a z-RSDS, because it uniquely represents S. The *privacy-utility trade-off* we consider here is thus to provide access only to the substrings of S whose length is at most d, for some $d \in [1, |S|)$. In particular, we want our z-RSDS to support the following types of on-line queries.

Decision Query: check if a string P of length $m \leq d$ is a substring of S.

Counting Query: count the occurrences of a string P of length $m \leq d$ in S.

Given a string S and a privacy threshold z, the computational challenge is to compute the *maximal* d for which a z-RSDS for indexing S can be constructed. The maximality of d offers *data utility*, since any query for a substring of S of length d or less has the same answer, irrespectively of whether it is posed on S or on the z-RSDS. The fact that the data structure is z-reverse-safe offers *data privacy*, since the probability that an adversary infers S, based solely on knowledge of the z-RSDS, is no more than $1/z$.

We are now in a position to formally define the main computational problem considered in this chapter[1].

Problem 1. *Given a string S of length n and a privacy threshold $1 < z \leq n^c$, for some constant $c \geq 1$, construct a z-RSDS that answers decision and counting pattern matching queries for any pattern of length $m \leq d$, such that d is maximal, or output* FAIL *if no such d exists.*

In Problem 1, d is maximal and *uniform for all queries*. Another related problem definition would be to maximize the total number of supported queries, not necessarily of uniform maximal length.

[1] The problem of inferring a string from a text indexing data structure (see [223] and references therein) is conceptually related but fundamentally different to the problem investigated here.

Our Contributions. We consider the word-RAM model of computations with w-bit machine words, where $w = \Omega(\log n)$, for stating our results. The main theoretical result of this chapter is the following, where ω denotes the matrix multiplication exponent[2].

Theorem 37. *Given a string S of length n, there exists an $O(n^\omega \log d)$-time algorithm to construct an $O(n)$-sized z-RSDS over S for a maximal d that answers decision and counting pattern matching queries, for any pattern of length $m \leq d$, in the optimal $O(m)$ time per query. The algorithm outputs* FAIL *if no such d exists.*

The main ingredients of our construction algorithm include (truncated) suffix trees [280, 357], a combinatorial theorem on de Bruijn graphs [203, 221], and fast matrix multiplication [362, 242]. To the best of our knowledge we are the first to combine these ingredients. We show that, despite the n^ω factor, our engineered implementation can construct z-RSDSs over million-letter texts in only a few minutes. To achieve this practical performance, we rely on further theoretical insight. We also show that plugging our method in data analysis applications gives insignificant or no data utility loss. Furthermore, we show how our technique can be extended at no extra cost to construct a z-RSDS that supports applications under two realistic adversary models: one with positive adversarial knowledge (an adversary knows a pattern that occurs in S); and the other with negative adversarial knowledge (an adversary knows that a pattern does not occur in S). We also show how the z-RSDS for both adversary models can be generalized to an arbitrary number of patterns. Both positive and negative adversarial knowledge have been studied in the context of z-anonymity [249, 35]. The works in [249] and [35] have a different privacy goal than ours (preventing the disclosure of identities of individuals and/or their sensitive information vs. preventing dataset reconstruction) and do not consider a string but a relational and a set-valued dataset, respectively. Finally, we show a different z-RSDS for decision pattern matching queries, whose size can be sublinear in n.

Organization of the Chapter. The basic definitions and notation are introduced in Section E.2. In Section E.3, we propose a z-RSDS for text indexing. In Section E.4, we present our construction algorithm. We then describe a series of practical improvements in Section E.5. In Section E.6, we present our implementation and extensive experimental results. In Section E.7, we discuss how to construct an adapted version of our z-RSDS under two different adversary models. In Section E.8, we show a different z-RSDS for answering decision pattern matching queries. We conclude this chapter in Section E.9 with some final remarks and open problems.

E.2 Preliminaries

We fix a string $S = S[0] \cdots S[n-1]$ over $\Sigma = \{1, \ldots, n^{O(1)}\}$. We will assume that S contains at least two different letters, otherwise the problem considered in this chapter is trivial. Given a positive integer k, we denote by $(S)_{k,i}$ the length-k substring of S starting at position i, i.e., $(S)_{k,i} = S[i\,..\,i+k-1]$, for all $0 \leq i < n-k+1$. A string P

[2] At the time of writing this thesis, $\omega < 2.373$ [362, 242].

has an *occurrence* in S or, more simply, it *occurs* in S if $P = (S)_{|P|,i}$, for some i. An occurrence of P is thus characterized by its starting position i in S.

We will denote by $G_{S,k} = (V_{S,k}, E_{S,k})$ the *weighted de Bruijn graph* of order k over S: a formal definition can be found in Section A.2 (inspect Fig. E.3 for an example).

E.3 A z-RSDS for Text Indexing

Let S be a string of length n. For a positive integer d, we define a *d-substring* of S as a substring of length d of S, or a suffix of S whose length is less than d.

The *d-truncated suffix tree* of a string S, denoted by $\mathcal{T}_d(S)$, is a path-compacted trie representing every d-substring of S [280]. We make use of a terminating letter $\# \notin \Sigma$ for technical purposes. Formally, $\mathcal{T}_d(S)$ is a rooted tree satisfying the following conditions (see Fig. E.1 for an example):

1. Each edge is labeled with a non-empty substring of string $S\#$ encoded as an $[i, j]$ interval over $[0, n]$.
2. Each internal node v, except possibly the root, has at least two children. The labels of edges from v to its children start with distinct letters.
3. Let $\mathcal{L}(v)$ denote the string obtained by concatenating labels on the path from the root to node v. For every d-substring U, there is exactly one leaf w such that $U = \mathcal{L}(w)$ (if $|U| = d$) or $U\# = \mathcal{L}(w)$ (if $|U| < d$). For each leaf w, there is at least one d-substring U such that $\mathcal{L}(w) = U$ or $\mathcal{L}(w) = U\#$.
4. Each node v other than the root has a counter that stores the number of substrings of string $S\#$ that are equal to $\mathcal{L}(v)$.

Therefore, the number of leaves is at most n and the total number of nodes is less than $2n$. Recall that the label of the edge between a node u and its child v, denoted by $\mathit{label}(u, v)$, is represented implicitly by an interval over $[0, n]$. Thus the space occupied by $\mathcal{T}_d(S)$ is $O(n)$. The children of internal nodes are indexed by the alphabet letters using perfect hashing to ensure $O(1)$-time access [154]. Importantly, $\mathcal{T}_d(S)$ supports the following on-line pattern matching operations:

Decision Query: Check if a string P of length $m \leq d$ is a substring of S in $O(m)$ time.

Counting Query: Count the occurrences of a string P of length $m \leq d$ in S in $O(m)$ time.

Theorem 38 ([280, 95])**.** *Given a string S of length n and $0 < d \leq n$, $\mathcal{T}_d(S)$ has size $O(n)$ and it can be constructed in $O(n)$ time. $\mathcal{T}_d(S)$ answers decision and counting pattern matching queries, for any pattern of length $m \leq d$, in the optimal $O(m)$ time per query.*

The following off-line operations are also supported:

Frequent Substrings: Find all most frequent substrings, for all lengths $1, 2, \ldots, d$, in $O(n)$ time.

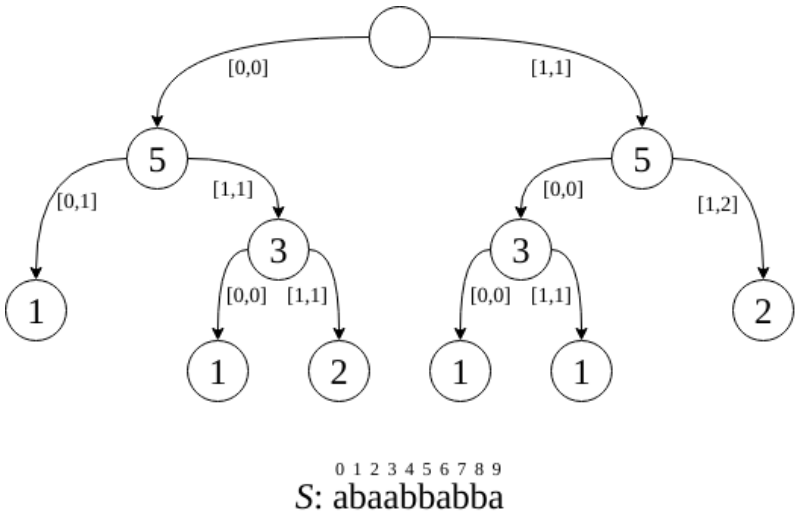


Figure E.1: $\mathcal{T}_d(S)$ for $S = \texttt{abaabbabba}$ and $d = 3$. We omit edges whose labels start with letter # for clarity.

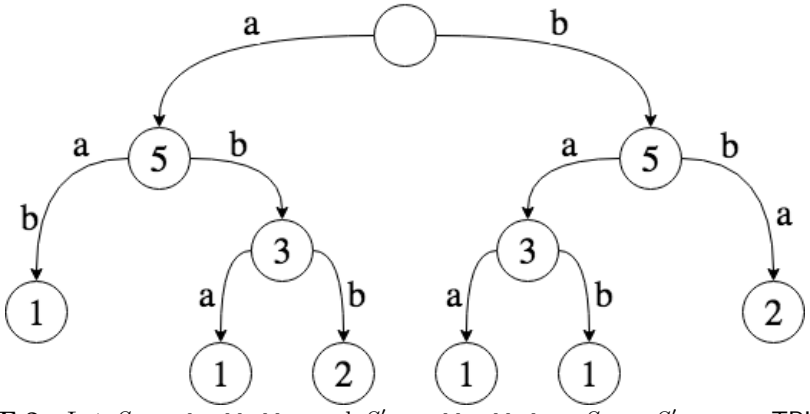


Figure E.2: Let $S = \texttt{abaabbabba}$ and $S' = \texttt{abbaabbaba}$. $S \sim_3 S' \iff \mathsf{TRIE}_3(S) = \mathsf{TRIE}_3(S')$.

Repeated Substrings: Find all longest repeated substrings of length at most d in $O(n)$ time.

Unique Substrings: Find all shortest unique substrings of length at most d in $O(n)$ time.

We next consider a different representation of $\mathcal{T}_d(S)$ towards defining the notion of z-reverse-safe data structure. If $\mathit{label}(u, v)$ is represented explicitly by a string we denote the resulting data structure by $\mathsf{TRIE}_d(S)$. In this case, string S is not part of the data structure, and thus $\mathsf{TRIE}_d(S)$ *does not*, generally, define S uniquely.

Definition 21 (d-Equivalent Strings)**.** *Given the set of all possible strings of length n over an alphabet Σ and an integer d, string S is* d-equivalent *to string S' if and only*

if $\mathsf{TRIE}_d(S) = \mathsf{TRIE}_d(S')$*. In this case, we write* $S \sim_d S'$ *and say that* S' *is* consistent *with* $\mathcal{T}_d(S)$*.*

See Fig. E.2 for an example. We can now formally define a z-reverse-safe data structure for text indexing.

Definition 22 (z-RSDS for Text Indexing)**.** *Given an integer* $z > 1$*,* $\mathcal{T}_d(\cdot)$ *is called* z-reverse-safe *if and only if there exist at least* z *distinct strings that are consistent with* $\mathcal{T}_d(\cdot)$*.*

In what follows, we denote the set of strings that are consistent with $\mathcal{T}_d(S)$ by $\mathcal{A}_d(S)$, and $|\mathcal{A}_d(S)|$ by $\alpha_d(S)$, for $d \in [1, n]$. We omit (S) when this is clear from the context, and we also set $\alpha_0(S) = \infty$ for completeness.

E.4 Constructing z-RSDS

Clearly, $\mathcal{T}_n(S)$, the (non-truncated) suffix tree of S, has $\alpha_n = 1$ (i.e., it uniquely represents S), so it can never be a solution to Problem 1 since $z > 1$, by definition.

The following lemma is important for efficiency.

Lemma 64. *The sequence* $\alpha_0, \alpha_1, \ldots, \alpha_n$ *is monotonically non-increasing.*

Proof. Let $\mathcal{A}_d$ be the set of strings consistent with $\mathcal{T}_d$, $d \in [1, n]$, and $\alpha_d = |\mathcal{A}_d|$. Further let S be any element of $\mathcal{A}_d$. By construction, if U is a d-substring of S, then $U = \mathcal{L}(w)$ or $U\# = \mathcal{L}(w)$, for some leaf w of $\mathcal{T}_d$. Every $(d-1)$-substring $S[i\,..\,i+d-2]$ of S is a prefix of the d-substring $S[i\,..\,i+d-1]$ of S. Thus string S is consistent with $\mathcal{T}_{d-1}$, the path-compacted trie that represents every such $(d-1)$-substring, and thus $S \in \mathcal{A}_{d-1}$. This implies the following relation: $\mathcal{A}_n \subseteq \mathcal{A}_{n-1} \subseteq \cdots \subseteq \mathcal{A}_1$. The statement follows directly from this relation and the fact that $\alpha_0 = \infty$. □

By Lemma 64, for increasing d, $\mathcal{T}_d(S)$ generally decreases α_d and increases utility. We thus need an algorithm to compute the maximum possible d that results in a z-RSDS. We next provide an algorithm, called z-RC (for z-RSDS Construction), to find this d.

As can be seen in the pseudocode, z-RC performs binary search on n (the length of S), computing α_d until d results in a z-RSDS and d is maximal. An alternative is to perform exponential search instead of binary search. We refer to this variation of z-RC as z-RCE (for z-RSDS Construction Exponential). At this point, the z-RSDS $\mathcal{T}_d(S')$ is output, where S' is an element of $\mathcal{A}_d$ chosen at random, and the algorithm terminates. If $\ell > 0$ and $\alpha_{\ell-1} = z$, then $\alpha_{\ell-1}$ is the rightmost element that equals z. Even if such an element is not found, $n - \ell$ is the number of elements that are smaller than z.

The computational challenge is thus to implement the check of Line 5 efficiently and to find a consistent S' when this is possible (Line 11). To this end, we start with the following simple yet crucial observation.

Observation 39. *Given two strings* X *and* Y*,* X *is* d*-equivalent to* Y *if and only if* X *and* Y *have the same multisets of substrings of length* i*, for every* $i \in [1, d]$*.*

In the terminology of combinatorics on words, d-equivalence is known as d-*abelian equivalence* [222]. We report a lemma from [222], which gives several equivalent conditions that characterize d-equivalence.

Algorithm 12: z-RC

Input: string S of length n and integer $z > 1$
Output: d and $\mathcal{T}_d(S')$, for some $S' \in \mathcal{A}_d$, or FAIL

1 $\ell \leftarrow 0$; $r \leftarrow n$;
2 **if** $\ell \geq r$ **then**
3 go to Line 10;
4 $d \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$;
5 **if** $\alpha_d(S) \geq z$ **then**
6 $\ell \leftarrow d + 1$;
7 **else**
8 $r \leftarrow d$;
9 go to Line 2;
10 **if** $\ell > 0$ **then**
11 **output** $d \leftarrow \ell - 1$ and $\mathcal{T}_d(S')$, for some $S' \in \mathcal{A}_d$
12 **else**
13 **output** FAIL

Lemma 65 ([222]). *Let X and Y be two strings of length at least d that have the same multiset of substrings of length d. The following are equivalent:*

1. *X and Y have the same multiset of substrings of length i for every $1 \leq i \leq d$;*
2. *X and Y have the same prefix of length $d-1$ and the same suffix of length $d-1$;*
3. *X and Y have the same prefix of length $d-1$;*
4. *X and Y have the same suffix of length $d-1$.*

Lemma 65 tells us that we should rely on the construction of weighted de Bruijn graphs over string S in order to compute $\alpha_d(S)$. The weighted de Bruijn graph of order d over string S is denoted by $G_{S,d} = (V_{S,d}, E_{S,d})$. Recall that its set of vertices $V_{S,d}$ is the set of distinct substrings of S of length $d-1$ (we implicitly identify a vertex by the string it represents) and there is an edge $(u, v) \in E_{S,d}$ with multiplicity m if and only if $u[0] \cdot v = u \cdot v[d-2]$ and this string occurs in S exactly m times. We borrow the terminology used in [229]. Let $d^-(u)$ and $d^+(u)$ be, respectively, the in- and out-degree of vertex u of $G_{S,d}$. Let s and t be the vertices of $G_{S,d}$ corresponding, respectively, to the prefix and to the suffix of length $d-1$ of S. Since any weighted de Bruijn graph is either Eulerian (if $s = t$) or semi-Eulerian (if $s \neq t$), we have that $d^+(u) = d^-(u)$ for all u with the possible exception of the two nodes s and t for which $d^-(s) = d^+(s) - 1$ and $d^+(t) = d^-(t) - 1$, if $s \neq t$. Clearly, S corresponds to an Eulerian path in $G_{S,d}$ that starts at s and ends at $t \neq s$ (if $s = t$, then it corresponds to an Eulerian cycle starting from s). The graph $G_{S,d}$ may contain other Eulerian paths (resp. cycles). Notice, however, that if two distinct Eulerian paths (resp. cycles) traverse the vertices of $G_{S,d}$ in the same order, but the edges in different order, then they give rise to the same string. We call these Eulerian paths (resp. cycles) *equivalent*. We summarize these observations into the following statement, which is crucial for the correctness of the z-RC algorithm.

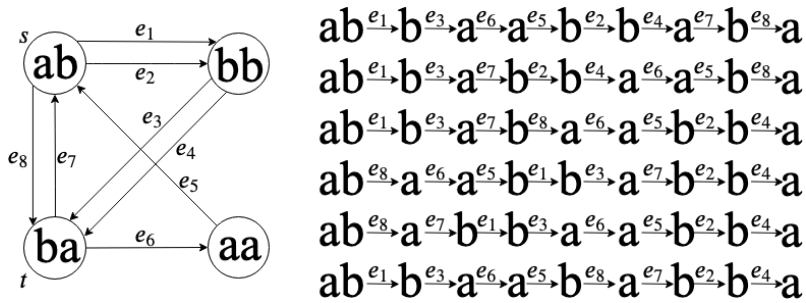


Figure E.3: $G_{S,d}$ with $S = \texttt{abaabbabba}$ and $d = 3$ (on the left); and the set of d-equivalent strings (on the right).

Observation 40. *(a) If $S \sim_d S'$, then S' corresponds to an Eulerian path in $G_{S,d}$ that starts from vertex s and ends at vertex $t \neq s$ (if $s = t$, then it corresponds to an Eulerian cycle starting from s). (b) The number of distinct strings that are d-equivalent to S is the number of* non-equivalent *Eulerian paths (resp. cycles) in $G_{S,d}$.*

The number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d}$ can be computed via the following theorem, which is attributed to Hutchinson [203].

Theorem 41 ([203], cf. [229, 221])**.** *Let $A = (a_{uv})$ be the adjacency matrix of the weighted de Bruijn graph $G_{S,d} = (V_{S,d}, E_{S,d})$, with both $a_{uv} > 1$ (multi-edges) and $a_{uu} > 0$ (self-loops) allowed. Let $r_u = d^+(u) + 1$ if $u = t$ or $r_u = d^+(u)$ otherwise. The number of non-equivalent Eulerian paths starting at s and ending at t (resp. the number of non-equivalent Eulerian cycles starting at s, when $t = s$) is given by*

$$(\det L_{S,d}) \cdot \left(\prod_{u \in V_{S,d}} (r_u - 1)! \right) \cdot \left(\prod_{(u,v) \in E_{S,d}} a_{uv}! \right)^{-1}, \tag{E.1}$$

where $L_{S,d} = (l_{uv})$ is the $|V_{S,d}| \times |V_{S,d}|$ matrix with $l_{uu} = r_u - a_{uu}$ and $l_{uv} = -a_{uv}$.

Let us denote by $|S|_x$ the number of occurrences of a string x in S. Since, by definition, $r_u = |S|_u$ and $a_{uv} = |S|_{u \cdot v[k-2]} = |S|_{u[0] \cdot v}$, Eq. E.1 is equivalent to

$$(\det L_{S,d}) \cdot \left(\prod_{u \in V_{S,d}} (|S|_u - 1)! \right) \cdot \left(\prod_{a \in \Sigma} |S|_{ua}! \right)^{-1}. \tag{E.2}$$

Eq. E.2, together with a combinatorial study of the strings that belong to the same d-equivalence class, can be found in [221]. An example is provided with Fig. E.3.

It is, however, not immediate that Eq. E.1 (or the equivalent Eq. E.2), involved in the check of Line 5 in algorithm z-RC, can be computed efficiently. We show this next, starting with a known fact on de Bruijn graphs.

Fact 42 ([89])**.** *Given a string S of length n and $d < n$, its weighted de Bruijn graph $G_{S,d}$ can be constructed in $O(n)$ time.*

Lemma 66. $\det A$ *of an $n \times n$ non-singular matrix A can be computed in $O(n^\omega)$ time.*

Proof. The decomposition of a non-singular matrix $A = LU$, where L and U is a lower and upper triangular matrix, respectively, is known as LU decomposition and can be computed in the same time as matrix multiplication [84]. Given this decomposition, the determinant can be computed as $\det A = \det L \cdot \det U = \prod_{i=1}^{n} l_{ii} \cdot \prod_{i=1}^{n} u_{ii}$. This is because the determinant of any triangular matrix (such as L and U) is the product of its diagonal entries. □

Lemma 67. *Given* $\det L_{S,d}$*, the check of Line 5 in algorithm* z-RC *can be performed in* $O(n \log n)$ *time.*

Proof. We unfold all factorials involved in the two products of Eq. E.1. Let us first consider the leftmost product. Observe that the total number of multiplications involved is no more than n because the sum of out-degrees over all nodes of $G_{S,d}$ is no more than n. Moreover, observe that each factor of the product is represented by $\log n$ bits because its value is no more than n. We assume a word-RAM algorithm that takes $O(n_1 + n_2)$ arithmetic operations to multiply an n_1-bit integer by an n_2-bit integer [232] resulting in an $(n_1 + n_2)$-bit integer. Using a $\log n$-depth divide and conquer, we can multiply these n integers in time $O(\frac{n}{2^1} 2^0 \log n + \frac{n}{2^2} 2^1 \log n + \ldots + \frac{n}{2^{\log n}} 2^{\log n - 1} \log n) = O(n \log n)$. Using an analogous argument, the rightmost product can be computed in $O(n \log n)$ time, because $G_{S,d}$ has no more than n edges, which implies that the product has at most n factors.

The leftmost product results in an $(n \log n)$-bit integer (we multiply n $\log n$-bit integers). By Hadamard's inequality [175] an upper bound on the value of $\det L_{S,d}$ is $B^n \cdot n^{n/2}$, where B is an upper bound on the values in $L_{S,d}$. Since here $B \leq n$, an upper bound on $\det L_{S,d}$ is $n^n \cdot n^{n/2} = n^{3n/2}$, which can be expressed using $\log(n^{3n/2}) = 1.5n \log n$ bits. Multiplying $\det L_{S,d}$ by the leftmost product is thus done in $O(n \log n)$ time. The rightmost product also results in an $(n \log n)$-bit integer, which we multiply by z. Since $z \leq n^c$ is a $c \log n$-bit integer, this is done in $O(n \log n)$ time. Thus, Line 5 is checked in $O(n \log n)$ time if $\det L_{S,d}$ is known. □

We arrive at the main theoretical result.

Theorem 37. *Given a string* S *of length* n*, there exists an* $O(n^\omega \log d)$*-time algorithm to construct an* $O(n)$*-sized* z*-RSDS over* S *for a maximal* d *that answers decision and counting pattern matching queries, for any pattern of length* $m \leq d$*, in the optimal* $O(m)$ *time per query. The algorithm outputs* FAIL *if no such* d *exists.*

Proof. The correctness of z-RC algorithm follows by Lemma 64 and Observation 40. The correctness of querying follows by the definition of d-equivalent strings.

The construction time follows by Fact 42, Lemmas 66-67, Theorem 38, and the binary search cost over $[0, n]$. Specifically, the check of Line 5 is implemented in $O(n^\omega)$ time by Fact 42 and Lemmas 66-67. If we find a valid d, we choose an Eulerian path (resp. cycle) of $G_{S,d}$ to construct a string S' and then construct $\mathcal{T}_d(S')$ using Theorem 38 in $O(n)$ time (Line 11). The z-RSDS size and the time per query follow by Theorem 38. If no such d exists the algorithm outputs FAIL.

If we apply exponential search (instead of binary search) as in z-RCE, we get an $O(n^\omega \log d)$-time construction. □

Colbourn et al. [106] gave an algorithm allowing for sampling of a random arborescence rooted at a given node to be carried out in the same time as counting all such arborescences, which forms the basis of counting Eulerian paths and cycles in directed multigraphs. Hence, by plugging the algorithm of Colbourn et al. in our construction algorithm (Line 11), we can also choose a string $S' \sim_d S$ *randomly* in the same time complexity.

E.5 Engineering the z-RC Algorithm

In what follows we describe a series of practical improvements, which are based on theoretical insight.

E.5.1 Improvement I: Reducing the BS Interval

Lemma 68. *Let S be a string and $r(S)$ be the length of a longest substring of S occurring at least twice in S. $\mathcal{T}_d(S)$ cannot be a z-RSDS over S if $d \geq r(S) + 2$.*

Proof. Let I be the set of substrings of length $r(S) + 2$ of string S. Having set I is a sufficient condition for the unique reconstruction of S from I [145, 88]. This implies that, if $d \geq r(S) + 2$, $\mathcal{T}_d(S)$ defines S in a unique way (i.e., $\alpha_d = 1$), and thus $\mathcal{T}_d(S)$ cannot be a z-RSDS (since by definition $z > 1$). □

Note that the upper bound of $r(S) + 1$ can be computed in $O(n)$ time using the suffix tree of S [139], which is much faster than computing the bounds found by an exponential search. This is because exponential search takes $O(n^\omega)$ time for each of its iterations. As a consequence of Lemma 68, we can reduce the binary search interval from $[0, n]$ to $[0, r(S) + 1]$ in $O(n)$ time. Furthermore, it is known that $r(S)$ tends to $\log_{|\Sigma|} n$ as n tends to infinity under a Bernoulli i.i.d. model (cf. [145]).

E.5.2 Improvement II: Checking Prefixes of S

Lemma 69. *Let S be a string and P be a prefix of S. Further, let $\mathcal{A}_d(P)$ (respectively, $\mathcal{A}_d(S)$) be the set of strings that are consistent with $\mathcal{T}_d(P)$ (respectively, with $\mathcal{T}_d(S)$). It holds that $\alpha_d(P) \leq \alpha_d(S)$.*

Proof. We show the lemma by showing that for any string X and any letter a, $\alpha_d(X) \leq \alpha_d(Xa)$. This implies that $\alpha_d(P) \leq \alpha_d(S)$. Indeed, by Lemma 65, it follows that if X' is d-equivalent to X, then $X'a$ is d-equivalent to Xa. □

Lemma 69 lets us implement the check in Line 5 of the z-RC algorithm by operating on the prefixes of S. The length of a longest substring of every prefix P of S occurring at least twice in P can be computed by means of the longest previous factor (LPF) array [115]. The LPF array gives, for each position i in S, the length of a longest substring occurring both at i and to the left of i in S. We can thus construct an array R, where $R[i]$ stores the length of a longest substring occurring at least twice in the prefix $P = S[0\,..\,i]$ of S, by traversing the LPF array. Then, we only need to perform the check $\alpha_d(P) \geq z$ when $d < R[i] + 2$. This is because of applying Improvement I

on P. The LPF array, and thus array R, can be computed both in $O(n)$ time [115]. Note that $R[i] \leq R[i+1]$. Thus, having R, we can find (whether there exists) a prefix $P = S[0\,..\,i]$ satisfying $d < R[i] + 2$, for all d, in $O(n)$ time in total.

E.5.3 Improvement III: Sparse LU Decomposition

Let $G_{S,d} = (V_{S,d}, E_{S,d})$ be the weighted de Bruijn graph for which we must compute the determinant $\det L_{S,d}$. $L_{S,d}$ is a $|V_{S,d}| \times |V_{S,d}|$ non-singular matrix, where $|V_{S,d}|$ is the number of distinct substrings of length $d-1$ occurring in S. Hence we have that $|V_{S,d}| \leq \min(|\Sigma|^{d-1}, n-d+1)$. If $|V_{S,d}| = O(n^{1/\omega})$, then $\det L_{S,d}$ is computed in $O(n)$ time by Lemma 66. If $|V_{S,d}| = \Theta(n)$, then $L_{S,d}$ is sparse: it has no more than $|V_{S,d}|+n-d+1$ non-zero elements, because in the worst case there is a non-zero element for each edge and there are $n-d+1$ edges with multiplicity 1. Thus, in any case, $L_{S,d}$ cannot contain more than $2n-d+1$ non-zero elements. We can therefore employ highly-optimized algorithms for sparse LU decomposition (e.g., [171, 124]) to compute $\det L_{S,d}$ efficiently. Let $\text{flops}(XY)$ be the number of multiplications of non-zero elements performed while computing the product XY by conventional matrix multiplication. The algorithm of [171], for instance, takes $O(\text{flops}(LU)+m)$ time to compute the LU decomposition of a matrix with m non-zero elements. Thus, in our case, computing $\det L_{S,d}$ takes $O(\text{flops}(LU)+n)$ time.

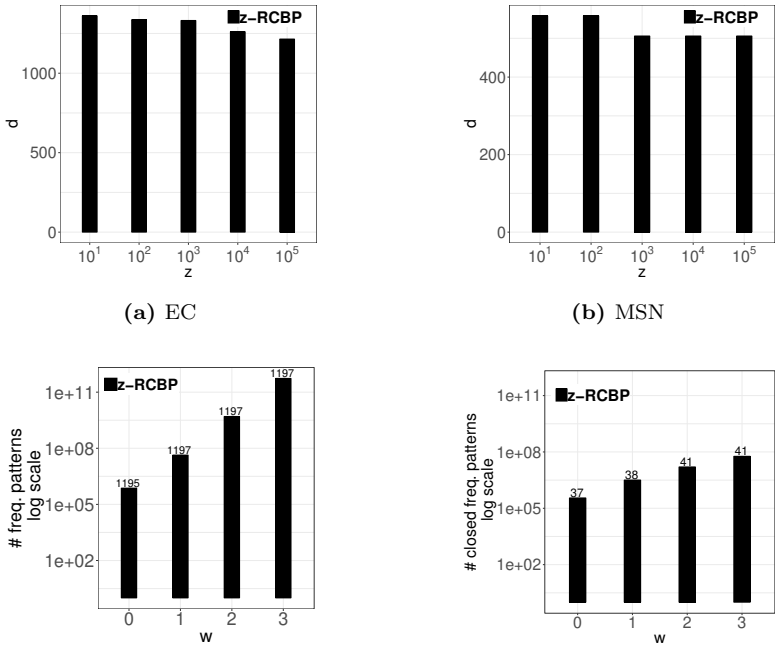
E.6 Implementations and Experiments

E.6.1 Implementations

We have implemented the following algorithms in `C++`: (I) z-RC with Improvement III; (II) z-RCB (for Binary search interval reduction), which implements Improvements I and III; (III) z-RCE with Improvement III; and (VI) z-RCBP (for Binary search interval reduction and Prefix checking), which implements Improvements I, II, and III. Our implementation is available at: `https://www.dropbox.com/s/rt5z50ro2o37h2s/rsds_code.zip?dl=0` [3]. We have omitted the results of the versions of the algorithms without Improvement III, because they were too slow to be practical.

For Improvement II, we have combined the idea described in Section E.5.2 with exponential search: we start from an initial prefix P_0 of S that has length $|P_0| = \kappa$ and use it to perform the check in Line 5 of our algorithm in Section E.4. Due to Lemma 69, we know that $\alpha_d(P_0) \geq z$ implies $\alpha_d(S) \geq z$; we thus check if $\alpha_d(P_0) \geq z$, because this is clearly more efficient than checking $\alpha_d(S) \geq z$. If $\alpha_d(P_0) < z$, $\alpha_d(S) \geq z$ may or may not hold. In this case, we consider a longer prefix of S that has length $|P_1| = 2^1 \cdot \kappa$ and proceed similarly. Clearly, significant computational savings can be brought when the last considered prefix P_i has small length $|P_i| = 2^i \cdot \kappa$, while in the worst case $P_i = S$, and the total cost of our algorithm with Improvement II is twice the cost of the algorithm without it due to doubling. We also apply Improvement I on these prefixes: if $d \geq R[i]+2$ for prefix P_i, we do not check $\alpha_d(P_i) \geq z$, because Lemma 68 already ensures that $\alpha_d(P_i) = 1 < z$.

[3] Our implementation will be made publicly available upon acceptance.



(a) EC **(b)** MSN

(c) EC **(d)** EC

Figure E.4: Length d for different z values for (a) EC and (b) MSN. Number of (c) frequent patterns and (d) closed frequent patterns with up to $w \in [0, 3]$ wildcards mined from EC using $\mathsf{minSup} = 1.8 \cdot 10^{-6}$. The length of the longest mined pattern is on the top of each bar.

For Improvement III, we used the Sparse LU decomposition function of the open-source Eigen library (v. 3.3.7) [330], which is based on the algorithm of [124], to compute $\det L_{S,d}$.

E.6.2 Experimental Setup and Datasets

We have evaluated z-RC, z-RCB, z-RCE and z-RCBP in terms of data utility and efficiency. We do not compare our methods to existing approaches, because they are not alternatives to our work as mentioned in Section E.1.

We used the following publicly available datasets: MSNBC (MSN), which contains page categories visited by users on `msnbc.com` over a 24-hour period; the complete genome of *Escherichia coli* (EC); and a dataset containing 27 Primate mitochondrial genomes (PR). MSN was used in [172, 187, 256], EC was used in [53], and PR was used in [343]. We also generated a uniformly random string of length 50M over an alphabet of size 10, and used its prefixes of length $1\text{M}, \ldots, 50\text{M}$ as synthetic datasets, referred to as $\text{SYN}_{1\text{M}}, \ldots, \text{SYN}_{50\text{M}}$, respectively. Each dataset contains a single string, except for PR which contains 27 strings (one for each mitochondrial genome). In PR, we applied our methods to each string independently. Table E.1 summarizes the characteristics of the datasets.

To evaluate data utility, we report the length d found by our methods for different

Dataset	Data domain	Total length n	Alphabet size $\lvert\Sigma\rvert$
MSN	Web	4,698,764	17
EC	Genomic	4,641,652	4
PR	Genomic	446,246 (27 strings)	4
SYN	Synthetic	50,000,000	10

Table E.1: Characteristics of datasets used.

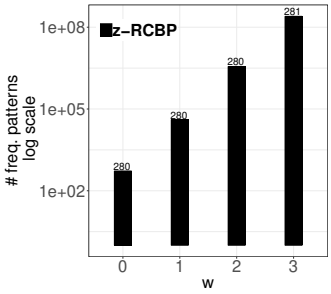
values of z, and also investigate the accuracy of performing two classes of data analysis applications: *pattern mining* [375] and *phylogenetic tree reconstruction* [343]. Unlike decision and counting pattern matching queries of length at most d, which are answered exactly using the z-RSDS constructed by our methods, these applications are not guaranteed to be performed accurately on the output encoding. Yet, we show that plugging in our approach gives insignificant or no data utility loss in these applications.

We now discuss each of these applications.

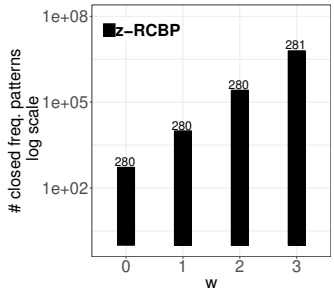
(Closed) Frequent Pattern Mining. Frequent patterns and closed frequent patterns in string datasets model knowledge that aids decision making [33, 321] and can be used for data classification and clustering [375]. Given a string S and a user-specified threshold minSup, a pattern is *frequent* if its relative frequency in S, also referred to as *support*, is at least minSup. A frequent pattern of S is *closed* if none of its superstrings has the same relative frequency in S. Closed frequent patterns are typically fewer than the frequent ones and they are mined much more efficiently. Their benefit is that they uniquely determine the set of frequent patterns and their exact frequency. Our methods allow mining the frequent and closed frequent patterns of length at most d and only those. Thus, our methods preserve data utility well when the d computed is sufficiently large for low minSup values. In our experiments, we used the algorithm of [33] to mine a more general class of frequent and closed frequent patterns having up to $w \in [0, 3]$ occurrences of a wildcard letter $\diamond$. A pattern with wildcards occurs in a string S if it is a subtring of S after replacing the wildcard letters with alphabet letters (e.g., pattern $\mathtt{a}\diamond\diamond\mathtt{e}$ occurs in $S = \mathtt{babdeb}$). Mining patterns with wildcards poses a further challenge to our approach, since (closed) frequent patterns with wildcards are a superset of the (closed) frequent patterns and are typically longer.

Phylogenetic Tree Reconstruction. A phylogenetic tree illustrates the evolutionary relationships among a set of species. To reconstruct phylogenetic trees, we applied the methodology in [343] on the PR dataset. That is, we compute the pairwise Average Common Substring with k mismatches (k-ACS) distance [349, 244] between the 27 strings in PR, using the ALFRED-G [343] algorithm, and then apply the neighbor-joining (NJ) algorithm [314] to reconstruct the phylogenetic tree. We apply the methodology to S and to $S' \sim_d S$, $S' \neq S$: intuitively, data utility is preserved well when the phylogenetic tree for S is similar to the one for S'. Following [343], we measured similarity using the normalized Robinson-Foulds (nRF) distance [310].

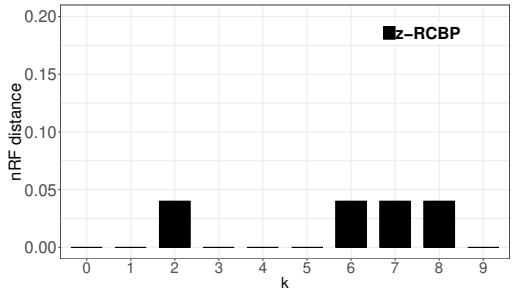
Unless otherwise stated, we used $z = 100$ and $\kappa = 1000$. All experiments ran on a machine with an Intel Xeon E5-2640 at 2.66GHz and 160GB RAM.



(a) MSN



(b) MSN



(c) PR

Figure E.5: Number of (a) frequent patterns and (b) closed frequent patterns with up to $w \in [0, 3]$ wildcards mined from MSN using $\mathsf{minSup} = 3.1 \cdot 10^{-3}$. The length of the longest mined pattern is on the top of each bar. (c) nRF distance vs. k between phylogenetic trees constructed for S and for $S' \sim_d S$, $S' \neq S$, using the k-ACS distance.

E.6.3 Data Utility

Recall that our approach allows for answering pattern matching queries of length at most d in optimal time, and at the same time it prevents the reconstruction of the original dataset. In this section, we demonstrate that z-RCBP (and z-RC, z-RCB, which by design create the same output as z-RCBP), allow for other meaningful data analysis tasks to be applied with insignificant or no utility loss.

Length d.

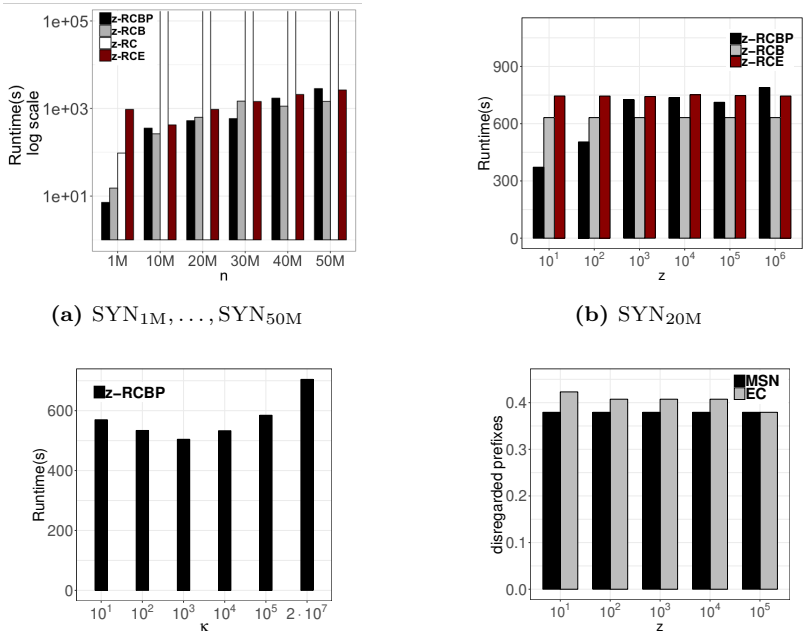
We first show that z-RCBP provides access to very long substrings of the original dataset (i.e., the output length d is large). Figs. E.4a and E.4b show d for different values of the privacy threshold z in EC and MSN, respectively. As expected, d decreases when z increases. However, d is in the order of several hundreds, even when z is set to $100,000$. This implies (I) no accuracy loss for applying the pattern matching queries described in Section E.3 on very long substrings and (II) strong privacy against dataset reconstruction.

Frequent Pattern Mining.

We demonstrate that z-RCBP allows for accurately mining frequent and closed frequent patterns with up to $w \in [0, 3]$ wildcard letters at very low minSup values. To this aim, we have computed the *smallest possible* value of minSup such that the mined frequent and closed frequent patterns have length no more than d. We denote this value by τ. Clearly, our method has no data utility loss for any $\mathsf{minSup} \geq \tau$. For EC, the smallest such minSup value (up to 8 decimal digits) was $\tau = 1.8 \cdot 10^{-6}$. Figs. E.4c and E.4d show the number and the maximal length of the mined patterns with $\mathsf{minSup} = \tau = 1.8 \cdot 10^{-6}$ for EC. For MSN, the smallest such minSup value (up to 4 decimal digits) was $\tau = 3.1 \cdot 10^{-3}$. The results for mining MSN with $\mathsf{minSup} = \tau = 3.1 \cdot 10^{-3}$ in Figs. E.5a and E.5b are qualitatively similar to those in Figs. E.4c and E.4d, respectively. The plots show that a large number of (potentially interesting) patterns can still be mined from the randomly selected S', even if some of them occur a small number of times in S (since τ was very low). Thus, our method permits the fundamental task of frequent pattern mining to be performed accurately.

Phylogenetic Tree Reconstruction.

We next demonstrate that z-RCBP leads to phylogenetic trees constructed from $S' \sim_d S$, $S' \neq S$, which are either the same or very similar with respect to the nRF distance to the phylogenetic trees constructed from S. Fig. E.5c shows the nRF distance between these trees. The trees were obtained using the k-ACS distance for different k values in $[0, 9]$ and the NJ algorithm as in [343]. Note that the tree constructed from S was the same to the one constructed from S' in six out of ten cases, implying no data utility loss, and in the remaining four cases the nRF had a very small value of 0.04, implying insignificant data utility loss for this fundamental bioinformatics task.



(a) $\mathrm{SYN}_{1\mathrm{M}}, \ldots, \mathrm{SYN}_{50\mathrm{M}}$ **(b)** $\mathrm{SYN}_{20\mathrm{M}}$

(c) $\mathrm{SYN}_{20\mathrm{M}}$ **(d)** MSN and EC

Figure E.6: Runtime vs. (a) n (z-RC did not finish within 48 hours for $\mathrm{SYN}_{10\mathrm{M}}, \ldots, \mathrm{SYN}_{50\mathrm{M}}$). Runtime vs. (b) z and (c) κ. (d) Ratio of disregarded prefixes vs. z.

E.6.4 Runtime

In this section, we show that, despite the n^ω factor, z-RCBP takes only a few minutes to finish for million-letter texts. Fig. E.6a shows the runtime of z-RC, z-RCB, z-RCE, and z-RCBP using the synthetic datasets as input. Recall that the largest synthetic dataset is SYN and the other datasets are prefixes of SYN. z-RCBP was substantially more efficient than all other algorithms and scaled better with the dataset size, confirming the necessity of Improvements I and II for being able to apply our methodology to large texts. On the other hand, z-RC was the slowest, and it did not finish within 48 hours for $\mathrm{SYN}_{10\mathrm{M}}, \ldots, \mathrm{SYN}_{50\mathrm{M}}$. Note that z-RCE was more efficient than z-RC, since it performs fewer iterations. The reason is that $d < 20$ for these datasets. In addition, z-RCE was comparable to z-RCB. The reason is that d was very close to $r(S) + 1$.

We also measured the runtime of z-RCB, z-RCE, and z-RCBP for different z values (see Fig. E.6b). We do not report the runtime of z-RC because it did not finish within 48 hours. The runtime of z-RCBP is much less when z is small, because z-RCBP considered fairly short prefixes. Specifically, z-RCBP was two times faster than z-RCB on average, and three times faster when $z = 10$. The runtime of z-RCB and of z-RCE was not affected substantially by z. This is because these algorithms output the same d as z-RCBP for all z values (i.e., they construct the same output) but operate on the entire string S. z-RCB was comparable to z-RCE, for the reason explained above.

Next, we studied the impact of the initial prefix length κ on the runtime of z-RCBP,

Dataset	z-RCB	z-RCE	z-RC	z-RCBP
MSN	438.49	421.96	659.17	347.34
EC	364.84	725.26	571.8	339.18

Table E.2: Runtime (in seconds) for all algorithms on MSN and EC.

the only method that uses Improvement II (see Fig. E.6c). The runtime of z-RCBP decreased when κ increased, but only up to $\kappa = 1000$. Until then, prefixes were too short (i.e., the condition $\alpha_d(S') \geq z$ did not hold), so longer prefixes were considered. For $\kappa > 1000$, z-RCBP took more time because it is more expensive to check the condition on longer prefixes (e.g., z-RCBP took 40% more time when $\kappa = |S|$ compared to when $\kappa = 1000$).

Similar results were observed for the other datasets. For example, as can be seen in Table E.2, in the case of MSN, z-RCBP was again the fastest algorithm, z-RC the slowest, and z-RCB and z-RCE performed similarly. In the case of EC dataset, however, z-RCE was the slowest of all algorithms. The reason is that EC led to a larger d than that of MSN, so z-RCE had to perform more iterations.

E.6.5 Disregarded Prefixes

Last, we demonstrate that, applying Improvement I on the prefixes of S, which are used in Improvement II, allows for disregarding a large ratio of them from the computation. That is, we often avoid computing $\alpha_d(P)$ for a prefix P of S, because when $d \geq r(P)+2$, we have that $\alpha_d(P) = 1 < z$ by Lemma 68. Specifically, Fig. E.6d shows that the ratio of disregarded prefixes over all prefixes considered for MSN and EC is at least 0.38. The benefit of the improvement when this ratio is large is time efficiency, since computing $\alpha_d(P)$ to check whether $\alpha_d(P) \geq z$ can be expensive particularly for a long prefix P of S.

E.7 Application to Adversary Models

In this section, we discuss adapted versions of our z-RSDS that can be applied to two different adversary models, in which an adversary possesses positive and negative knowledge, respectively. In the context of z-anonymity, positive adversarial knowledge has been considered in [341, 255, 301, 274, 340] and negative adversarial knowledge in [249, 35]. Unlike our work, none of these works considers adversarial knowledge in the context of a string.

E.7.1 Adversary Model I: Positive Adversarial Knowledge

Our privacy goal is to limit the probability of inferring string S when the adversary possesses the following knowledge.

Definition 23 (Positive Adversarial Knowledge)**.** *A pair* $\mathcal{K} = (\mathcal{T}_d(S'), \tilde{S})$*, where* $S' \sim_d S$ *and* $\tilde{S}$ *is a (possibly empty) substring of* S*.*

The adversarial knowledge $\mathcal{K}$ is comprised of $\mathcal{T}_d(S')$, which is accessible by the adversary, and of $\tilde{S}$ which is the adversary's *background knowledge*. Background knowledge is obtained by an adversary, typically from external data sources and/or communication with individuals represented in the input dataset [341, 255, 301, 274, 340]. As it will become clear later, such knowledge may make a z-RSDS more likely to be reversed. Thus, when certain background knowledge is known or can be assumed, it should be modeled and taken into account in the construction of a z-RSDS to ensure that the z-RSDS remains sufficiently unlikely to reverse.

We model the background knowledge as a substring to capture manifested attacks [255, 340] in which the adversary observes an individual's actions within a time-frame. The actions are represented by $\tilde{S}$. For example, when S models the diagnoses in an individual's electronic health record, $\tilde{S}$ models the diagnoses assigned to the individual during a hospital visit, which may be known by a hospital employee [255]. Similarly, when S models an individual's credit card purchases, $\tilde{S}$ models the products purchased by the individual during a visit to a shop, which may be known by a shop employee [340]. $\tilde{S}$ may be specified by the data provider [53] or the data custodian [342], according to policies. Note that from $\mathcal{T}_d(S')$, the adversary can also learn (see Fig. E.1): the length $n = |S|$, the maximal string depth d, and the suffixes of S of length at most $d-1$. Thus, we did not include such information in $\mathcal{K}$ explicitly.

An adversary may not be able to uniquely infer S, based on their knowledge $\mathcal{K}$. This is because they have to distinguish S among the set of strings that are d-equivalent to S and have $\tilde{S}$ as substring. In fact, the probability that the adversary infers S, based solely on their knowledge $\mathcal{K}$, is defined as follows.

Definition 24 (Inference Probability of S)**.** *The* inference probability *of S, based on the knowledge $\mathcal{K}$, is defined as $\mathcal{P}(I_S \mid \mathcal{K}) = 1/|\mathcal{A}_\mathcal{K}|$, where I_S is the event "the adversary infers S" and $\mathcal{A}_\mathcal{K}$ is the set of strings consistent with $\mathcal{T}_d(S')$ having $\tilde{S}$ as substring.*

$\mathcal{P}(I_S \mid \mathcal{K})$ is defined based on: (I) The fact that the adversary can construct all strings that are consistent with $\mathcal{T}_d(S')$ and contain $\tilde{S}$ as substring (see Section E.7.1). (II) The *random worlds assumption* [37] (i.e., each such instance is equally likely). This assumption is followed by most related works (see [367] and references therein).

We aim at constructing a $\mathcal{T}_d(S')$, for some $S' \sim_d S$ chosen at random, that an adversary cannot use to infer S with sufficiently large $\mathcal{P}(I_S \mid \mathcal{K})$. We also require $\mathcal{T}_d(S')$ to have maximal d in order to support the operations discussed in Section E.3 on larger substrings, thereby providing higher utility. This leads to the following computational problem.

Problem 2. *Given a string S of length n, a substring $\tilde{S}$ of S, and a privacy threshold $1 < z \leq n^c$, for some constant $c \geq 1$, construct a $\mathcal{T}_d(S')$ such that: (I) $S' \sim_d S$, (II) d is maximal, and (III) $\mathcal{P}(I_S \mid \mathcal{K}) \leq \frac{1}{z}$, for $\mathcal{K} = (\mathcal{T}_d(S'), \tilde{S})$; or output* FAIL *if no such d exists.*

Construction Algorithm

We next show how the z-RC algorithm for constructing a z-RSDS can be applied in an extended way to solve Problem 2. In this case, we need to account for $\mathcal{A}_\mathcal{K}$, a modified version of $\mathcal{A}_d$: a string S' is in $\mathcal{A}_\mathcal{K}$ if and only if it is d-equivalent to S and contains $\tilde{S}$

as a substring. In the graph formulation of the problem, we need to ensure that a path representing $\tilde{S}$ must always be visited. Thus, we modify the z-RC algorithm as follows.

Let $\alpha_{\mathcal{K}} = |\mathcal{A}_{\mathcal{K}}|$. Consider a binary search iteration for some value of d, in which we must check whether $\alpha_{\mathcal{K}} \geq z$. We first construct the weighted de Bruijn graph $G_{S,d}$. If $|\tilde{S}| \leq d$ all strings in $\mathcal{A}_d$ contain $\tilde{S}$ as a substring by construction and so we do not need to modify the algorithm for such an $\tilde{S}$. Intuitively, in this case, knowledge of $\tilde{S}$ is of no use to the adversary. We thus consider the case when $|\tilde{S}| > d$. A substring $\tilde{S}$ of length $|\tilde{S}|$ is represented by a path $v_1 v_2 \dots v_h$ in $G_{S,d}$, with $h = |\tilde{S}| - d + 2$, as the first node v_1 represents a prefix of $|\tilde{S}|$ of length $d-1$, and each traversed edge appends one letter to this prefix. We remove the edges of a path $v_1 v_2 \dots v_h$ in $G_{S,d}$ representing *some occurrence* of $\tilde{S}$ in S, $|\tilde{S}| > d$. (There may be multiple such paths). We add a *shortcut edge* $e_{\tilde{S}} = (v_1, v_h)$ directed from node v_1 to node v_h to represent an occurrence of string $\tilde{S}$. We refer to this procedure as *collapsing* the path $v_1 v_2 \dots v_h$. Let us denote the resulting graph by $G_{S,d,\tilde{S}}$ (see Fig. E.7 vs. Fig. E.3).

Lemma 70. *(a) If $S \sim_d S'$ and $\tilde{S}$ is a substring of S', then S' corresponds to an Eulerian path in $G_{S,d,\tilde{S}}$ that starts from vertex s and ends at vertex $t \neq s$ (if $s = t$, then it corresponds to an Eulerian cycle starting from s). (b) $\alpha_{\mathcal{K}}$ is equal to the number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d,\tilde{S}}$.*

Proof. (a) Trivial. (b) We first observe that $G_{S,d,\tilde{S}}$ is Eulerian (resp. semi-Eulerian) by construction, because $G_{S,d}$ is Eulerian (resp. semi-Eulerian) and the procedure above does not change the parity of any vertex. Indeed, consider path $v_1 v_2 \dots v_h$ in $G_{S,d}$ representing the string $\tilde{S}$, which we replace with a shortcut edge $e_{\tilde{S}}$. Exactly one outgoing and one incoming edge is removed from each $v_2, \dots, v_{h-1}$; one outgoing edge is removed from v_1 and replaced with outgoing edge $e_{\tilde{S}}$, one incoming edge is removed from v_h and replaced with $e_{\tilde{S}}$ incoming. Moreover, since the multiplicity of any substring U of length d is given by the multiplicity of the edge from node $U[0\,..\,d-2]$ to node $U[1\,..\,d-1]$, the multiplicities of d-substrings are not affected by this transformation.

To show that the number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d,\tilde{S}}$ is at most $\alpha_{\mathcal{K}}$, consider any Eulerian path (resp. cycle) in $G_{S,d,\tilde{S}}$. By definition, there is at least one occurrence of $\tilde{S}$ given by edge $e_{\tilde{S}}$, and it thus represents a string that belongs to $\mathcal{A}_{\mathcal{K}}$. Symmetrically, to show that $\alpha_{\mathcal{K}}$ is at most the number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d,\tilde{S}}$, consider a string $U \in \mathcal{A}_{\mathcal{K}}$. Among the (possibly multiple) Eulerian paths (resp. cycles) in $G_{S,d}$ that represent U, consider one that has $v_1 v_2 \dots v_h$ as subpath as representative of its equivalence class: such path exists because of Observation 40 and the properties of weighted de Bruijn graphs. This path corresponds to the path in $G_{S,d,\tilde{S}}$, where $v_1 v_2 \dots v_h$ is replaced with $v_1 v_h$. □

Theorem 43. *Problem 2 can be solved in $O(n^\omega \log d)$ time.*

Proof. The correctness of the construction algorithm follows by Lemma 70 and the fact that it is correct to apply binary or exponential search due to the monotonicity of $\alpha_{\mathcal{K}}$ (the monotonicity proof is almost identical to that of Lemma 64 and is thus omitted).

For a given d, finding a path $v_1 v_2 \dots v_h$ in $G_{S,d}$ and replacing it with $v_1 v_h$ can be done while constructing $G_{S,d}$ at no extra cost. Recall that $v_1 v_2 \dots v_h$ represents some

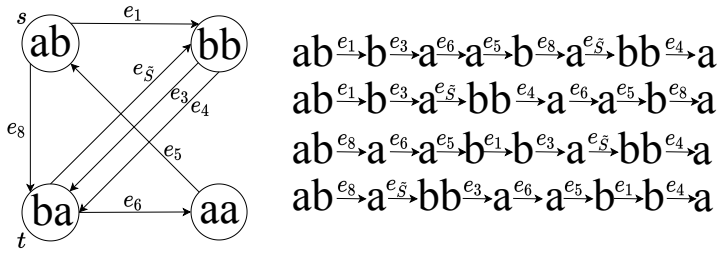


Figure E.7: $G_{S,d,\tilde{S}}$ with $S = \texttt{abaabbabba}$, $d = 3$, $\tilde{S} = \texttt{babb}$, and $\alpha_{\mathcal{K}} = 4$.

occurrence of string $\tilde{S}$ in S, and that all occurrences of $\tilde{S}$ in S can be found in $O(n)$ time using the suffix tree of S [357]. The time complexity thus follows from the proof of Theorem 37. □

E.7.2 Adversary Model II: Negative Adversarial Knowledge

In this section, we consider the case where an adversary possesses negative knowledge. The adversary model is somewhat dual to the one defined in Section E.7.1. Specifically, in our model, the adversary possesses the following *negative* knowledge.

Definition 25 (Negative Adversarial Knowledge)**.** *A pair $\bar{\mathcal{K}} = (\mathcal{T}_d(S'), R)$, where $S' \sim_d S$ and R is a nonempty string that does not occur in S.*

It should be clear that the background knowledge in Definition 25 may be used to reverse the z-RSDS, and it should therefore be modeled and taken into account in the construction of a z-RSDS. A procedure that is entirely analogous to the one described in Section E.7.1 can be used to ensure that the z-RSDS remains sufficiently unlikely to reverse, as we explain next.

We now have to account for $\mathcal{A}_{\bar{\mathcal{K}}}$, which is again a modified version of $\mathcal{A}_d$: a string S' is in $\mathcal{A}_{\bar{\mathcal{K}}}$ if and only if it is d-equivalent to S and does not contain R as a substring. The inference probability $\mathcal{P}(I_S \mid \bar{\mathcal{K}})$, based on the negative knowledge $\bar{\mathcal{K}}$, is defined in much the same way as the one for positive adversarial knowledge: $\mathcal{P}(I_S \mid \bar{\mathcal{K}}) = 1/|\mathcal{A}_{\bar{\mathcal{K}}}|$. Once again, the problem is to construct a $\mathcal{T}_d(S')$ such that $S' \sim_d S$, d is maximal and $\mathcal{P}(I_S \mid \bar{\mathcal{K}}) \leq 1/z$.

Problem 3. *Given a string S of length n, a string R that does not occur in S, and a privacy threshold $1 < z \leq n^c$, for some constant $c \geq 1$, construct a $\mathcal{T}_d(S')$ such that: (I) $S' \sim_d S$, (II) d is maximal, and (III) $\mathcal{P}(I_S \mid \bar{\mathcal{K}}) \leq \frac{1}{z}$, for $\bar{\mathcal{K}} = (\mathcal{T}_d(S'), R)$; or output* FAIL *if no such d exists.*

Let $\alpha_{\bar{\mathcal{K}}} = |\mathcal{A}_{\bar{\mathcal{K}}}|$, and consider a binary search iteration for some value of d, in which we must check whether $\alpha_{\bar{\mathcal{K}}} \geq z$. We construct a graph $G_{S,d,R}$ exactly as if R was required to occur in S', as described in Section E.7.1. By definition, all the strings corresponding to non-equivalent Eulerian paths in $G_{S,d,R}$ contain at least one occurrence of R: the complement of such set in $\mathcal{A}_d$ is the set of d-equivalent strings that do not contain any occurrence of R, which is precisely what we aim at. The following lemma thus connects $\alpha_{\bar{\mathcal{K}}}$ with the number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d}$ and $G_{S,d,R}$. The proof is similar to the one of Lemma 70 and is therefore omitted.

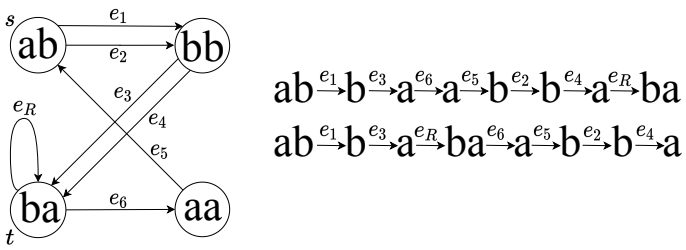


Figure E.8: $G_{S,d,R}$ with $S = \texttt{abaabbabba}$, $d = 3$, $R = \texttt{baba}$, and so $\alpha_{\bar{\mathcal{K}}} = \alpha_d - \alpha_{\mathcal{K}} = 6 - 2 = 4$.

Lemma 71. *$\alpha_{\bar{\mathcal{K}}}$ is equal to the difference between the number α_d of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d}$ and the number $\alpha_{\mathcal{K}}$ of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d,R}$.*

Notice that, in contrast with the case of positive adversarial knowledge, it may happen that R is not represented by any path in $G_{S,d}$: in this case, we simply set $\alpha_{\bar{\mathcal{K}}} = \alpha_d$, as no string $S' \sim_d S$ has R as a substring. Figure E.8 illustrates an example where R is actually represented in $G_{S,d}$: the four strings $S' \sim_d S$ that do not have R as a substring are, in fact, the ones depicted in Figure E.7. The following theorem can be proved much the same way as Theorem 43.

Theorem 44. *Problem 3 can be solved in $O(n^\omega \log d)$ time.*

E.7.3 Generalization to a Collection of Patterns

Both the model of positive and the model of negative adversarial knowledge can be straightforwardly extended to the case of multiple disjoint pattern occurrences in S. We begin by defining the notion of disjoint pattern occurrences in S, as follows:

Definition 26 (Disjoint Pattern Occurrences)**.** *Given a string S and a pair of strings (S_1, S_2), we call (i_1, i_2)* disjoint pattern occurrences *of S_1 and S_2 in S, when S_1 occurs at position i_1 in S, S_2 occurs at position i_2 in S, and the intervals $[i_1, i_1 + |S_1| - 1]$ and $[i_2, i_2 + |S_2| - 1]$ are disjoint.*

Formally, in the positive adversarial knowledge model, the adversary possesses the following knowledge: a collection $\mathcal{S}$ of k patterns such that there exists an ordering $S_1, S_2, \ldots, S_k$ of these patterns that forms a sequence of pairwise disjoint pattern occurrences in $\mathcal{S}$.

The case of multiple disjoint pattern occurrences we consider is of practical importance. It can correspond to a collection of adversary observations within multiple disjoint time-frames; each time-frame corresponds to a pattern, and thus the patterns satisfy Definition 26. In fact, the examples of attacks discussed in Section E.7.1 naturally generalize to multiple patterns. For example, S_i can model the diagnoses assigned to an individual during the i-th hospital visit of the individual.

It can be readily verified that a solution for the case when an adversary possesses a collection $\mathcal{S}$ can be obtained by repeating the procedure of collapsing patterns in the de Bruijn graph to obtain $G_{S,d,\mathcal{S}}$: this is possible because all such patterns are edge-wise

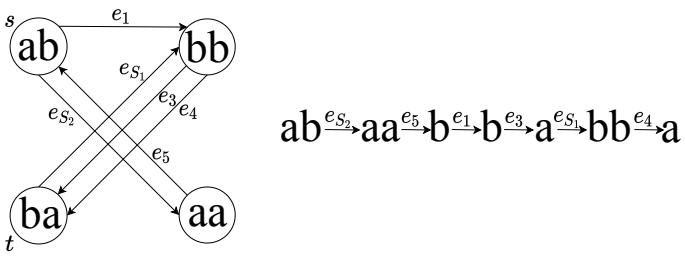


Figure E.9: G_{S,d,S_1,S_2} with $S = \texttt{abaabbabba}$, $d = 3$, $S_1 = \texttt{babb}$, $S_2 = \texttt{abaa}$ and so the only Eulerian path in the graph spells S itself.

disjoint, thanks to the condition that any two patterns have disjoint pattern occurrences in S. In the example of Figure E.9 the knowledge of two non-overlapping substrings of S is sufficient to uniquely identify S, thus to make the construction algorithm fail for any $z > 1$. Note that in this example the adversary knows an additional string S_2 compared to the example of Figure E.7, and this additional knowledge makes the construction algorithm fail.

The negative adversarial knowledge model can be extended in a similar way. In this case, the adversary possesses the following knowledge: a collection of patterns $\mathcal{R} = \{R_1, R_2, \ldots, R_k\}$ such that no permutation of these patterns occurs in S with pairwise disjoint occurrences. Since this condition is precisely complementary to the positive knowledge for multiple patterns, we follow the same strategy as for the negative knowledge model for one pattern: we construct the graph $G_{S,d,\mathcal{R}}$, compute the number of non-equivalent Eulerian paths in this graph, and subtract it from α_d.

E.8 A z-RSDS for Decision Queries

Let us recall that a nonempty string R that does not occur in a string S is called *absent* from S, and it is called *minimal absent* if furthermore all the proper substrings of R do occur at least once in S. Minimal absent words (MAWs) are used in many applications [325, 302, 145, 163, 90, 290, 119] and their theory is well developed [273, 144, 146], also from an algorithmic and data structure point of view [272, 114, 43, 94, 93, 36, 155, 44, 117]. For example, it is well known that, given two strings S and S', one has $S = S'$ if and only if S and S' have the same set of MAWs [273].

We now prove that all strings in the same d-equivalence class have precisely the same set of MAWs of length up to d (and therefore any two distinct d-equivalent strings must have some longer MAW not in common).

Lemma 72. *Let S be a string and R a minimal absent word of S of length at most d. If S' is d-equivalent to S, then R is a minimal absent word of S'.*

Proof. Let us first assume that the length of R is exactly d. Given a de Bruijn graph G of order d, we call edge (u, v) *fake-feasible* if it does not exist in G but it *could* exist in G, i.e., if u and v overlap by $d-2$ letters. It holds that a string R of length d is a MAW of S if and only if $R = u \cdot v[d-2]$ and (u, v) is a fake-feasible edge in the de Bruijn graph G of order d of string S. This follows by the definition of MAWs: all proper

substrings of R (in particular its longest proper prefix u and suffix v) do occur in S but R does not. Since all strings in the d-equivalence class of S have the same G, it follows that they have the same set of fake feasible edges and hence the same set of MAWs of length d. The statement follows from the fact that if two strings are d-equivalent, then they are also $(d-1)$-equivalent [222]. □

Example 33. *All strings in Fig. E.3 have the same MAWs of length up to 3, namely* `aaa` *and* `bbb`*. Let us remark that Lemma 72 is not a characterization of a d-equivalence class:* `aaa` *and* `bbb` *are also the MAWs of length up to 3 of string* `aabaababba`*, which is not d-equivalent to the strings of Fig. E.3.*

Let us now describe an application of Lemma 72. We start with a straightforward fact.

Fact 45. *A string X occurs in a string Y if and only if no minimal absent word of Y occurs in X. Equivalently, a string X does not occur in a string Y if and only if a minimal absent word of Y occurs in X.*

Let S be a string of length n and d be an integer so that the number of distinct strings that are d-equivalent to S is at least z. (Note that S is d-equivalent to itself.) Then Lemma 72 and Fact 45 tell us that the set of MAWs of S of length up to d suffices to construct a z-RSDS over S for d that answers decision pattern matching queries of length $m \leq d$ in the optimal $O(m)$ time per query. In particular, we can construct the Aho-Corasick (AC) automaton [20, 129] of the set of MAWs of S of length up to d in $O(M)$ time, where M is the total length of these MAWs. Then, given a string P of length m, we check if any MAW of S of length up to d occurs in P in $O(m)$ time. By Fact 45, we give a positive answer if no MAW of S of length up d occurs in P. The size of this z-RSDS is the size of the AC automaton, which is $O(M)$ and can be sublinear in $|S| = n$. Note that M can be computed in $O(n)$ time using the data structure presented in [94]. Thus, if $M < n$, we can construct the AC automaton instead of the z-RSDS presented in Section E.3. The data structure of [94] can also enumerate the set of MAWs of S of length up to d in the optimal $O(n + M)$ time.

Let us conclude this section with an example.

Example 34. *Recall that all strings in Fig. E.3 have the same MAWs of length up to $d = 3$, namely* `aaa` *and* `bbb`*, and let $z = 6$. The AC automaton of $\{$*`aaa`*,* `bbb`*$\}$ is a z-RSDS over S answering decision pattern matching queries of length $m \leq d = 3$. Given, for instance, query $P =$* `aba`*, we check that the AC automaton does not locate any occurrence of $\{$*`aaa`*,* `bbb`*$\}$ in P, and thus we return a positive answer. Given query $Q =$* `aaa`*, we check that the AC automaton locates an occurrence of* `aaa` *from $\{$*`aaa`*,* `bbb`*$\}$ at position 0 of Q, and thus we return a negative answer. Note that $M = 6 < n = 10$.*

E.9 Final Remarks

We have introduced the notion of z-reverse-safe data structures (z-RSDSs) and presented such data structures for text indexing. Let us remark that our encoding model can be used in conjunction with other privacy-preserving techniques (e.g., [53]) to ensure that certain privacy-utility trade-offs are maintained. Let us also remark that the algorithmic

contribution of this chapter is computing d and constructing a string $S' \sim_d S$. With S' at hand, one could construct *any compressed index* over S' (e.g., [180, 237]). Such an index would answer queries of length $m \leq d$, and output FAIL for queries of length $m > d$ by storing d using $\log d$ extra bits.

There are at least four directions for future work:

1. Improve the time complexity of the construction (Theorem 37).
2. Improve the time complexity of the construction (Theorem 37) for certain values of z.
3. In Problem 1, the z-RSDS is truncated at a maximal uniform string depth d. Another related problem definition would be to truncate the z-RSDS at maximal non-uniform string depths.
4. In Section E.8, we proposed a small z-RSDS for decision pattern matching queries of length at most d, when d is given. An open problem is to efficiently compute the maximal such d.

Appendix F

Hide and Mine in Strings

Key Points

Problem. Current methods for string sanitization, like the ones presented in Chapters C and D, hide confidential patterns introducing, however, a number of spurious patterns that may harm the utility of frequent pattern mining. The main computational problem we address in this chapter is to minimize this harm, based on the fundamental relation between data sanitization and frequent pattern mining in the context of sequential (string) data.

Model. We consider strings output by string sanitization methods like the ones presented in Chapters C and D, where substrings over a fixed alphabet Σ are separated by a special letter $\# \notin \Sigma$. As noted in Chapter C, the occurrences of # reveal the *locations* of confidential patterns and thus must be ultimately replaced by letters of the original alphabet Σ. However, this replacement may create spurious patterns that could not be mined from the input at a minimum frequency threshold τ but would be mined from the output at the same frequency threshold. These patterns are referred to as τ-*ghosts*. We investigate the crucial interplay between # replacements and τ-ghosts, seeking to replace the #'s with letters in Σ so that the number of τ-ghosts in the resulting string is minimized.

Included Works

This chapter presents the results of the paper **Hide and Mine in Strings: Hardness and Algorithms**, to be submitted to the journal *Hide and Mine in Strings: Hardness and Algorithms*. A short version of this paper [57] has been accepted to the *20th IEEE International Conference on Data Mining (ICDM 2020)*.

F.1 Introduction

A string is a sequence of letters over some alphabet Σ. Strings are commonly used to represent individuals' data in domains ranging from transportation to web analytics and

bioinformatics. For example, a string can represent a user's location profile, with each letter corresponding to a visited location [370], a user's purchasing history, with each letter corresponding to a purchased product [18], or a patient's genome sequence, with each letter corresponding to a DNA base [233]. Mining patterns from such strings is thus useful in a gamut of applications. For instance, mining patterns from location history data helps route planning [97]; mining patterns of co-purchased products from market-basket data improves business decision making [18]; mining patterns from genome sequences can improve clinical diagnostics [233]. To support these applications while preserving privacy, strings representing individuals' data are often being disseminated after sanitization [365, 13] or anonymization [16].

In this chapter, we study the fundamental relation between *data sanitization* [365, 13, 53] (also known as *knowledge hiding*) and *frequent pattern mining* [268, 226, 267, 322]. The objective of frequent pattern mining in strings is to obtain all patterns occurring frequently enough (according to a given frequency threshold τ) in a string, or in a collection of strings. There may also be constraints for the mined strings (e.g., to be of fixed length k [33, 112]). In string sanitization, the privacy objective is to transform a string to ensure that a given set of *sensitive patterns*, modeling confidential knowledge, does not occur in the sanitized version of the string; sensitive patterns are selected based on domain expertise [365, 172, 53]. Typically, this transformation incurs some utility loss that should be minimized. Recently published methods achieve this goal using combinatorial algorithms [53, 56]. Let W be the input string over Σ, k be a positive integer, and $\mathcal{S}$ be the set of sensitive length-k substrings. These methods construct a string X such that (I) X contains no element of $\mathcal{S}$ as a substring; (II) the total order and thus the frequency of all non-sensitive length-k substrings of W is preserved in X; and (III) the length of X is minimized [53], or the edit distance between W and X is minimized [56]. These methods work by copying carefully selected substrings of W into X and separating them by a special letter $\# \notin \Sigma$.

Example 35. *Let* $W = \texttt{GACAAAAACCCAT}$, $k = 3$, *and the set of sensitive patterns* $\mathcal{S} = \{\texttt{ACA}, \texttt{CAA}, \texttt{AAA}, \texttt{AAC}, \texttt{CCA}\}$. *Further, let* $X_{\mathrm{TR}} = \texttt{GAC\#ACC\#CCC\#CAT}$, $X_{\mathrm{MIN}} = \texttt{GACCC\#CAT}$ *and* $X_{\mathrm{ED}} = \texttt{GAC\#AA\#ACCC\#CAT}$ *be three sanitized strings. All three strings contain* no sensitive pattern *and preserve the* total order *and thus the* frequency *of all non-sensitive length-*3 *patterns of* W*:* X_{TR} *is the trivial solution of interleaving the non-sensitive length-*3 *patterns of* W *with* #*;* X_{MIN} *is the* shortest *possible such string [53]; and* X_{ED} *is a string* closest *to* W *in terms of edit distance [56].*

Unfortunately, as noted in [53], the occurrences of # reveal the *locations* of sensitive patterns and thus must be ultimately replaced by letters of the original alphabet Σ. This replacement gives rise to another string over Σ, which we denote by Z. However, this replacement may create spurious patterns that could not be mined from X at a minimum frequency threshold τ but would be mined from Z at the same frequency threshold. These patterns are referred to as τ-*ghosts*.

We investigate the crucial interplay between # replacements and τ-ghosts, posing here the following question that, to the best of our knowledge, has not been addressed: *Given a string* X *containing* #*'s, a positive integer* k*, and a positive integer* τ*, how should we replace the* #*'s in* X *with letters in* Σ*, so that the number of length-*k τ*-ghosts in the resulting string* Z *is minimized?* This question helps preserving the accuracy

of frequent pattern mining and tasks based on it (e.g., pattern-based clustering [182] and classification [306], as well as sequential rule mining [332]) that we may not know a priori.

The above question is also of quite general interest, as it applies to sequential datasets that may have occurrences of a special letter for a variety of reasons beyond data sanitization. This special letter, denoted here by # for consistency, represents some information that is *missing* from these datasets. For instance, in genome sequencing data, # corresponds to an unknown DNA base [211]; in databases, # represents a value that has not been recorded [64, 148]; and in masked data outputted by other privacy-preserving methods [63], # is introduced deliberately to achieve their privacy goal.

Like in data outputted by sanitization methods, the occurrences of # in other string datasets often have to be replaced. For example, since the DNA alphabet consists of four letters (`A`, `C`, `G`, and `T`), off-the-shelf algorithms for processing DNA data use a two-bits-per-base encoding to represent the DNA alphabet. In order to use these algorithms with input strings containing unknown bases, we would have to amend them to work on the extended alphabet $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}, \texttt{\#}\}$. This solution may have a negative impact on the time efficiency of the algorithms or the space efficiency of the data structures they use. Thus, instead, in several state-of-the-art DNA data processing tools (e.g., [248, 246]), the occurrences of # are replaced by an arbitrarily chosen letter from the DNA alphabet, so that off-the-shelf algorithms can be directly employed. This, however, may introduce a large number of spurious patterns, negatively affecting the accuracy of DNA analyses.

Replacing the occurrences of # in a database is often needed to be able to perform frequent pattern mining with off-the-shelf algorithms [148]. To this end, the occurrences of # are commonly replaced by some statistical estimate, such as the most frequent value [148, 181]. However, such a replacement does not generally maintain the accuracy of frequent pattern mining, since it may introduce many spurious patterns [148].

Example 36. *Let again* W=`GACAAAAACCCAT`, k=3, *and* $\mathcal{S} = \{$`ACA`, `CAA`, `AAA`, `AAC`, `CCA`$\}$. *Further, let the frequency threshold be* $\tau = 2$. *Note that the frequency of all non-sensitive length-*3 *patterns in* W *is preserved in all three sanitized strings* X_{TR} = `GAC#ACC#CCC#CAT`, X_{MIN} = `GACCC#CAT`, *and* X_{ED} = `GAC#AA#ACCC#CAT`. *Replacing, however, all #'s with* `G` *would create* τ*-ghost* `GAC` *both in* X_{TR} *and in* X_{ED}.

Contributions. To our knowledge, there does not exist a general solution to the question we pose here that simultaneously guarantees effectiveness and efficiency. In this work, we provide compelling evidence as to why this is the case. Within the string sanitization context, we also provide algorithms for answering this question. Specifically: **1)** We embark on a theoretical study to understand the relation between replacing #'s and creating τ-ghosts. In particular, we define the following problems and examine their hardness:

- HMD (Hide and Mine decision): This is the core decision version of the problem asking whether or not we can replace all #'s in X, so that no sensitive pattern and no τ-ghost occurs in Z. Deciding this may allow for sanitizing X with no utility loss in frequent pattern mining. We show that HMD is *strongly NP-complete* via a reduction from a variant of the well-known Bin Packing problem [164] (see

Section F.4). This is the most technically involved part of the chapter, as the provided reduction is highly non-trivial.

- HM (Hide and Mine): This is the optimization version of HMD asking how we can replace all #'s, while ensuring that no sensitive patterns and a minimal number of τ-ghosts occur in Z. This would minimize the utility loss in frequent pattern mining. HM is clearly NP-hard as a consequence of HMD being NP-complete, but we also show that it is *hard to approximate*.

- HMMT (Hide and Mine minimum threshold): Given a parameter τ, this problem asks for the minimum frequency threshold $\tau_1 \geq \tau$ for which no sensitive pattern and no τ_1-ghost occurs in Z. Solving HMMT would imply no utility loss in frequent pattern mining at a higher frequency threshold τ_1 that is as close as possible to τ. We show that HMMT is (*NP-hard* and) *hard to approximate*.

The hardness (see Section F.4) and inapproximabilty (see Section F.5) results for our problems provide solid evidence for the lack of exact polynomial-time or approximation algorithms for these problems and motivate our next contributions.
2) We develop *exact algorithms* for HMD and HM that require polynomial time, under certain realistic assumptions on the problem parameters. We also develop an efficient and effective heuristic for HM. In particular, we develop the following:

- Exact algorithms based on an Integer Linear Programming (ILP) formulation of HMD. The main idea is to identify all length-k strings over Σ in X that may potentially become τ-ghosts in Z, and then decide whether each of the #'s can be replaced by a letter in Σ without creating any τ-ghost pattern or any sensitive pattern in Z. We prove that HMD is *fixed-parameter tractable* [121] in most cases encountered in practice (e.g., when the number of distinct letters in the string and the length k of sensitive patterns are upper bounded by a constant).

- Exact algorithms based on an ILP formulation of HM. This ILP formulation differs from the HMD formulation in that it takes into account the number of τ-ghosts created by replacing #'s, so as to minimize their number. We prove that HM is fixed-parameter tractable in many cases encountered in practice (e.g., when the length k of sensitive patterns and the number of distinct patterns that may become τ-ghosts are upper bounded by a constant).

- A greedy heuristic that replaces the #'s from left to right, while avoiding the creation of non-sensitive patterns that may become τ-ghosts. The heuristic has three variants which aim to minimize different measures based on: the number of newly created patterns with frequency $f < \tau$, the sum of $(\tau - f)^{-1}$, or the max of $(\tau - f)^{-1}$, where frequency f is taken over the newly created patterns.

The ILP-based algorithms are presented in Section F.6 and the greedy heuristic in Section F.7.
3) We conduct an extensive experimental study (see Section F.8). We show that our methods: (I) allow for frequent length-k pattern mining with no or insignificant utility loss (i.e., they create zero or few τ-ghosts); (II) incur very low distortion; and (III) are

practical, requiring, for instance, less than 2.5 seconds to process a 20-million letter string.

Outlook. Frequent pattern mining is also applied on data types other than strings, such as graphs, trees, itemsets etc. [322]. Given the fact that string is the most basic and perhaps most widely used data type, our hardness results support the intuition that replacing missing values with no utility loss for frequent pattern mining in these more complex data types may not be possible in polynomial time; based on our results, we further anticipate that it might even be hard to approximate such solutions in polynomial time. Given the successful deployment of ILP in string representations presented in this chapter, ILP might be a promising strategy to be applied for replacing missing values in other data representations and settings. Thus, this work may inspire further research on the topic of replacing missing values for frequent pattern mining.

F.2 Related Work

Our work focuses on sanitization, a privacy-preserving data mining technique aiming to prevent the mining of confidential knowledge from published data [365, 13, 256, 53, 56]. Data sanitization approaches are typically applied to a collection of transactions [365, 173, 333], a collection of sequences [13, 172], or a single sequence [256]. These approaches employ integer programming [173, 333], dynamic programming [256], or heuristics [365, 13, 172, 187]. The objective of these approaches is twofold: to reduce the frequency (support) of sensitive patterns, so that they cannot be mined at a given frequency threshold τ; and to preserve data utility, often by preserving the set of frequent patterns that can be mined at threshold τ [173, 333, 365, 172, 187]. The patterns considered in these approaches are: itemsets in [365, 173, 333], subsequences in [13, 172, 187], and single letters in [256]. Unlike our work, these approaches do not aim at hiding sensitive strings, nor at minimizing changes to the set of frequent substrings.

As discussed in Introduction, there are two recently proposed approaches for string sanitization [53, 56]. In these approaches, #'s must be ultimately replaced so that the locations of sensitive patterns are not exposed. To this end, [53] considered the problem of replacing #'s so as to minimize the total weight of τ-ghost occurrences. It further showed that this problem is NP-hard and proposed a heuristic inspired by algorithms for Multiple-Choice Knapsack. Note that HM, the problem of minimizing the *total number* of τ-ghosts we consider here, is fundamentally different from the problem of minimizing the *total weight* of τ-ghost occurrences, and it cannot be reduced from Multiple-Choice Knapsack, because no arbitrary weights or costs are involved. On the hardness side, this makes our hardness proof considerably more challenging. On the algorithmic side, our algorithms significantly outperform the heuristic of [53], as shown in our experiments.

As discussed in Introduction, our work is also related to missing value treatment, which is often required to improve the quality of obtained statistics [252], query answers [64], and data mining models (e.g., association rules [86, 6], sequential patterns [148], clustering [348], and classification [128]). To deal with these problems, existing works remove [252] or replace missing values [128, 348, 64], or alternatively utilize interestingness measures that are suited to mining patterns with missing values [86, 148]. Hence these works are tailored to specific settings and cannot deal with

our problem; they do not aim at minimizing the impact of replacing missing values in a string on frequent pattern mining.

F.3 Preliminaries and Problem Statement

We consider alphabets Σ and $\Sigma_{\#} = \Sigma \cup \{\#\}$, where $\#$ is a special letter not in Σ. We fix a *string* $X = X[0] \cdots X[n-1]$ of length $|X| = n$ over $\Sigma_{\#}$. $\mathrm{Freq}_X(U)$ denotes the number of occurrences (starting positions) of string U as a substring of X. A *dictionary* over Σ is a set of strings over Σ. The dictionary used in our work is a set of length-k strings that do not occur in X; we refer to these strings as *sensitive patterns*. Any element of Σ^k that is not in this dictionary is referred to as a *non-sensitive pattern*. In combinatorics on words, such a dictionary is known as *antidictionary* and the sensitive patterns are known as *forbidden patterns* (e.g., [118]).

Problem 4 (Hide & Mine (HM))**.** *Given an integer $k > 0$, a string $X = X_0\#X_1\#\cdots\#X_\delta$ of length n over an alphabet $\Sigma_{\#}$, with $|X_i| \geq k-1$, for all $i \in [0, \delta]$, a dictionary $\mathcal{S} \subseteq \Sigma^k$ such that no $S \in \mathcal{S}$ occurs in X, and an integer $\tau > 0$, compute a function $g : [\delta] \to \Sigma$ such that the following hold for string $Z = X_0 g(1) X_1 g(2) \cdots g(\delta) X_\delta$:*

I The number of strings $U \in \Sigma^k$, with $\mathrm{Freq}_X(U) < \tau$ and $\mathrm{Freq}_Z(U) \geq \tau$ in Z, is minimized.

II No $S \in \mathcal{S}$ occurs in Z.

Note that function g replaces each $\#$ by exactly one letter from Σ. Condition $|X_i| \geq k-1$ means that any two $\#$'s in X are at least k positions apart. Thus, any length-k substring $X[i\,..\,i+k-1]$ of X is affected by at most one $\#$ replacement. The sanitization method of [53, Lemma 1] produces an X satisfying this condition, for *any* given set $\mathcal{S}$, to guarantee that the frequency of every non-sensitive pattern is preserved in X. Thus, HM is directly applicable to the output of [53].

A string $U \in \Sigma^k$ with $\mathrm{Freq}_X(U) < \tau$ and $\mathrm{Freq}_Z(U) \geq \tau$ is referred to as *τ-ghost*. To prove NP-completeness, we consider the decision variant HMd of HM, which asks to decide if there exists any function $g : [\delta] \to \Sigma$ such that the following hold:

I No τ-ghost pattern occurs in Z.

II No $S \in \mathcal{S}$ occurs in Z.

F.4 HMd is NP-complete

Problem HMd is clearly in NP. In this section, we show that HMd is strongly NP-complete via exhibiting a reduction from a variant of the Bin Packing problem [164]. As a consequence, HM is NP-hard. Notice that, since HM assumes that any two occurrences of $\#$ are at least k positions apart, then clearly, the more general problem (the question posed in Introduction), in which this restriction is waived, is also NP-hard.

F.4.1 The Unique-Weights Bin Packing problem

The Bin Packing (BP) problem is defined as follows. Given three positive integers, M (number of bins), B (capacity of every bin), and N (number of items), as well as a vector $[w_1, \ldots, w_N]$ of positive integers representing the weights of the items, the BP problem asks whether we can partition the items into M subsets (bins) without exceeding the capacity of any bin. Formally, we need to decide whether there exists a function $f : [N] \to [M]$, assigning items to bins, such that:

$$\forall i \in [M], \sum_{j \in [N], f(j)=i} w_j \leq B.$$

It is crucial to remark that BP is *strongly* NP-complete [164], i.e., it is NP-complete even when weights and bin capacities are bounded by a polynomial function of N and M. In the following, we will consider this case, and use gadgets whose size is proportional to the numerical values in $\mathcal{I}_{\text{BP}}$, as if we were representing those numbers in unary notation.

We will assume that no two items have the same weight.[1] We refer to this variant of BP as the Unique-Weights Bin Packing (UWBP) problem. To justify the assumption, we show that the UWBP variant is still strongly NP-complete by a polynomial-time reduction from standard BP.

Lemma 73. UWBP *is strongly NP-complete.*

Proof. Consider an instance $\mathcal{I}_{\text{BP}} = M, B, N, w_1, \ldots, w_N$ of BP with possibly duplicated weights, where all values are polynomial in the size of $\mathcal{I}_{\text{BP}}$: we will construct, in polynomial time, an instance $\mathcal{I}'_{\text{BP}} = M', B', N', w'_1, \ldots, w'_{N'}$ of UWBP (where no two weights are the same) that has polynomial values, and has a positive answer if and only if $\mathcal{I}_{\text{BP}}$ does.

To obtain $\mathcal{I}'_{\text{BP}}$ we proceed as follows. Firstly, set $M' = M$, $N' = N$ and $B' = B \cdot N^2 + (N^2 - 1)$. To obtain the weights w'_i multiply each by N^2, then add "flavoring" by taking groups of items with the same weight one by one and, for each group, adding 0 to its first item, 1 to the second, 2 to the third, and so on. Essentially, we increase the scale of the numbers (a 1-weight item becomes N^2-weight) so much that we can make all weights different without affecting the way groups of items fit in bins: the extra $(N^2 - 1)$ capacity in B' does not allow to fit an extra unit of item-weight (that is N^2 weight), but is enough to account for the flavoring of any set of items. Indeed, in the worst case (when all items have the same weight), the cumulative amount of flavoring added to all N items is $0 + 1 + 2 + \ldots + N - 1 < N^2/2$. Hence, an assignment of items to bins is valid for $\mathcal{I}_{\text{BP}}$ if and only if it is valid for $\mathcal{I}'_{\text{BP}}$. □

F.4.2 Overview of the Reduction from UWBP to HMd

We now show that for any UWBP instance, we can produce in polynomial time an instance of HMd that has positive answer if and only if the UWBP instance has positive answer. To this end, we will introduce several gadgets which will serve to

[1] The assumption is not strictly necessary, but it simplifies the reduction.

model the different constraints of UWBP. Each gadget consists of a string of length $2k-1$ over a specific alphabet, with a # in the middle. We will explain how all UWBP constraints are linked to the gadgets. It will then suffice to concatenate the gadgets into one long string, to obtain an instance of HMd that implies a solution of UWBP.

First, we will consider gadgets t_{ij}, which model whether item j is placed in bin i. The structure of these gadgets will ensure that the maximum capacity B of the bins is not exceeded. Then, gadgets u_{ij} will be introduced. The structure of these second kind of gadgets, together with t_{ij}, ensures that each item is placed in some bin.

The set of sensitive patterns $\mathcal{S}$ and the threshold τ will be carefully chosen to build and link these gadgets. Sensitive patterns will be used to force a specific subset of letters to replace a # (by forbidding the length-k strings obtained from unwanted replacements). The threshold is essential in linking the gadgets and modelling the capacity of the bins: since no pattern that occurs fewer than τ times is allowed to exceed that same threshold after the replacements, we will repeat the pattern in the string so as to bound the number of its additional occurrences that can be created by replacing a #.

F.4.3 Construction of an Instance of HMd

The alphabet of the string X of the instance of HMd will be made of letters #, x, y, \$, and a letter b_i for each $i \in [M]$.

For $i \in [M]$, $j \in [N]$, and a suitable value of k to be fixed later, we define the gadgets t_{ij} and u_{ij} as follows:

$$t_{ij} = b_i \underbrace{x \dots x}_{k-1-w_j} \underbrace{b_i \dots b_i}_{w_j-1} \# \underbrace{b_i \dots b_i}_{k-1}$$

$$u_{ij} = b_i \underbrace{x \dots x}_{k-1-w_j} \underbrace{b_i \dots b_i}_{w_j-1} \# \underbrace{y \dots y}_{w_j} \underbrace{x \dots x}_{k-w_j-2} y$$

For the sake of readability, from now on we will write U^ℓ to denote $\underbrace{U \dots U}_{\ell}$; i.e., ℓ concatenations of a string U starting with the empty string. We then define $\mathcal{S}$, the set of sensitive patterns, as the union of the following sets:

1. $\{b_{i'} b_i^{k-1} \mid i, i' \in [M], i' \neq i\}$ which forbids putting a $b_{i'}$ to replace the # in any t_{ij} if $i' \neq i$.
2. $\{b_i y b_i^{k-2} \mid i \in [M]\}$, which forbids putting a y to replace the # in a t_{ij}.
3. $\{b_i \$ b_i^{k-2} \mid i \in [M]\}$, which forbids putting a \$ to replace the # in a t_{ij}.
4. $\{b_i y^{w_j} x^{k-w_j-2} y \mid i \in [M], j \in [N]\}$, which forbids putting any b_i to replace the # in a u_{ij}.
5. $\{b_i \$ y^{w_j} x^{k-w_j-2} \mid i \in [M], j \in [N]\}$, which forbids putting a \$ to replace the # in a u_{ij}.

As explained below, we will use t_{ij} and u_{ij} to construct an instance of string X. By this definition of $\mathcal{S}$, the # in a t_{ij} can only be replaced with b_i or x, and the # in a u_{ij} only with x or y, so that X does not contain sensitive patterns.

We model the size B of the bins using the threshold τ: specifically, we link the filling of the ith bin with the number of occurrences of a specific non-sensitive pattern (namely, b_i^k). However, this is not the only pattern we need to constrain: we have many different length-k substrings that come into play, all of which need specific thresholds. Thus, a common threshold τ for all non-sensitive patterns is too restrictive. We implement this by choosing τ high enough, and artificially lowering the allowed occurrences of each specific non-sensitive pattern by adding an appropriate amount of extra copies of the non-sensitive pattern itself at the end of the string. This way we can choose a different threshold for each non-sensitive pattern.

In accordance with this reasoning, we choose $k = \max_j w_j + 3$ and $\tau = \max\{M, B\} + 1$. We finally construct the string X as a concatenation of the following components, separated by \$\$ as follows:

1. t_{ij}, $\forall i, j$
2. u_{ij}, $\forall i, j$
3. $\tau - B - 1$ occurrences of b_i^k, $\forall i$
4. $\tau - 2$ occurrences of $b_i x^{k-w_j-1} b_i^{w_j-1} x$, $\forall i, j$
5. $\tau - M$ occurrences of $y^{w_j+1} x^{k-w_j-2} y$, $\forall j$.

Component (3) ensures that a valid solution of this instance cannot add more than B occurrences of any b_i^k. Each time we replace the # in a t_{ij} with b_i (intuitively corresponding to assigning item j to bin i), we introduce w_j additional occurrences of b_i^k: this models the consumption of space in each bin, and the limit B ensures that no bin overflows.

By Component (4), for each i, j, only one additional occurrence of $b_i x^{k-w_j-1} b_i^{w_j-1} x$ can be created, either by replacing the # with x in a t_{ij} or in a u_{ij}. This ensures that, if we substitute x for # in one of the two gadgets, then we cannot do the same in the other one. Let us consider a specific item j; if we do not place it in bin i, then we are forced to substitute y for # in u_{ij}, creating an occurrence of length-k substring $y^{w_j+1} x^{k-w_j-2} y$. Since, by Component (5), we can only add $M-1$ occurrences of this latter pattern over all M bins, there must be an i such that the # in u_{ij} is replaced with x. The corresponding # in t_{ij} is then forced to be replaced with a b_i, ensuring that item j is assigned to some bin.

F.4.4 Correctness

We have shown how to construct in polynomial time an instance $\mathcal{I}_{\text{HMD}}$ from any given instance $\mathcal{I}_{\text{UWBP}}$. We now prove that $\mathcal{I}_{\text{HMD}}$ has a positive answer if and only if $\mathcal{I}_{\text{UWBP}}$ does. For the sake of readability, let us refer to the # in t_{ij} and u_{ij} as $\#_{ij}^t$ and $\#_{ij}^u$, respectively. The solution to $\mathcal{I}_{\text{HMD}}$ can then be expressed via a function $g : \{\#_{ij}^t, \#_{ij}^u, \ \forall i, j\} \to \Sigma$. Let $f : [N] \to [M]$ be a solution for a given $\mathcal{I}_{\text{UWBP}}$. We

create the corresponding solution to $\mathcal{I}_{\text{HMD}}$ in the following manner, for each item $j \in [N]$ and bin $i \in [M]$:

$$f(j) = i \Rightarrow g(\#_{ij}^t) = b_i \text{ and } g(\#_{ij}^u) = x;$$
$$f(j) \neq i \Rightarrow g(\#_{ij}^t) = x \text{ and } g(\#_{ij}^u) = y.$$

For a given item j such that $f(j) = i$, we get w_j occurrences of b_i^k, one occurrence of $b_i x^{k-w_j-1} b_i^{w_j-1} x$, and for all $h \neq i$ one occurrence of $b_h x^{k-w_j-1} b_h^{w_j-1} x$ and $y^{w_j+1} x^{k-w_j-2} y$. Since the bin capacity in the solution of UWBP is not overflown, we added at most B copies of b_i^k for each i. Finally, since each element is taken once in UWBP, we created exactly $(M-1)$ occurrences of $y^{w_j+1} x^{k-w_j-2} y$ and one occurrence of $b_i x^{k-w_j-1} b_i^{w_j-1} x$. We thus do not create τ-ghosts, and we have a valid solution for HMD.

Vice versa, given a solution g to our HMD instance, to obtain the solution to the original UWBP it suffices to prove that the following two claims are satisfied:

1. We do not overload any bin; formally

$$\forall i \in [M] \quad \sum_{j \in [N] \text{ s.t. } g(\#_{ij}^t) = b_i} w_j \leq B.$$

2. Each item is assigned to some bin; formally

$$\forall j \in [N] \qquad |\{i \in [M] \text{ s.t. } g(\#_{ij}^t) = b_i\}| \geq 1.$$

If these claims are satisfied, we can extract an assignment for UWBP: for every item j we choose an arbitrary bin i such that $g(\#_{ij}^t) = b_i$, and set $f(j) = i$. By construction of the instance of HMD, these claims are satisfied. By Lemma 73, we obtain the following result.

Theorem 46. HMD *is strongly NP-complete.*

F.5 HM is Hard to Approximate

Given the hardness of HMD, in this section, we shift our focus on checking whether an approximately optimal solution of HM can be obtained instead. Given any instance $\mathcal{I}_M$ of a minimization problem, an algorithm is called an *α-approximation*, for some $\alpha \geq 1$, if it runs in polynomial time in the size of $\mathcal{I}_M$ and always outputs a solution value $\Gamma \leq \alpha \cdot \text{OPT}$, where OPT denotes the optimal value for $\mathcal{I}_M$. We start with the following result.

Theorem 47. *There is no α-approximation algorithm for* HM*, for any $\alpha \geq 1$, unless P=NP.*

Proof. Suppose by contradiction that an α-approximation algorithm A existed for minimizing the number of τ-ghosts in HM. We could then use A to solve HMD: the answer to HMD would be positive (i.e., there would exist a function g that creates 0 τ-ghosts) if and only if the answer of A was $\Gamma = 0 \leq \alpha \cdot \text{OPT} = 0$, which contradicts Theorem 46. □

The reader may now wonder whether the problem becomes easier should one relax the requirement for a *fixed threshold* τ. Thus, the following problem arises naturally.

Problem 5 (HMMT)**.** *Given an integer $k > 0$, a string $X = X_0\#X_1\#\cdots\#X_\delta$ of length n over alphabet $\Sigma_\#$, with $|X_i| \geq k-1$, for all $i \in [0,\delta]$, a dictionary $\mathcal{S} \subseteq \Sigma^k$ such that no $S \in \mathcal{S}$ occurs in X, and an integer $\tau_0 > 0$, compute the smallest integer $\tau_1 \geq \tau_0$ so that there exists a function $g : [\delta] \to \Sigma$, such that the following hold for string $Z = X_0 g(1) X_1 g(2) \cdots g(\delta) X_\delta$:*

I No $U \in \Sigma^k$, with $\mathrm{Freq}_X(U) < \tau_1$ and $\mathrm{Freq}_Z(U) \geq \tau_1$ occurs in Z.

II No $S \in \mathcal{S}$ occurs in Z.

The practical rationale for considering HMMT is that it could be useful if, for instance, τ_1 is only slightly larger than τ in a given HM instance. Unfortunately, we show that HMMT is NP-hard, and it is even hard to approximate.

Theorem 48. HMMT *is NP-hard.*

Proof. We reduce HMD to HMMT as follows. Let $\mathcal{I}_{\text{HMD}}$ be the instance of HMD we would like to solve for some threshold τ. We construct an instance of HMMT consisting of the X, k, and $\mathcal{S}$ from $\mathcal{I}_{\text{HMD}}$, and we also set $\tau_0 = \tau$. We denote this instance by $\mathcal{I}_{\text{HMMT}}$. The reduction takes linear time in the size of HMD. We seek to find the minimum threshold $\tau_1 \geq \tau_0$ such that no length-k substring of Z is a τ_1-ghost. Then $\mathcal{I}_{\text{HMD}}$ has a positive answer if and only if the answer τ_1 of $\mathcal{I}_{\text{HMMT}}$ is equal to $\tau_0 = \tau$. The statement thus follows. □

Observe that a pattern U is a τ-ghost if and only if $\tau \in (\mathrm{Freq}_X(U), \mathrm{Freq}_Z(U)]$. Therefore, the minimal number of τ-ghosts is not monotonous in τ. On the contrary, the minimal number of τ-ghosts is zero when $\tau = 0$ and all patterns are already frequent (i.e., they appear at least τ times), or when $\tau > n$ and the threshold is so high that no pattern can ever become a τ-ghost. In between, the minimal number of τ-ghosts increases whenever τ equals the frequency of some patterns in X, and then slowly decreases again. We will use this behavior, and the fact that HMD is NP-hard, to construct a string for which we cannot determine in polynomial time whether $\tau_1 = \tau_0$ or $\tau_1 > \alpha\tau_0$ (and for which we can prove that $\tau_1 \notin [\tau_0 + 1, \alpha\tau_0]$), implying inapproximability.

Theorem 49. *There is no α-approximation algorithm for* HMMT*, for any $\alpha \geq 1$, unless P=NP.*

Proof. Let X be an arbitrary string and $\mathcal{S}$ be the set of sensitive patterns as defined in HMD. Further, let T be the length-$(k-2)$ suffix of X and Z be a string obtained by replacing the #'s of X. From this instance of HMD, we will construct an instance of HMMT consisting of a string Y and a set $\mathcal{S}'$ of sensitive patterns, so that if an α-approximation algorithm existed for HMMT, we could decide HMD in polynomial time. We define Y over $\Sigma \cup \{\#, \&\}$ to be

$$Y = X(\&\&T)^{\tau_0}\&(\#T\&)^{\lceil(\alpha-1)\tau_0\rceil}.$$

Let $\mathcal{R}$ be the set of all strings $\&sT$, with $s \in \Sigma$. We define the dictionary of sensitive patterns be $\mathcal{S}' = \mathcal{S} \cup \mathcal{R}$. Note that we need to replace all #'s in $(\#T\&)^{\lceil(\alpha-1)\tau_0\rceil}$ by &'s

in order not to introduce any sensitive patterns. However, doing so increases the number of $\&T\&$ patterns (and all other newly created patterns) from τ_0 to $\lceil \alpha\tau_0 \rceil$. Therefore, if $\tau = \tau_0$, then the number of τ-ghosts in Z equals that in $Z(\&\&T)^{\tau_0}\&(\&T\&)^{\lceil(\alpha-1)\tau_0\rceil}$, because the additional new patterns were already occurring at least τ times in Y. However if $\tau_0 < \tau \leq \lceil \alpha\tau_0 \rceil$, then there will always be at least one τ-ghost, namely $\&T\&$. Recall that deciding HMD is NP-complete. Therefore it is NP-complete to decide whether or not $\tau_1 = \tau_0$ or $\tau_1 > \lceil \alpha\tau_0 \rceil$. We conclude that there exists no α-approximation algorithm for HMMT, unless P=NP. □

F.6 Exact Algorithms for HM

We resort to ILP to design exact algorithms for HMD and HM. In particular, we show that both problems are fixed-parameter tractable for several combinations of parameters.

We say that the length-$(k-1)$ substring U preceding an occurrence of # in X, and the length-$(k-1)$ substring V following it, form its *context* UV. Recall that there are δ occurrences of # in X, and that any two occurrences are at least k letters apart, so UV is in Σ^{2k-2}. We assign to every context UV a unique identifier (id). We write $\#_i$ for # in X if its context UV has id i. A string $N \in \Sigma^k$ is *critical* if it may become a τ-ghost, i.e., if an additional occurrence of N can be created by replacing some # by a letter in Σ and $\mathrm{Freq}_X(N) \in [\tau - k\delta, \tau - 1]$. This is because the frequency of N cannot increase by more than $k\delta$, and the frequency of N in X must be less than τ for N to become τ-ghost. We assign to each critical string N a unique id ℓ, and denote it by N_ℓ. We introduce the following parameters:

$\boldsymbol{\gamma}$ number of distinct contexts present in X;

$\boldsymbol{\delta_i}$ number of occurrences of letter $\#_i$ in X, for $i \in [\gamma]$;

$\boldsymbol{\lambda}$ number of distinct critical length-k strings;

$\boldsymbol{\alpha^i_{\ell,j}}$ additional number of occurrences of N_ℓ introduced by replacing a $\#_i$ with a letter $j \in \Sigma$, for $\ell \in [\lambda]$;

$\boldsymbol{e_\ell}$ difference $(\tau - 1) - \mathrm{Freq}_X(N_\ell)$, for $\ell \in [\lambda]$.

Intuitively, $\boldsymbol{e_\ell}$ is the *budget* we have for N_ℓ: the number of its additional occurrences we can afford. Since replacing an occurrence of $\#_i$ by $j \in \Sigma$ adds k new strings in Σ^k, $\boldsymbol{\alpha^i_{\ell,j}}$ counts how many of them are equal to N_ℓ. Let $\boldsymbol{x_{i,j}}$ be the number of times we replace $\#_i$ by $j \in \Sigma$, and let $\mathcal{F} \subseteq [\gamma] \times \Sigma$ be the set of *forbidden* replacements: $(i,j) \in \mathcal{F}$ if and only if replacing $\#_i$ by j introduces a sensitive pattern. To determine whether there exists a way of replacing all #'s with letters without introducing any sensitive patterns nor τ-ghosts, we need to find a solution $x \in \mathbb{Z}^{\gamma \times |\Sigma|}$ to the following problem:

$$\begin{cases} x_{i,j} \geq 0 & \forall (i,j) \in [\gamma] \times \Sigma \\ x_{i,j} = 0 & \forall (i,j) \in \mathcal{F} \\ \sum_{i \in [\gamma], j \in \Sigma} \alpha^i_{\ell,j} x_{i,j} \leq e_\ell & \forall \ell \in [\lambda] \\ \sum_{j \in \Sigma} x_{i,j} = \delta_i & \forall i \in [\gamma] \end{cases} \tag{F.1}$$

The first and fourth constraints ensure that each # is replaced by exactly one letter, the second constraint that we do not reinstate any sensitive patterns, and the third constraint that we do not introduce any τ-ghosts. This is clearly an ILP with $m = \gamma|\Sigma|$ variables and at most $2m+\lambda+\gamma$ constraints. The well-known algorithm by Megiddo [271] solves the ILP problem in linear time in the number constraints (resp. variables) when the number of variables (resp. constraints) is upper bounded by a constant. Hence, although HMD is NP-complete in general, if appropriate subsets of parameters are bounded by a constant, we can count on polynomial-time solutions.

To show that HMD takes polynomial time in certain cases, let us start with a general preprocessing step. We construct a static dictionary with $\mathcal{O}(1)$ access time of the letters in X and the letters in strings of $\mathcal{S}$. The value (id) of each key (letter) is chosen from $\{1, \ldots, k|\mathcal{S}| + n\}$. This construction can be done in $\mathcal{O}(n + k|\mathcal{S}|)$ time using perfect hashing [154]. We can thus lexicographically sort all length-k substrings of X and all length-k strings in $\mathcal{S}$ (viewed as strings over letter id's) using radix sort in $\mathcal{O}(nk + |\mathcal{S}|k)$ time, and construct two dictionaries, one for X and one for $\mathcal{S}$, as follows. For X, we construct a trie of all its non-sensitive length-k substrings. The value of each key (non-sensitive pattern) is its multiplicity in X. We also construct a trie of all strings in $\mathcal{S}$ in a similar fashion (no multiplicities are relevant here, so no values are stored). We store in both tries, for every node, the first letter on each of its outgoing edges in a static dictionary with $\mathcal{O}(1)$ access time [154]. Thus both trie dictionaries support $\mathcal{O}(k)$ access time: if a length-k string Q is given as a query, we first convert it to a string $I(Q)$ of id's in $\mathcal{O}(k)$ time using the letter dictionary, and then search for $I(Q)$ from the root of the tries in $\mathcal{O}(k)$ time. The total construction time is $\mathcal{O}(nk + |\mathcal{S}|k)$.

When $\delta = \mathcal{O}(1)$, the brute-force algorithm checking all possible ways to replace the #'s with letters of Σ runs in polynomial time. There are $|\Sigma|^\delta$ ways to replace the #'s. Each of these ways generates δk new length-k strings for which we have to check if they are sensitive or create a τ-ghost. Checking if they are sensitive can be done using the trie of $\mathcal{S}$ in $\mathcal{O}(k)$ time per each length-k string. Counting the additional number of occurrences of each length-k substring of X can be done using the trie of X in $\mathcal{O}(k)$ time. Counting the number of occurrences of each length-k string that does not occur in X can be done by constructing a trie of all such strings (we have at most δk of them per way), similar to the preprocessing step. This gives $\mathcal{O}(nk + |\mathcal{S}|k + |\Sigma|^\delta \delta k^2)$ time in total.

A problem with parameters p and q is *fixed-parameter tractable* (FPT) in p if there exists a function f and a polynomial P such that the problem has time complexity $\mathcal{O}(f(p) \cdot P(q))$ [121]. The following theorem shows three scenarios where an FPT algorithm exists for HMD.

Theorem 50. HMD *is fixed-parameter tractable if*

(*a*) $|\Sigma| = \mathcal{O}(1)$ *and* $\gamma = \mathcal{O}(1)$*; or*

(*b*) $|\Sigma| = \mathcal{O}(1)$ *and* $k = \mathcal{O}(1)$*; or*

(*c*) $k = \mathcal{O}(1)$ *and* $\lambda = \mathcal{O}(1)$.

Proof. We first perform the above-mentioned preprocessing. (*a*) We will solve this case by constructing and solving the ILP in Eq. F.1. We can count the number of occurrences

of each length-k substring of X using the trie of X (and thus determine e_ℓ for these strings) in $\mathcal{O}(nk)$ time. The id i of each context $\#_i$ and its number δ_i of occurrences can be determined within the same complexity using a similar preprocessing: this is possible because the length of every context is $2k-2=\mathcal{O}(k)$. Finally, the $\alpha^i_{\ell,j}$'s and $\mathcal{F}$ can be computed in $\mathcal{O}(\gamma|\Sigma|k^2)$ total time as follows. For a context $\#_i$ and a letter $j\in\Sigma$, we create k new length-k strings when replacing $\#_i$ with j, each of which is either sensitive (in which event we add (i,j) to $\mathcal{F}$) or non-sensitive (we increase $\alpha^i_{\ell,j}$ by 1). Checking if they are sensitive can be done using the trie of $\mathcal{S}$ in $\mathcal{O}(k)$ time per length-k string. Counting the additional number of occurrences of a critical length-k substring of X can be done using the trie of X in $\mathcal{O}(k)$ time. Counting the number of occurrences of a critical length-k string that does not occur in X (note that $e_\ell=\tau-1$ for these strings) can be done by constructing a trie of all such strings, similar to the preprocessing step. The ILP is thus constructed in $\mathcal{O}(nk+|\mathcal{S}|k+\gamma|\Sigma|k^2)$ total time. Since the number of variables in the ILP is $m=\gamma|\Sigma|=\mathcal{O}(1)$ and solving ILP's is fixed-parameter linear in the number of variables [271], HMD is FPT if γ and $|\Sigma|$ are fixed.

(b) Since every context has length $2k-2$ and also $|\Sigma|=\mathcal{O}(1)$ and $k=\mathcal{O}(1)$, we have that $\gamma\leq|\Sigma|^{2k-2}=\mathcal{O}(1)$. Thus, if k and $|\Sigma|$ are fixed, we are in case (a), and HMD is FPT.

(c) If $k=\mathcal{O}(1)$ and $\lambda=\mathcal{O}(1)$, the numbers of constraints and variables in the ILP are not necessarily upper bounded by a constant. Therefore, we cannot directly solve the ILP in polynomial time. However, since the λ critical length-k strings contain overall at most λk different letters, we actually only need to distinguish among a bounded number of letters. Since we do not need to consider explicitly the remaining letters, we rather represent them by a single special letter. Let $\sigma\subseteq\Sigma$ denote the set of letters contained in critical length-k strings. Note that critical length-k strings can be determined as described in part (a). Thus σ can be specified and indexed using perfect hashing [154] within the same time complexity. We introduce a new letter \$ representing all the letters in $\Sigma\setminus\sigma$, and we denote by $\mathcal{F}|_\$$ the set of forbidden replacements where all pairs $(i,j)\in\mathcal{F}$ with $j\in\Sigma\setminus\sigma$ are collapsed in a single pair $(i,\$)$. We thus need to find a solution $x\in\mathbb{Z}^{\gamma\times(|\sigma|+1)}$ for:

$$\begin{cases} x_{i,j}\geq 0 & \forall i\in[\gamma], j\in\sigma\cup\{\$\} \\ x_{i,j}=0 & \forall (i,j)\in\mathcal{F}|_\$ \\ \sum_{i\in[\gamma],j\in\sigma}\ \alpha^i_{\ell,j}x_{i,j}\leq e_\ell & \forall \ell\in[\lambda] \\ \sum_{j\in\sigma\cup\{\$\}}\ x_{i,j}=\delta_i & \forall i\in[\gamma] \end{cases} \tag{F.2}$$

This new ILP can be constructed in $\mathcal{O}(nk+|\mathcal{S}|k+\gamma|\Sigma|k^2)$ time, like Eq. F.1. Since the ILP has only $\gamma(|\sigma|+1)=\mathcal{O}(1)$ variables, HMD is FPT for fixed k and λ [271]. We can obtain a solution to the original problem by replacing \$ by any letter in $\Sigma\setminus\sigma$ that does not create a sensitive pattern. □

We can decide in polynomial time if HM has a solution: we check all $|\Sigma|$ letter replacements at each of the δ positions where a # occurs. If, at each position, there exists at least one letter replacement that does not create a sensitive pattern, then HM has a solution. Thus, without loss of generality we assume that HM always has a solution. To minimize τ-ghosts in Z, we define a binary variable z_ℓ, $\ell\in[\lambda]$, which is

equal to 1 (resp. 0) when N_ℓ has (resp. has not) become τ-ghost. The ILP formulation for HM is to find $x \in \mathbb{Z}^{\gamma \times |\Sigma|}$ so as to:
Minimize $\sum_{\ell=1}^{\lambda} z_\ell$ subject to

$$\begin{cases} x_{i,j} \geq 0 & \forall (i,j) \in [\gamma] \times \Sigma \\ x_{i,j} = 0 & \forall (i,j) \in \mathcal{F} \\ z_\ell \geq 0 & \forall \ell \in [\lambda] \\ \sum_{i \in [\gamma], j \in \Sigma} \alpha^i_{\ell,j} x_{i,j} - k\delta z_\ell \leq e_\ell & \forall \ell \in [\lambda] \\ \sum_{j \in \Sigma} x_{i,j} = \delta_i & \forall i \in [\gamma] \end{cases} \tag{F.3}$$

Note that, in the ILP of Eq. F.3, $\sum_{i \in [\gamma], j \in \Sigma} \alpha^i_{\ell,j} x_{i,j} - k\delta z_\ell \leq e_\ell$ if and only if N_ℓ is not a τ-ghost or $z_\ell = 1$.

Theorem 51. HM *is fixed-parameter tractable if*

(*a*) $|\Sigma| = \mathcal{O}(1)$, $\gamma = \mathcal{O}(1)$, *and* $\lambda = \mathcal{O}(1)$; *or*

(*b*) $k = \mathcal{O}(1)$ *and* $\lambda = \mathcal{O}(1)$.

Proof. (*a*) We can obtain the ILP of Eq. F.3 in $\mathcal{O}(\lambda)$ time from the ILP of Eq. F.1, which can be constructed in $\mathcal{O}(nk + |\mathcal{S}|k + \gamma|\Sigma|k^2)$ time; see the proof of Theorem 50(a). The ILP of Eq. F.3 has at most $2m + 2\lambda + \gamma$ constraints and $m + \lambda = |\Sigma|\gamma + \lambda$ variables. Therefore HM is FPT if $|\Sigma|$, γ and λ are fixed [271].

(*b*) Similar to the ILP of Eq. F.2 (see Theorem 50(c)), we can reduce the alphabet Σ to the letters of the critical length-k strings and a special letter \$. This new minimization ILP has $\gamma(|\sigma| + 1) + \lambda \leq (k\lambda + 1)^{2k-1} + \lambda = \mathcal{O}(1)$ variables. Therefore HM is FPT if k and λ are fixed [271]. □

F.7 Greedy Heuristic for HM

Our heuristic aims to minimize τ-ghosts by controlling the number and frequency of length-k strings that may become τ-ghosts. It performs two left-to-right passes over X. In the first pass, it computes statistics on the number $\mathrm{Freq}_X(U)$ of occurrences of every length-k substring U of X. It also indexes all (at most $\delta|\Sigma|k$) non-sensitive patterns that may occur in Z but do not occur in X using a trie dictionary. In the second pass, it constructs Z by greedily replacing the occurrences of #, based on the statistics maintained on-line.

Consider the ith occurrence of # and let Z_i be the current version of Z, in which the first $i-1$ occurrences of # have already been replaced. For each letter $j \in \Sigma$, let S_j be the set of length-k substrings of string $U \cdot j \cdot V$ that do not contain any sensitive pattern (otherwise if *at least* one length-k substring of $U \cdot j \cdot V$ is sensitive $S_j = \emptyset$), where UV is the context of the ith occurrence of #. Further, let $S_j^{<\tau}$ be the subset of strings $Y \in S_j$ with $\mathrm{Freq}_{Z_\mathrm{i}}(Y) < \tau$; $S_j^{<\tau}$ is undefined if $S_j = \emptyset$.

The heuristic has the following three variants differing in how they choose j to replace the ith occurrence of #:

- *count*: it minimizes $|S_j^{<\tau}|$;

- *min-dist-sum*: it minimizes $\sum_{Y \in S_j^{<\tau}} [\tau - \mathrm{Freq}_{Z_i}(Y)]^{-1}$;
- *min-dist-max*: it minimizes $\max_{Y \in S_j^{<\tau}} [\tau - \mathrm{Freq}_{Z_i}(Y)]^{-1}$.

After replacing # by j, the statistics on Freq_{Z_i} are updated with the strings from S_j. If $i+1 \leq \delta$, the $(i+1)$th occurrence of # is replaced in Z_{i+1}. Otherwise, $Z = Z_i$ is returned.

Our heuristic takes $\mathcal{O}(nk + |\mathcal{S}|k + \delta|\Sigma|k^2)$ time. The first two terms of the sum come from a pre-processing analogous to that in Section F.6. The last one is because, for each of the δ replacements and for every letter in Σ, we process (and also index) k strings of length k to choose letter j.

F.8 Experiments

Experimental Setup and Datasets. The string sanitization method of [53] takes as input a string W over Σ, a positive integer k, and a set $\mathcal{S}$ of sensitive patterns, and then it performs the following three steps: (I) It constructs the shortest string X over $\Sigma_\#$ such that X contains no sensitive pattern and the order (and thus frequency) of all non-sensitive patterns in X and W is the same (see Section F.1). (II) It further tries to minimize the length of X by preserving the exact frequency of non-sensitive patterns but waiving their order property. The output of this step is a string Y over $\Sigma_\#$. (III) It replaces #'s in Y by the Multiple-Choice Knapsack based heuristic (see Section F.2). The output of this step is a string Z over Σ.

We compare the ILP formulation in Eq. F.3 (denoted by ILP) and the variant *min-dist-sum* of our heuristic (denoted by HEU) to Step (III) of [53] (denoted by TPM), both in terms of data utility and runtime. We omit the results for the other variants of our heuristic because HEU outperformed them.

The utility of a sanitized string Z is measured by:

1. The number of τ-ghosts in Z; i.e., the size of the set $\{U \in \Sigma^k : \mathrm{Freq}_X(U) < \tau \text{ and } \mathrm{Freq}_Z(U) \geq \tau\}$. All tested methods are guaranteed to create no τ-*lost*, i.e., the set $\{U \in \Sigma^k : \mathrm{Freq}_X(U) \geq \tau \text{ and } \mathrm{Freq}_Z(U) < \tau\}$ is empty. Clearly, zero τ-lost and τ-ghost patterns imply no utility loss for frequent length-k substring mining.
2. The *Distortion* measure [53], which is defined as $\sum_U (\mathrm{Freq}_W(U) - \mathrm{Freq}_Z(U))^2$, where $U \in \Sigma^k$ is a non-sensitive pattern. This measure penalizes changes in the frequency of non-sensitive patterns; low values imply high utility for frequency-based tasks [298].

We used publicly available datasets that were also used in the evaluation of [53]: Oldenburg (OLD) [1], Trucks (TRU) [2], MSNBC (MSN) [3], and the complete genome of *Escherichia coli* (DNA) [4]. We also used synthetic data (uniformly random strings, the largest of which is referred to as SYN). See Table F.1a for the characteristics of these datasets.

Dataset	length n	alphabet size $\lvert\Sigma\rvert$	no sens patterns $\lvert\mathcal{S}\rvert$	no sens positions $\lvert\mathcal{P}\rvert$	pattern length k	threshold τ
OLD	85,563	100	[60, 320]	[926, 5673]	[3, 6]	[3, 15]
TRU	5,763	100	[10, 70]	[363, 3813]	[2, 5]	[5, 30]
MSN	4,698,764	17	[60, 480]	[16792, 133590]	[3, 8]	[100, 300]
DNA	4,641,652	4	[30, 60]	[715, 1617]	[9, 15]	[5, 30]
SYN	20,000,000	10	[10, 1000]	[1967, 2001226]	[3, 6]	[5, 20]

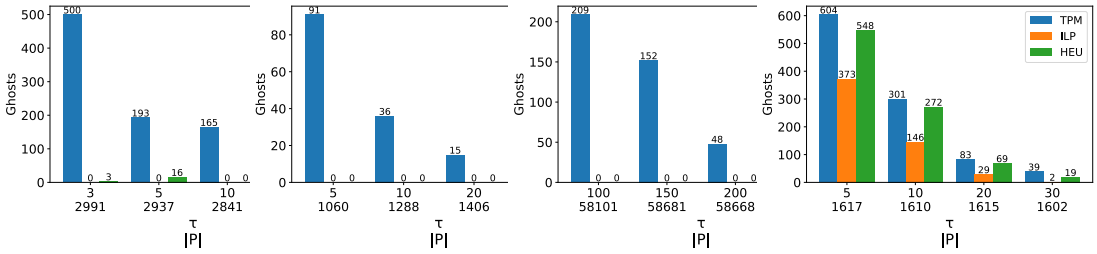
(a)

Dataset	no sens patterns $\lvert\mathcal{S}\rvert$	pattern length k	threshold τ
OLD	120	6	10
TRU	30	3	20
MSN	240	8	200
DNA	50	11	20
SYN	100	5	10

(b)

Table F.1: (a) Dataset characteristics. (b) Default values used.

The configuration of parameters was performed as in [53] (see Table F.1b for default values). That is, the sensitive patterns were selected randomly among the frequent length-k substrings of minimum support τ and the weight, costs, and θ parameters in TPMwere configured as in [53]. Our code was written in C++ and is available at https://anonymous.4open.science/r/d5b692e0-b6f4-432a-9961-a0928e8bc3fb/. The code of TPMwas also written in C++ and is available at [5]. We used the Gurobi solver v. 9.0.1 (single-thread configuration) to solve ILP instances. All experiments ran on an Intel i7-3770 CPU @ 3.40GHz with 16GB RAM, which indicates the low computational requirements of the methods.

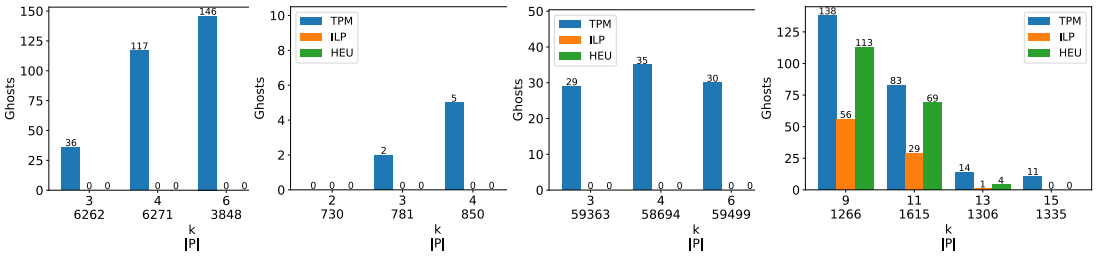


(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Figure F.1: Number of τ-ghosts for each dataset and varying τ (on the top of each bar, we show the number of τ-ghosts). ($|P|$ is the total number of positions where a sensitive pattern occurs in the input string.)

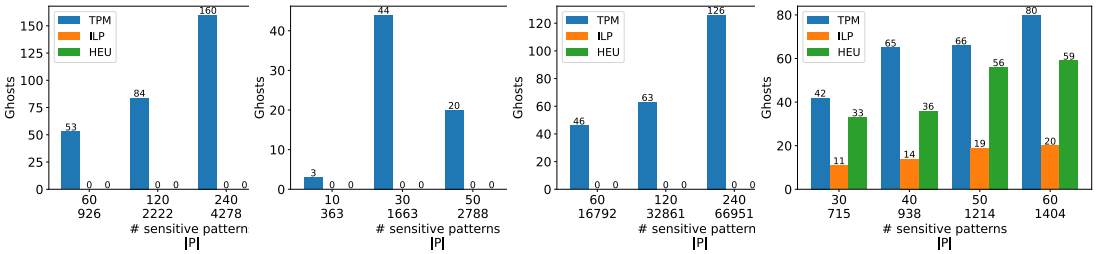
Data Utility. We show that our methods: (I) allow for frequent length-k pattern mining with no or insignificant utility loss (i.e., they create zero or few τ-ghosts), in contrast to TPM, and (II) incur substantially lower distortion than TPM.

Number of τ-ghosts. We examined the impact of τ, k, and $|\mathcal{S}|$ on τ-ghosts, in Figs. F.1, F.2, and F.3, respectively. As can be seen, ILP created no τ-ghosts, while TPM created tens or hundreds in all settings, for all datasets except DNA. In DNA, ILP outperformed TPM by 72% on average. HEU created no τ-ghosts in all datasets except OLD and DNA. Also, HEU outperformed TPM by 99% (respectively, 38%)



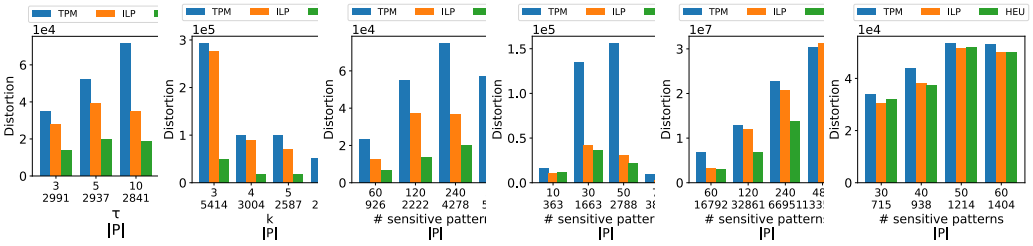
(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Figure F.2: Number of τ-ghosts for each dataset and varying pattern length k (on the top of each bar, we show the number of τ-ghosts). ($|P|$ is the total number of positions where a sensitive pattern occurs in the input string.)



(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Figure F.3: Number of τ-ghosts for each dataset and varying number of sensitive patterns $|\mathcal{S}|$ (on the top of each bar, we show the number of τ-ghosts). ($|P|$ is the total number of positions where a sensitive pattern occurs in the input string.)

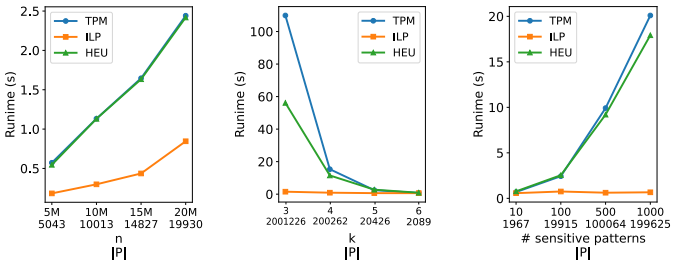


(a) OLD **(b)** OLD **(c)** OLD **(d)** TRU **(e)** MSN **(f)** DNA

Figure F.4: Distortion for varying: (a) τ, (b) k, and (c) number of sensitive patterns $|\mathcal{S}|$. Distortion for varying $|\mathcal{S}|$. ($|P|$ is the total number of positions where a sensitive pattern occurs in the input string.)

on average on OLD (respectively, DNA). DNA was challenging to sanitize with few τ-ghosts. This is because its small alphabet size makes it difficult to find letters that replace #'s without creating τ-ghosts. Our results show that both our methods allow for substantially more accurate frequent pattern mining than TPM and indicate that the method of replacing #'s used in TPM (see Section F.2) is ineffective to minimize τ-ghosts.

Distortion. We examined the impact of τ, k and $|\mathcal{S}|$ on Distortion, in Fig. F.4. Our



(a) Substr. of SYN **(b)** SYN **(c)** SYN

Figure F.5: Runtime on SYN for varying: (a) n, (b) k, and (c) number of sensitive patterns. ($|P|$ is the total number of positions where a sensitive pattern occurs in the input string.)

methods outperformed TPMsubstantially, which is very encouraging because TPM is specifically designed to minimize Distortion (by preserving the frequency of length-k non-sensitive patterns). A benefit of HEU is that it incurs lower distortion than ILP (25% on average).

Runtime. We examined the impact of input string length n, k, τ, and $|\mathcal{S}|$ on runtime. As can be seen in Fig. F.5a, our methods are practical, requiring less than 2.5 seconds to process a 20-million letter string. They also remained efficient when sanitizing patterns of different length (Fig. F.5b) or of minimum frequency threshold τ, as well as when sanitizing a large number of sensitive patterns (Fig. F.5c). Overall, ILP was the fastest and HEU took roughly the same time as TPM.

Bibliography

[1] https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm.

[2] https://bitbucket.org/stringsanitization/stringsanitizationpkdd19/src/master/truck_char.txt.

[3] http://archive.ics.uci.edu/ml/datasets/msnbc.com+anonymous+web+data.

[4] http://bacteria.ensembl.org/Escherichia_coli_str_k_12_substr_mg1655/.

[5] https://bitbucket.org/stringsanitization/stringsanitizationpkdd19/src/master/.

[6] A. A. Ragel and B. Crémilleux. Treatment of missing values for association rules. In *2nd PAKDD*, pages 258–270, 1998.

[7] A. Abboud, A. Backurs, and V. Williams. If the current clique algorithms are optimal, so is Valiant's parser. In *56th FOCS*, pages 98–117, 2015.

[8] A. Abboud, L. Georgiadis, G. F. Italiano, R. Krauthgamer, N. Parotsidis, O. Trabelsi, P. Uznanski, and D. Wolleb-Graf. Faster algorithms for all-pairs bounded min-cuts. In *46th ICALP*, pages 7:1–7:15, 2019.

[9] A. Abboud, R. Krauthgamer, and O. Trabelsi. New algorithms and lower bounds for all-pairs max-flow in undirected graphs. In *31st SODA*, pages 48–61, 2020.

[10] A. Abboud and V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th FOCS*, pages 434–443, 2014.

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

[12] O. Abul. Knowledge hiding in emerging application domains. In *Privacy-Aware Knowledge Discovery: Novel Applications and New Techniques*. CRC Press, 2010.

[13] O. Abul, F. Bonchi, and F. Giannotti. Hiding sequential and spatiotemporal patterns. *TKDE*, 22(12):1709–1723, 2010.

[14] M. Adamczyk, M. Alzamel, P. Charalampopoulos, C. S. Iliopoulos, and J. Radoszewski. Palindromic decompositions with gaps and errors. In *12th CSR*, pages 48–61, 2017.

[15] C. C. Aggarwal and P. S. Yu. On anonymization of string data. In *7th SDM*, pages 419–424, 2007.

[16] C. C. Aggarwal and P. S. Yu. A framework for condensation-based anonymization of string data. *DMKD*, 16(3):251–275, 2008.

[17] C. C. Aggarwal and P. S. Yu. *Privacy-Preserving Data Mining: Models and Algorithms*. Springer, 2008.

[18] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th ICDE*, pages 3–14, 1995.

[19] N. Aguse, Y. Qi, and M. El-Kebir. Summarizing the solution space in tumor phylogeny inference by multiple consensus trees. *Bioinformatics*, 35(14):i408–i416, 2019.

[20] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[21] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[22] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Comput.*, 10(3):405–421, 1981.

[23] A. Alatabbi, C. S. Iliopoulos, and M. S. Rahman. Maximal palindromic factorization. In *18th PSC*, pages 70–77, 2013.

[24] B. L. Allen and M. Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics*, 5(1):1–15, 2001.

[25] Y. Almirantis, P. Charalampopoulos, J. Gao, C. S. Iliopoulos, M. Mohamed, S. P. Pissis, and D. Polychronopoulos. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for molecular biology : AMB*, 12, 2017.

[26] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[27] M. Alzamel, L. Ayad, G. Bernardini, R. Grossi, C. Iliopoulos, N. Pisanti, S. Pissis, and G. Rosone. Degenerate string comparison and applications. In *18th WABI*, pages 21:1–21:14, 2018.

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

[29] M. Alzamel, J. Gao, C. S. Iliopoulos, C. Liu, and S. P. Pissis. Efficient computation of palindromes in sequences with uncertainties. In *18th EANN*, pages 620–629. 2017.

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

[31] K. Aoyama, Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda. Faster online elastic degenerate string matching. In *29th CPM*, pages 9:1–9:10, 2018.

[32] A. Apostolico, D. Breslauer, and Z. Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1):163–173, 1995.

[33] H. Arimura and T. Uno. An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. *Journal of Combinatorial Optimization*, 13(3):243–262, 2007.

[34] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradžev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11(5):1209–1210, 1970.

[35] M. Atzori, F. Bonchi, F. Giannotti, and D. Pedreschi. Anonymity preserving pattern discovery. *VLDB J.*, 17(4):703–727, 2008.

[36] L. A. K. Ayad, G. Badkobeh, G. Fici, A. Héliou, and S. P. Pissis. Constructing antidictionaries in output-sensitive space. In *29th DCC*, pages 538–547, 2019.

[37] F. Bacchus, A. J. Grove, J. Y. Halpern, and D. Koller. From statistical knowledge bases to degrees of belief. *Artificial Intelligence*, 87(1-2):75–143, 1996.

[38] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *47th STOC*, pages 51–58, 2015.

[39] A. Backurs and P. Indyk. Which regular expression patterns are hard to match? In *57th FOCS*, pages 457–466, 2016.

[40] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. *Inf. Process. Lett.*, 59(1):21–27, 1996.

[41] I. Banerjee, K. Li, M. Seneviratne, M. Ferrari, T. Seto, J. D. Brooks, D. L. Rubin, and T. Hernandez-Boussard. Weakly supervised natural language processing for assessing patient-centered outcome following prostate cancer treatment. *Journal of the American Medical Informatics Association Open*, 2(1):150–159, 2019.

[42] N. Bansal and R. Williams. Regularity lemmas and combinatorial algorithms. In *50th FOCS*, pages 745–754, 2009.

[43] C. Barton, A. Héliou, L. Mouchard, and S. P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinformatics*, 15:388, 2014.

[44] C. Barton, A. Héliou, L. Mouchard, and S. P. Pissis. Parallelising the computation of minimal absent words. In *11th PPAM. Revised Selected Papers, Part II*, pages 243–253, 2015.

[45] C. Barton, C. Liu, and S. P. Pissis. On-line pattern matching on uncertain sequences and applications. In *10th COCOA*, pages 547–562, 2016.

[46] A. Bashir, C. Ye, A. L. Price, and V. Bafna. Orthologous repeats and mammalian phylogenetic inference. *Genome Research*, 15(7):998–1006, 2005.

[47] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *4th LATIN*, pages 88–94, 2000.

[48] D. R. Bentley. Whole-genome re-sequencing. *Current Opinion in Genetics & Development*, 16(6):545–552, 2006.

[49] J. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[50] G. Bernardini, P. Bonizzoni, and P. Gawrychowski. Incomplete directed perfect phylogeny in linear time. *arXiv:2010.05644*, 2020.

[51] G. Bernardini, P. Bonizzoni, and P. Gawrychowski. On Two Measures of Distance Between Fully-Labelled Trees. In *31st CPM*, pages 6:1–6:16, 2020.

[52] G. Bernardini, P. Bonizzoni, G. D. Vedova, and M. Patterson. A rearrangement distance for fully-labelled trees. In *30th CPM*, pages 28:1–28:15, 2019.

[53] G. Bernardini, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, and G. Rosone. String sanitization: A combinatorial approach. In *ECML/PKDD*, pages 627–644, 2019.

[54] G. Bernardini, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Rosone, and M. Sweering. Combinatorial algorithms for string sanitization. *TKDD*, 2020.

[55] G. Bernardini, H. Chen, G. Fici, G. Loukides, and S. P. Pissis. Reverse-safe data structures for text indexing. In *22nd ALENEX*, pages 199–213, 2020.

[56] G. Bernardini, H. Chen, G. Loukides, N. Pisanti, S. P. Pissis, L. Stougie, and M. Sweering. String sanitization under edit distance. In *31st CPM*, pages 7:1–7:14, 2020.

[57] G. Bernardini, A. Conte, G. Gourdel, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Punzi, L. Stougie, and M. Sweering. Hide and mine in strings: Hardness and algorithms. In *20th ICDM*, 2020.

[58] G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone. Even Faster Elastic-Degenerate String Matching via Fast Matrix Multiplication. In *46th ICALP*, pages 21:1–21:15, 2019.

[59] G. Bernardini, N. Pisanti, S. P. Pissis, and G. Rosone. Pattern matching on elastic-degenerate text with errors. In *24th SPIRE*, pages 74–90, 2017.

[60] G. Bernardini, N. Pisanti, S. P. Pissis, and G. Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020.

[61] E. Bertino, G. Ghinita, and A. Kamra. *Access Control for Databases: Concepts and Systems*. Now Foundations and Trends, 2011.

[62] B. Bezawada, A. X. Liu, B. Jayaraman, A. L. Wang, and R. Li. Privacy preserving string matching for cloud computing. In *35th ICDCS*, pages 609–618, 2015.

[63] E. Bier, R. Chow, P. Golle, T. H. King, and J. Staddon. The rules of redaction: Identify, protect, review (and repeat). *IEEE Secur. Priv.*, 7(6):46–53, 2009.

[64] F. Biessmann, D. Salinas, S. Schelter, P. Schmidt, and D. Lange. "deep" learning for missing value imputationin tables with non-numerical data. In *27th CIKM*, pages 2017–2025, 2018.

[65] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.

[66] H. L. Bodlaender, M. R. Fellows, M. T. Hallett, H. T. Wareham, and T. J. Warnow. The hardness of perfect phylogeny, feasible register assignment and other problems on thin colored graphs. *Theoretical Computer Science*, 244(1-2):167–188, 2000.

[67] C. Böhm and F. Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowledge and Information Systems*, 6(6):728–749, 2004.

[68] F. Bonchi and E. Ferrari. *Privacy-Aware Knowledge Discovery: Novel Applications and New Techniques*. CRC Press, 2010.

[69] P. Bonizzoni, C. Braghin, R. Dondi, and G. Trucco. The binary perfect phylogeny with persistent characters. *Theoretical Computer Science*, 454:51–63, 2012.

[70] P. Bonizzoni, S. Ciccolella, G. Della Vedova, and M. Soto. Beyond perfect phylogeny: Multisample phylogeny reconstruction via ilp. In *8th ACM-BCB*, pages 1–10, 2017.

[71] P. Bonizzoni, S. Ciccolella, G. Della Vedova, and M. Soto. Does relaxing the infinite sites assumption give better tumor phylogenies? an ilp-based comparative approach. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 16(5):1410–1423, 2018.

[72] L. Bonomi, L. Fan, and H. Jin. An information-theoretic approach to individual sequential data sanitization. In *9th WSDM*, pages 337–346, 2016.

[73] L. Bonomi and L. Xiong. A two-phase algorithm for mining sequential patterns with differential privacy. In *22nd CIKM*, pages 269–278, 2013.

[74] M. Bordewich and C. Semple. On the computational complexity of the rooted subtree prune and regraft distance. *Annals of Combinatorics*, 8(4):409–423, 2005.

[75] K. Borozdin, D. Kosolobov, M. Rubinchik, and A. M. Shur. Palindromic Length in Linear Time. In *28th CPM*, pages 23:1–23:12, 2017.

[76] R. S. Boyer and J. S. Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[77] V. Brendel, J. S. Beckmann, and E. N. Trifonov. Linguistics of nucleotide sequences: Morphology and comparison of vocabularies. *Journal of Biomolecular Structure and Dynamics*, 4(1):11–21, 1986.

[78] K. Bringmann, F. Grandoni, B. Saha, and V. Williams. Truly sub-cubic algorithms for language edit distance and RNA-folding via fast bounded-difference min-plus product. In *56th FOCS*, pages 375–384, 2016.

[79] K. Bringmann, A. Grønlund, and K. Larsen. A dichotomy for regular expression membership testing. In *58th FOCS*, pages 307–318, 2017.

[80] K. Bringmann and M. Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th FOCS*, pages 79–97, 2015.

[81] G. S. Brodal, R. Fagerberg, T. Mailund, C. N. Pedersen, and A. Sand. Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. In *24th SODA*, pages 1814–1832, 2013.

[82] D. Brown, D. Smeets, B. Székely, D. Larsimont, A. M. Szász, P.-Y. Adnet, F. Rothé, G. Rouas, Z. I. Nagy, Z. Faragó, A.-M. Tokés, M. Dank, G. Szentmártoni, N. Udvarhelyi, G. Zoppoli, L. Pusztai, M. Piccart, J. Kulka, D. Lambrechts, C. Sotiriou, and C. Desmedt. Phylogenetic analysis of metastatic progression in breast cancer using somatic mutations and copy number aberrations. *Nature Communications*, 8:14944 EP –, 2017.

[83] D. Bryant. A classification of consensus methods for phylogenetics. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 61:163–184, 2003.

[84] J. R. Bunch and J. E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974.

[85] P. Buneman. The recovery of trees from measures of dissimilarity. *Mathematics in the Archaeological and Historical Sciences*, 1971.

[86] T. Calders, B. Goethals, and M. Mampaey. Mining itemsets in the presence of missing values. In *22nd SAC*, pages 404–408, 2007.

[87] J. H. Camin and R. R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, pages 311–326, 1965.

[88] A. Carpi and A. de Luca. Words and special factors. *Theoretical Computer Science*, 259(1-2):145–182, 2001.

[89] B. Cazaux, T. Lecroq, and E. Rivals. Linking indexing data structures to de bruijn graphs: Construction and update. *Journal of Computer and System Sciences*, 104:165–183, 2019.

[90] S. Chairungsee and M. Crochemore. Using minimal absent words to build phylogeny. *Theoret. Comput. Sci.*, 450:109–116, 2012.

[91] T. Chan. Speeding up the four Russians algorithm by about one more logarithmic factor. In *26th SODA*, pages 212–217, 2015.

[92] Y.-J. Chang. Hardness of RNA folding problem with four symbols. In *27th CPM*, pages 13:1–13:12, 2016.

[93] P. Charalampopoulos, M. Crochemore, G. Fici, R. Mercas, and S. P. Pissis. Alignment-free sequence comparison using absent words. *Inf. Comput.*, 262(Part):57–68, 2018.

[94] P. Charalampopoulos, M. Crochemore, and S. P. Pissis. On extended special factors of a word. In *25th SPIRE*, pages 131–138, 2018.

[95] P. Charalampopoulos, C. S. Iliopoulos, C. Liu, and S. P. Pissis. Property suffix array with applications in indexing weighted sequences. *ACM J. Exp. Algorithmics*, 25(1), 2020.

[96] K. Chatterjee, B. Choudhary, and A. Pavlogiannis. Optimal Dyck reachability for data-dependence and alias analysis. In *45th POPL*, pages 30:1–30:30, 2018.

[97] M. Chen, X. Yu, and Y. Liu. Mining moving patterns for predicting next location. 54(C):156–168, 2015.

[98] R. Chen, G. Acs, and C. Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *19th CCS*, pages 638–649, 2012.

[99] R. Chen, B. C. Fung, B. C. Desai, and N. M. Sossou. Differentially private transit data publication: A case study on the montreal transportation system. In *18th KDD*, pages 213–221, 2012.

[100] S. Ciccolella, G. Bernardini, L. Denti, P. Bonizzoni, M. Previtali, and G. Della Vedova. Triplet-based similarity score for fully multilabeled trees with poly-occurring labels. *Bioinformatics*, 2020.

[101] S. Ciccolella, M. S. Gomez, M. Patterson, G. Della Vedova, I. Hajirasouliha, and P. Bonizzoni. Gpps: an ilp-based approach for inferring cancer progression with mutation losses from single cell data. In *8th ICCABS*, 2018.

[102] S. Ciccolella, M. S. Gomez, M. Patterson, G. Della Vedova, I. Hajirasouliha, and P. Bonizzoni. Inferring cancer progression from single-cell sequencing while allowing mutation losses. *Bioinformatics*, 2018.

[103] A. Cislak and S. Grabowski. Sopang 2: online searching over a pan-genome without false positives. *CoRR*, abs/2004.03033, 2020.

[104] A. Cisłak, S. Grabowski, and J. Holub. SOPanG: online text searching over a pan-genome. *Bioinformatics*, 34(24):4290–4292, 2018.

[105] P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007.

[106] C. J. Colbourn, W. J. Myrvold, and E. Neufeld. Two algorithms for unranking arborescences. *Journal of Algorithms*, 20(2):268–281, 1996.

[107] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *34th STOC*, pages 592–601, 2002.

[108] T. C. P.-G. Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, pages 1–18, 2016.

[109] U. consortium et al. The uk10k project identifies rare variants in health and disease. *Nature*, 526(7571):82–90, 2015.

[110] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[111] G. Cormode, F. Korn, and S. Tirthapura. Exponentially decayed aggregates on data streams. In *25th ICDE*, pages 1379–1381, 2008.

[112] N. Cristianini and M. W. Hahn. *Introduction to computational genomics - a case studies approach*. Cambridge University Press, 2007.

[113] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[114] M. Crochemore, A. Héliou, G. Kucherov, L. Mouchard, S. P. Pissis, and Y. Ramusat. Absent words in a sliding window with applications. *Inf. Comput.*, 270, 2020.

[115] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen. Computing the longest previous factor. *Eur. J. Comb.*, 34(1):15–26, 2013.

[116] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Covering problems for partial words and for indeterminate strings. *Theor. Comput. Sci.*, 698:25–39, 2017.

[117] M. Crochemore, F. Mignosi, and A. Restivo. Automata and forbidden words. *Information Processing Letters*, 67:111–117, 1998.

[118] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *26th ICALP*, volume 1644, pages 261–270, 1999.

[119] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Data compression using antidictionaries. *Proceedings of the IEEE*, (11):1756–1768, 2000.

[120] M. Crochemore and D. Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991.

[121] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[122] A. Czumaj and A. Lingas. Finding a heaviest vertex-weighted triangle is not harder than matrix multiplication. *SIAM J. Comput.*, 39(2):431–444, 2009.

[123] B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp, and L. Zhang. On distances between phylogenetic trees. In *8th SODA*, pages 427–436, 1997.

[124] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[125] U. Department of Health & Human Services. Health Insurance Portablility and Accountability Act. https://aspe.hhs.gov/report/health-insurance-portability-and-accountability-act-1996, 1996.

[126] Z. DiNardo, K. Tomlinson, A. Ritz, and L. Oesper. Distance measures for tumor evolutionary trees. *Bioinformatics*, 2019.

[127] A. J. Dobson. Comparing the shapes of trees. In *Combinatorial Mathematics III*, pages 95–100. Springer, 1975.

[128] B. Dong, S. Xie, J. Gao, W. Fan, and P. S. Yu. Onlinecm: Real-time consensus classification with missing values. In *15th SDM*, pages 685–693, 2015.

[129] S. Dori and G. M. Landau. Construction of aho corasick automaton in linear time for integer alphabets. *Inf. Process. Lett.*, 98(2):66–72, 2006.

[130] J. Droppo and A. Acero. Context dependent phonetic string edit distance for automatic speech recognition. In *35th ICASSP*, pages 4358–4361, 2010.

[131] B. Dudek and P. Gawrychowski. Computing quartet distance is equivalent to counting 4-cycles. In *51st STOC*, pages 733–743, 2019.

[132] J. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[133] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *3rd TCC*, pages 265–284, 2006.

[134] P. Eirew, A. Steif, J. Khattra, G. Ha, D. Yap, H. Farahani, K. Gelmon, S. Chia, C. Mar, A. Wan, et al. Dynamics of genomic clones in breast cancer patient xenografts at single-cell resolution. *Nature*, 518(7539):422–426, 2015.

[135] M. El-Kebir. Sphyr: tumor phylogeny estimation from single-cell sequencing data under loss and error. *Bioinformatics*, 34(17):i671–i679, 2018.

[136] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[137] G. F. Estabrook, F. McMorris, and C. A. Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Systematic Zoology*, 34(2):193–200, 1985.

[138] S. Even and Y. Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981.

[139] M. Farach. Optimal suffix tree construction with large alphabets. In *38th FOCS*, pages 137–143, 1997.

[140] M. Farach-Colton and S. Muthukrishnan. Perfect hashing for strings: formalization and algorithms. In *7th CPM*, pages 130–140, 1996.

[141] J. Felsenstein and J. Felenstein. *Inferring phylogenies*, volume 2. Sinauer Associates Sunderland, MA, 2004.

[142] D. Fernández-Baca and L. Liu. Tree compatibility, incomplete directed perfect phylogeny, and dynamic graph connectivity: An experimental study. *Algorithms*, 12(3):53, 2019.

[143] G. Fici, T. Gagie, J. Kärkkäinen, and D. Kempa. A subquadratic algorithm for minimum palindromic factorization. *Journal of Discrete Algorithms*, 28:41–48, 2014.

[144] G. Fici and P. Gawrychowski. Minimal absent words in rooted and unrooted trees. In *26th SPIRE*, pages 152–161. Springer, 2019.

[145] G. Fici, F. Mignosi, A. Restivo, and M. Sciortino. Word assembly through minimal forbidden words. *Theoretical Computer Science*, 359(1-3):214–230, 2006.

[146] G. Fici, A. Restivo, and L. Rizzo. Minimal forbidden factors of circular words. *Theoret. Comput. Sci.*, 792:144–153, 2019.

[147] N. Fine and H. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

[148] C. Fiot, A. Laurent, and M. Teisseire. Approximate sequential patterns for incomplete sequence database mining. In *FUZZ*, pages 1–6, 2007.

[149] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[150] M. Fischer and A. Meyer. Boolean matrix multiplication and transitive closure. In *12th SWAT/FOCS*, pages 129–131, 1971.

[151] M. Fischer and M. Paterson. String matching and other products. In *7th SIAM-AMS Complexity of Computation*, pages 113–125, 1974.

[152] J. Flum. Rg downey and mr fellows. parameterized complexity. monographs in computer science. springer, new york, berlin, and heidelberg, 1999, xv+ 533 pp. *Bulletin of Symbolic Logic*, 8(4):528–529, 2002.

[153] S. Foresti. Microdata protection. In *Encyclopedia of Cryptography and Security, 2nd Ed*, pages 781–783. Springer, 2011.

[154] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $\mathcal{O}(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[155] Y. Fujishige, Y. Tsujimaru, S. Inenaga, H. Bannai, and M. Takeda. Computing dawgs and minimal absent words in linear time for integer alphabets. In *41st MFCS*, pages 38:1–38:14, 2016.

[156] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Comput. Surv.*, 42(4), 2010.

[157] M. Furman. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Mathematics Doklady*, 11(5):1252, 1970.

[158] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *22nd ISAAC*, pages 653–662, 2011.

[159] T. Gagie and S. J. Puglisi. Searching and indexing genomic databases via kernelization. *Frontiers in Bioengineering and Biotechnology*, 3:12, 2015.

[160] J. Gallant, D. Maier, and J. A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980.

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

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

[163] S. P. Garcia, O. J. Pinho, J. M. O. S. Rodrigues, C. A. C. Bastos, and P. J. S. G. Ferreira. Minimal absent words in prokaryotic and eukaryotic genomes. *PLoS ONE*, 6, 2011.

[164] M. R. Garey and D. S. Johnson. "Strong" NP-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, 1978.

[165] P. Gawrychowski, S. Ghazawi, and G. M. Landau. On indeterminate strings matching. In *31st CPM*, pages 23:1–23:12, 2020.

[166] P. Gawrychowski, G. M. Landau, W. Sung, and O. Weimann. A faster construction of greedy consensus trees. In *45th ICALP*, pages 63:1–63:14, 2018.

[167] P. Gawrychowski and P. Uznanski. Towards unified approximate pattern matching for Hamming and L_1 distance. In *45th ICALP*, pages 62:1–62:13, 2018.

[168] M. Gerlinger, S. Horswell, J. Larkin, A. J. Rowan, M. P. Salm, I. Varela, R. Fisher, N. McGranahan, N. Matthews, C. R. Santos, et al. Genomic architecture and evolution of clear cell renal cell carcinomas defined by multiregion sequencing. *Nature genetics*, 46(3):225, 2014.

[169] D. Gibb, B. Kapron, V. King, and N. Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *arXiv:1509.06464*, 2015.

[170] D. Gibney. An efficient elastic-degenerate text index? not likely. In *International Symposium on String Processing and Information Retrieval*, pages 76–88. Springer, 2020.

[171] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific Computing*, 9(5):862–874, 1988.

[172] A. Gkoulalas-Divanis and G. Loukides. Revisiting sequential pattern hiding to enhance utility. In *17th KDD*, pages 1316–1324, 2011.

[173] A. Gkoulalas-Divanis and V. S. Verykios. Exact knowledge hiding through database extension. *TKDE*, 21(5):699–713, 2009.

[174] K. Govek, C. Sikes, and L. Oesper. A consensus approach to infer tumor evolutionary histories. In *9th BCB*, pages 63–72, 2018.

[175] I. S. Gradshteyn and I. M. Ryzhik. *Table of integrals, series, and products*. Elsevier/Academic Press, Amsterdam, seventh edition, 2007.

[176] R. D. Gray, A. J. Drummond, and S. J. Greenhill. Language phylogenies reveal expansion pulses and pauses in pacific settlement. *Science*, 323(5913):479–483, 2009.

[177] R. Grossi, J. Iacono, G. Navarro, R. Raman, and S. R. Satti. Asymptotically optimal encodings of range data structures for selection and top-k queries. *ACM Transactions on Algorithms*, 13(2):28:1–28:31, 2017.

[178] R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani, and L. Versari. On-line pattern matching on similar texts. In *28th CPM*, pages 9:1–9:14, 2017.

[179] R. Grossi, C. S. Iliopoulos, R. Mercas, N. Pisanti, S. P. Pissis, A. Retha, and F. Vayani. Circular sequence comparison: algorithms and applications. *AMB*, 11:12, 2016.

[180] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[181] J. W. Grzymala-Busse and M. Hu. A comparison of several approaches to missing attribute values in data mining. In *3rd RSCTC*, pages 378–385, 2001.

[182] V. Guralnik and G. Karypis. A scalable algorithm for clustering sequential data. In *1st ICDM*, pages 179–186, 2001.

[183] D. Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21(1):19–28, 1991.

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

[185] D. Gusfield. Persistent phylogeny: a galled-tree and integer linear programming approach. In *6th ACM-BCB*, pages 443–451, 2015.

[186] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[187] R. Gwadera, A. Gkoulalas-Divanis, and G. Loukides. Permutation-based sequential pattern hiding. In *13th ICDM*, pages 241–250, 2013.

[188] I. Hajirasouliha, A. Mahmoody, and B. J. Raphael. A combinatorial approach for analyzing intra-tumor heterogeneity from high-throughput sequencing data. *Bioinformatics*, 30(12):i78–i86, 2014.

[189] I. Hajirasouliha and B. J. Raphael. *Reconstructing Mutational History in Multiply Sampled Tumors Using Perfect Phylogeny Mixtures*, pages 354–367. Lecture Notes in Computer Science. Springer Nature, 2014.

[190] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Record*, 29(2):1–12, May 2000.

[191] R. D. Heatherly, G. Loukides, J. C. Denny, J. L. Haines, D. M. Roden, and B. A. Malin. Enabling genomic-phenomic association discovery without sacrificing anonymity. *PLOS ONE*, 8:1–13, 02 2013.

[192] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th STOC*, pages 21–30, 2015.

[193] M. R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.

[194] M. Hoffmann, J. Iacono, P. K. Nicholson, and R. Raman. Encoding nearest larger values. *Theoretical Computer Science*, 710:97–115, 2018.

[195] J. Holm, K. De Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

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

[197] J. E. Hopcroft and R. M. Karp. An $n^{2.5}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[198] L. Huang, V. Popic, and S. Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):361–370, 2013.

[199] S.-E. Huang, D. Huang, T. Kopelowitz, and S. Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *28th SODA*, pages 510–520, 2017.

[200] K. T. Huber and V. Moulton. Phylogenetic networks from multi-labelled trees. *Journal of Mathematical Biology*, 52(5):613–632, 2006.

[201] E. Husić, X. Li, A. Hujdurović, M. Mehine, R. Rizzi, V. Mäkinen, M. Milanič, and A. I. Tomescu. Mipup: minimum perfect unmixed phylogenies for multi-sampled tumors via branchings and ilp. *Bioinformatics*, 35(5):769–777, 2019.

[202] D. H. Huson and D. Bryant. Application of phylogenetic networks in evolutionary studies. *Molecular Biology and Evolution*, 23(2):254–267, 2006.

[203] J. P. Hutchinson. On words with prescribed overlapping subsequences. *Utilitas Mathematica*, 7:241–250, 1975.

[204] T. I, S. Sugimoto, S. Inenaga, H. Bannai, and M. Takeda. Computing palindromic factorizations and palindromic covers on-line. In *25th CPM*, pages 150–161, 2014.

[205] C. S. Iliopoulos, R. Kundu, and S. P. Pissis. Efficient pattern matching in elastic-degenerate texts. In *11th LATA*, pages 131–142, 2017.

[206] C. S. Iliopoulos and J. Radoszewski. Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties. In *27th CPM*, pages 8:1–8:12, 2016.

[207] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Journal of Computer and Systems Sciences*, 62(2):367–375, 2001.

[208] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[209] P. Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *39th FOCS*, pages 166–173, 1998.

[210] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. In *9th STOC*, pages 1–10, 1977.

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

[212] K. Jahn, J. Kuipers, and N. Beerenwinkel. Tree inference for single-cell data. *Genome Biology*, 17:86, 2016.

[213] J. Jansson and R. Rajaby. A More Practical Algorithm for the Rooted Triplet Distance. *Journal of Computational Biology*, 2016.

[214] J. Jansson, R. Rajaby, C. Shen, and W.-K. Sung. Algorithms for the majority rule (+) consensus tree and the frequency difference consensus tree. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 15(1):15–26, 2016.

[215] J. Jansson, C. Shen, and W.-K. Sung. Improved algorithms for constructing consensus trees. *Journal of the ACM*, 63(3):1–24, 2016.

[216] H. Jiang, J. Ma, J. Luan, and D. Zhu. Approximation and nonapproximability for the one-sided scaffold filling problem. In *21st COCOON*, pages 251–263, 2015.

[217] W. Jiao, S. Vembu, A. G. Deshwar, L. Stein, and Q. Morris. Inferring clonal evolution of tumors from single nucleotide somatic mutations. *BMC bioinformatics*, 15(1):35, 2014.

[218] L. Jin, C. Li, and R. Vernica. Sepia: estimating selectivities of approximate string predicates in large databases. *The VLDB Journal*, 17(5):1213–1229, 2008.

[219] A. Kalai. Efficient pattern-matching with don't cares. In *13th SODA*, pages 655–656, 2002.

[220] M. Kao, T. W. Lam, W. Sung, and H. Ting. A decomposition theorem for maximum weight bipartite matchings. *SIAM J. Comput.*, 31(1):18–26, 2001.

[221] J. Karhumäki, S. Puzynina, M. Rao, and M. A. Whiteland. On cardinalities of k-abelian equivalence classes. *Theoretical Computer Sciene*, 658:190–204, 2017.

[222] J. Karhumäki, A. Saarela, and L. Q. Zamboni. On a generalization of abelian equivalence and complexity of infinite words. *Journal of Combinatorial Theory, Series A*, 120(8):2189–2206, 2013.

[223] J. Kärkkäinen, M. Piatkowski, and S. J. Puglisi. String Inference from Longest-Common-Prefix Array. In *44th ICALP*, pages 62:1–62:14, 2017.

[224] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[225] N. Karpov, S. Malikic, M. K. Rahman, and S. C. Sahinalp. A multi-labeled tree dissimilarity measure for comparing "clonal trees" of tumor progression. *Algorithms for Molecular Biology*, 14(1):17, 2019.

[226] U. Keich and P. A. Pevzner. Finding motifs in the twilight zone. *Bioinformatics*, 18(10):1374–1381, 2002.

[227] H. Kellerer, U. Pferschy, and D. Pisinger. *The Multiple-Choice Knapsack Problem*, pages 317–347. Springer Berlin Heidelberg, 2004.

[228] G. Kimmel and R. Shamir. The incomplete perfect phylogeny haplotype problem. *Journal of Bioinformatics and Computational Biology*, 3(02):359–384, 2005.

[229] C. Kingsford, M. C. Schatz, and M. Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11(1):21, 2010.

[230] B. Kirkpatrick and K. Stevens. Perfect phylogeny problems with missing values. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(5):928–941, 2014.

[231] D. Knuth, J. M. Jr., and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[232] D. E. Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

[233] D. C. Koboldt, K. M. Steinberg, D. E. Larson, R. K. Wilson, and E. R. Mardis. The next-generation sequencing revolution and its impact on genomics. *Cell*, 155(1):27–38, 2013.

[234] T. Kociumaka, S. P. Pissis, and J. Radoszewski. Pattern matching and consensus problems on weighted sequences and profiles. In *27th ISAAC*, pages 46:1–46:12, 2016.

[235] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń. Internal pattern matching queries in a text and applications. In *26th SODA*, pages 532–551, 2015.

[236] T. Kopelowitz and R. Krauthgamer. Color-distance oracles and snippets. In *27th CPM*, volume 54, pages 24:1–24:10, 2016.

[237] D. Kosolobov and N. Sivukhin. Compressed Multiple Pattern Matching. In *30th CPM*, pages 13:1–13:14, 2019.

[238] R. Krauthgamer and O. Trabelsi. Conditional lower bounds for all-pairs max-flow. *ACM Trans. Algorithms*, 14(4):42:1–42:15, 2018.

[239] J. Kuipers, K. Jahn, B. J. Raphael, and N. Beerenwinkel. Single-cell sequencing data reveal widespread recurrence and loss of mutational hits in the life histories of tumors. *Genome Research*, 2017.

[240] G. M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *18th STOC*, pages 220–230, 1986.

[241] K. Larsen, I. Munro, J. Nielsen, and S. Thankachan. On hardness of several string indexing problems. *Theor. Comput. Sci.*, 582:74–82, 2015.

[242] F. Le Gall. Powers of tensors and fast matrix multiplication. In *39th ISSAC*, pages 296–303, 2014.

[243] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.

[244] C.-A. Leimeister and B. Morgenstern. Kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014.

[245] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.

[246] H. Li and R. Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinform.*, 26(5):589–595, 2010.

[247] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou. Fuzzy keyword search over encrypted data in cloud computing. In *29th INFOCOM*, pages 1–5, 2010.

[248] R. Li, C. Yu, Y. Li, T. W. Lam, S. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[249] T. Li and N. Li. Injector: Mining background knowledge for data anonymization. In G. Alonso, J. A. Blakeley, and A. L. P. Chen, editors, *24th ICDE*, pages 446–455, 2008.

[250] P.-C. Lin, Y.-D. Lin, Y.-C. Lai, and T.-H. Lee. Using string matching for deep packet inspection. *IEEE Computer*, 41(4):23–28, 2008.

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

[252] R. J. Little and D. B. Rubin. *Statistical Analysis with Missing Data (3rd ed.)*. John Wiley & Sons, Inc., USA, 2019.

[253] A. Liu, K. Zhengy, L. Liz, G. Liu, L. Zhao, and X. Zhou. Efficient secure similarity computation on encrypted trajectory data. In *31st ICDE*, pages 66–77, 2015.

[254] Y. P. Liu and A. Sidford. Faster divergence maximization for faster maximum flow. *arXiv preprint arXiv:2003.08929*, 2020.

[255] G. Loukides, A. Gkoulalas-Divanis, and B. Malin. Anonymization of electronic medical records for validating genome-wide association studies. *Proceedings of the National Academy of Sciences USA*, 107(17):7898–7903, 2010.

[256] G. Loukides and R. Gwadera. Optimal event sequence sanitization. In *15th SDM*, pages 775–783, 2015.

[257] W. Lu, X. Du, M. Hadjieleftheriou, and B. C. Ooi. Efficiently supporting edit distance based string similarity search using b^+-trees. *TKDE*, 26(12):2983–2996, 2014.

[258] S. Maciuca, C. del Ojo Elias, G. McVean, and Z. Iqbal. A natural encoding of genetic variation in a burrows-wheeler transform to enable mapping and genome inference. In *16th WABI*, pages 222–233, 2016.

[259] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

[260] V. Mäkinen, B. Cazaux, M. Equi, T. Norri, and A. I. Tomescu. Linear time construction of indexable founder block graphs. In *20th WABI*, pages 7:1–7:18, 2020.

[261] S. Malikic, K. Jahn, J. Kuipers, S. C. Sahinalp, and N. Beerenwinkel. Integrative inference of subclonal tumour evolution from single-cell and bulk sequencing data. *Nature Communications*, 10(1):2750, 2019.

[262] S. Malikic, F. R. Mehrabadi, S. Ciccolella, M. K. Rahman, C. Ricketts, E. Haghshenas, D. Seidman, F. Hach, I. Hajirasouliha, and S. C. Sahinalp. Phiscs: a combinatorial approach for subperfect tumor phylogeny reconstruction via integrative use of single-cell and bulk sequencing data. *Genome Research*, 29(11):1860–1877, 2019.

[263] B. Malin and L. Sweeney. Determining the identifiability of DNA database entries. In *AMIA*, pages 537–541, 2000.

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

[265] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[266] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[267] L. Marsan and M. Sagot. Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. *J. Comput. Biol.*, 7(3-4):345–362, 2000.

[268] H. M. Martinez. An efficient method for finding repeats in molecular sequences. *Nucleic Acids Research*, 11(13):4629–4634, 1983.

[269] J. Matoušek. Computing dominances in E^n. *Inf. Process. Lett.*, 38(5):277–278, 1991.

[270] M. McVicar, B. Sach, C. Mesnage, J. Lijffijt, E. Spyropoulou, and T. De Bie. Sumoted: An intuitive edit distance between rooted unordered uniquely-labelled trees. *Pattern Recognition Letters*, 79:52–59, 2016.

[271] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM*, 31(1):114–127, 1984.

[272] T. Mieno, Y. Kuhara, T. Akagi, Y. Fujishige, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Minimal unique substrings and minimal absent words in a sliding window. In *46th SOFSEM*, pages 148–160, 2020.

[273] F. Mignosi, A. Restivo, and M. Sciortino. Words and forbidden factors. *Theor. Comput. Sci.*, 273(1-2):99–117, 2002.

[274] N. Mohammed, B. C. M. Fung, P. C. K. Hung, and C.-K. Lee. Centralized and distributed anonymization for high-dimensional healthcare data. *ACM Transactions on Knowledge Discovery from Data*, pages 18:1–18:33, 2010.

[275] A. Monreale, D. Pedreschi, R. G. Pensa, and F. Pinelli. Anonymity preserving sequential pattern mining. *Artif. Intell. Law*, 22(2):141–173, 2014.

[276] A. S. Morrissy, L. Garzia, and others. Divergent clonal selection dominates medulloblastoma at recurrence. *Nature*, 529:351 EP –, 01 2016.

[277] I. Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett.*, 1(2):56–58, 1971.

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

[279] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[280] J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(1):87–101, 2003.

[281] J. C. Na, H. Kim, S. Min, H. Park, T. Lecroq, M. Léonard, L. Mouchard, and K. Park. Fm-index of alignment with gaps. *Theor. Comput. Sci.*, 710:148–157, 2018.

[282] S. U. Nabar, K. Kenthapadi, N. Mishra, and R. Motwani. A survey of query auditing techniques for data privacy. In *Privacy-Preserving Data Mining: Models and Algorithms*, pages 415–431. Springer US, 2008.

[283] L. Nakhleh, T. Warnow, D. Ringe, and S. N. Evans. A comparison of phylogenetic reconstruction methods on an indo-european dataset. *Transactions of the Philological Society*, 103(2):171–192, 2005.

[284] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *S&P*, pages 111–125, 2008.

[285] J. Natwichai, X. Li, and M. Orlowska. Hiding classification rules for data sharing with privacy preservation. In *7th DaWaK*, pages 468–477, 2005.

[286] G. Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Softw., Pract. Exper.*, 31(13):1265–1312, 2001.

[287] G. Navarro. Indexing highly repetitive collections. In *23rd IWOCA*, pages 274–279, 2012.

[288] M. Nikaido, A. P. Rooney, and N. Okada. Phylogenetic relationships among cetartiodactyls based on insertions of short and long interpersed elements: hippopotamuses are the closest extant relatives of whales. *Proceedings of the National Academy of Sciences*, 96(18):10261–10266, 1999.

[289] P. C. Nowell. The clonal evolution of tumor cell populations. *Science*, 194(4260):23–28, 1976.

[290] T. Ota and H. Morita. On the adaptive antidictionary code using minimal forbidden words with constant lengths. In *ISITA 2010*, pages 72–77, 2010.

[291] E. Parliament. General Data Protection Regulation. http://data.consilium.europa.eu/doc/document/ST-9565-2015-INIT/en/pdf.

[292] M. Pawlik and N. Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems*, 40(1):1–40, 2015.

[293] I. Pe'er, T. Pupko, R. Shamir, and R. Sharan. Incomplete directed perfect phylogeny. *SIAM Journal on Computing*, 33(3):590–607, 2004.

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

[295] D. Pissinger. A minimal algorithm for the multiple-choice knapsack problem. *Eur J Oper Res*, 83(2):394–410, 1995.

[296] S. Pissis and A. Retha. Dictionary matching in elastic-degenerate texts with applications in searching VCF files on-line. In *17th SEA*, volume 103 of *LIPIcs*, pages 16:1–16:14, 2018.

[297] S. P. Pissis. MoTeX-II: structured MoTif eXtraction from large-scale datasets. *BMC Bioinformatics*, 15:235, 2014.

[298] S. P. Pissis. Motex-ii: structured motif extraction from large-scale datasets. *BMC Bioinform.*, 15:235, 2014.

[299] S. P. Pissis and A. Retha. Dictionary matching in elastic-degenerate texts with applications in searching VCF files on-line. In *17th SEA*, pages 16:1–16:14, 2018.

[300] V. Popic, R. Salari, I. Hajirasouliha, D. Kashef-Haghighi, R. B. West, and S. Batzoglou. Fast and scalable inference of multi-sample cancer lineages. *Genome biology*, 16(1):91, 2015.

[301] G. Poulis, S. Skiadopoulos, G. Loukides, and A. Gkoulalas-Divanis. Apriori-based algorithms for k^m-anonymizing trajectory data. *Transactions on Data Privacy*, 7(2):165–194, 2014.

[302] D. Pratas and J. M. Silva. Persistent minimal sequences of SARS-CoV-2. *Bioinformatics*, 07 2020. btaa686.

[303] H. Qin, H. Wang, X. Wei, L. Xue, and L. Wu. Privacy-preserving wildcards pattern matching protocol for iot applications. *IEEE Access*, 7:36094–36102, 2019.

[304] R. Rahn, D. Weese, and K. Reinert. Journaled string tree - a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics*, 30(24):3499–3505, 2014.

[305] R. Raman. Encoding data structures. In *9th WALCOM*, pages 1–7, 2015.

[306] S. Rangavittal, R. S. Harris, M. Cechova, M. Tomaszkiewicz, R. Chikhi, K. D. Makova, and P. Medvedev. RecoverY: k-mer-based read classification for Y-chromosome-specific sequencing and assembly. *Bioinformatics*, 34(7):1125–1131, 2017.

[307] M. Rautiainen, V. Makinen, and T. Marschall. Bit-parallel sequence-to-graph alignment. *Bioinformatics*, 2019.

[308] M. Régnier and M. Vandenbogaert. Comparison of statistical significance criteria. *J. Bioinformatics and Computational Biology*, 4(2):537–552, 2006.

[309] D. F. Robinson and L. R. Foulds. Comparison of weighted labelled trees. In *Combinatorial Mathematics VI*, pages 119–126. Springer, 1979.

[310] D. F. Robinson and L. R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131–147, 1981.

[311] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *12th ESA*, pages 580–591, 2004.

[312] M. Rubinchik and A. M. Shur. Eertree: An efficient data structure for processing palindromes in strings. In *27th IWOCA*, volume 9538 of *Springer LNCS*, pages 321–333. 2015.

[313] M. Ružić. Constructing efficient dictionaries in close to sorting time. In *35th ICALP*, volume 5125 of *Springer LNCS*, pages 84–95, 2008.

[314] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 07 1987.

[315] P. Samarati and L. Sweeney. Generalizing data to provide anonymity when disclosing information (abstract). In *17th PODS*, page 188, 1998.

[316] P. Samarati and L. Sweeney. Generalizing data to provide anonymity when disclosing information (abstract). In *17th ACM SIGACT-SIGMOD-SIGART*, page 188, 1998.

[317] G. Satas, S. Zaccaria, G. Mon, and B. J. Raphael. Scarlet: Single-cell tumor phylogeny inference with copy-number constrained mutation losses. *Cell Systems*, 10(4):323–332, 2020.

[318] R. V. Satya and A. Mukherjee. The undirected incomplete perfect phylogeny problem. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 5(4):618–629, 2008.

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

[320] C. Semple, M. Steel, et al. *Phylogenetics*, volume 24. Oxford University Press on Demand, 2003.

[321] J. Shang, J. Peng, and J. Han. MACFP: Maximal approximate consecutive frequent pattern mining under edit distance. In *16th SDM*, pages 558–566, 2016.

[322] W. Shen, J. Wang, and J. Han. *Sequential Pattern Mining*. Springer, 2014.

[323] S. T. Sherry, M.-H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin. dbsnp: the ncbi database of genetic variation. *Nucleic acids research*, 29(1):308–311, 2001.

[324] Y. Shiloach and U. Vishkin. An $o(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[325] R. M. Silva, D. Pratas, L. Castro, A. J. Pinho, and P. J. S. G. Ferreira. Three minimal sequences found in Ebola virus genomes and absent from human DNA. *Bioinformatics*, 31(15):2421–2425, 2015.

[326] P. Sinha and A. A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27(3):431–627, 1979.

[327] J. Sirén. Indexing variation graphs. In *19th ALENEX*, pages 13–27, 2017.

[328] D. Sleator and R. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[329] H. J. Smith, T. Dinev, and H. Xu. Information privacy research: An interdisciplinary review. *MIS Quarterly*, 35(4):989–1015, 2011.

[330] Software. Eigen library. http://eigen.tuxfamily.org, 2020.

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

[332] M. Spiliopoulou. Managing interesting rules in sequence mining. In *3rd PKDD*, pages 554–560, 1999.

[333] E. C. Stavropoulos, V. S. Verykios, and V. Kagklis. A transversal hypergraph approach for the frequent itemset hiding problem. *Knowl. Inf. Syst.*, 47(3):625–645, June 2016.

[334] M. Steel. *Phylogeny: discrete and random processes in evolution*. SIAM, 2016.

[335] K. Stevens and D. Gusfield. Reducing multi-state to binary perfect phylogeny with applications to missing, removable, inserted, and deleted data. In *10th WABI*, pages 274–287. Springer, 2010.

[336] P. H. Sudmant, T. Rausch, E. J. Gardner, R. E. Handsaker, A. Abyzov, J. Huddleston, Y. Zhang, K. Ye, G. Jun, M. H.-Y. Fritz, et al. An integrated map of structural variation in 2,504 human genomes. *Nature*, 526(7571):75–81, 2015.

[337] X. Sun and P. S. Yu. A border-based approach for hiding sensitive frequent itemsets. In *5th ICDM*, pages 426–433, 2005.

[338] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.

[339] A. Tamersoy, G. Loukides, M. E. Nergiz, Y. Saygin, and B. Malin. Anonymization of longitudinal electronic medical records. *IEEE Transactions on Information Technology in Biomedicine*, 16(3):413–423, 2012.

[340] M. Terrovitis and N. Mamoulis. Privacy preservation in the publication of trajectories. In *9th MDM*, pages 65–72, 2008.

[341] M. Terrovitis, N. Mamoulis, and P. Kalnis. Local and global recoding methods for anonymizing set-valued data. *The VLDB Journal*, 20(1):83–106, 2011.

[342] M. Terrovitis, G. Poulis, N. Mamoulis, and S. Skiadopoulos. Local suppression and splitting techniques for privacy preserving publication of trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 29(7):1466–1479, 2017.

[343] S. V. Thankachan, S. P. Chockalingam, Y. Liu, A. Krishnan, and S. Aluru. A greedy alignment-free distance estimator for phylogenetic inference. *BMC Bioinformatics*, 18(8):238:1–238:8, 2017.

[344] The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[345] The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018.

[346] G. Theodorakopoulos, R. Shokri, C. Troncoso, J. Hubaux, and J. L. Boudec. Prolonging the hide-and-seek game: Optimal trajectory privacy for location-based services. In *13th WPES*, pages 73–82, 2014.

[347] M. Thorup. Decremental dynamic connectivity. *Journal of Algorithms*, 33(2):229–243, 1999.

[348] J. Tuikkala, L. Elo, O. Nevalainen, and T. Aittokallio. Missing value imputation improves clustering and interpretation of gene expression microarray data. *BMC Bioinform.*, 9, 2008.

[349] I. Ulitsky, D. Burstein, T. Tuller, and B. Chor. The average common substring approach to phylogenomic reconstruction. *Journal of Computational Biology*, 13(2):336–350, 2006.

[350] L. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, Apr. 1975.

[351] V. S. Verykios, A. K. Elmagarmid, E. Bertino, Y. Saygin, and E. Dasseni. Association rule hiding. *TKDE*, 16(4):434–447, 2004.

[352] R. S. Walker, S. Wichmann, T. Mailund, and C. J. Atkisson. Cultural phylogenetics of the tupi language family in lowland south america. *PLOS One*, 7(4), 2012.

[353] S. Wandelt and U. Leser. String searching in referentially compressed genomes. In *4th KDIR*, pages 95–102, 2012.

[354] D. Wang, Y. He, E. Rundensteiner, and J. F. Naughton. Utility-maximizing event stream suppression. In *39th SIGMOD*, pages 589–600, 2013.

[355] J. Wang, E. Cazzato, E. Ladewig, V. Frattini, D. I. S. Rosenbloom, S. Zairis, F. Abate, Z. Liu, O. Elliott, Y.-J. Shin, J.-K. Lee, I.-H. Lee, W.-Y. Park, M. Eoli, A. J. Blumberg, A. Lasorella, D.-H. Nam, G. Finocchiaro, A. Iavarone, and R. Rabadan. Clonal evolution of glioblastoma under therapy. *Nature Genetics*, 48:768 EP –, 06 2016.

[356] J. Wang, N. Ntarmos, and P. Triantafillou. Indexing query graphs to speedup graph query processing. In *19th EDBT*, pages 41–52, 2016.

[357] P. Weiner. Linear pattern matching algorithms. In *14th SWAT/FOCS*, pages 1–11, 1973.

[358] Z. Wen, D. Deng, R. Zhang, and R. Kotagiri. 2ed: An efficient entity extraction algorithm using two-level edit-distance. In *35th ICDE*, pages 998–1009, 2019.

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

[360] V. Williams and R. Williams. Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications. In *38th STOC*, pages 225–231, 2006.

[361] V. Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *51st FOCS*, pages 645–654, 2010.

[362] V. V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *44th STOC*, pages 887–898, 2012.

[363] V. V. Williams. On some fine-grained questions in algorithms and complexity. In *International Congress of Mathematicians*, 2018.

[364] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *USENIX Technical Conference*, pages 153–162, 1992.

[365] Y. Wu, C. Chiang, and A. L. P. Chen. Hiding sensitive association rules with limited side effects. *TKDE*, 19(1):29–42, 2007.

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

[367] X. Xiao, Y. Tao, and N. Koudas. Transparent anonymization: Thwarting adversaries who know the algorithm. *ACM Transactions on Database Systems*, 35(2):8:1–8:48, 2010.

[368] S. Xu, X. Cheng, S. Su, K. Xiao, and L. Xiong. Differentially private frequent sequence mining. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2910–2926, 2016.

[369] Y. Xu, K. Wang, A. W. Fu, and P. S. Yu. Anonymizing transaction databases for publication. In *24th KDD*, pages 767–775, 2008.

[370] J. J. Ying, W. Lee, T. Weng, and V. S. Tseng. Semantic trajectory mining for location prediction. In *19th SIGSPATIAL*, pages 34–43, 2011.

[371] H. Yu. An improved combinatorial algorithm for boolean matrix multiplication. In *42nd ICALP*, volume 9134 of *Springer LNCS*, pages 1094–1105, 2015.

[372] H. Yu. An improved combinatorial algorithm for boolean matrix multiplication. *Inf. Comput.*, 261(Part):240–247, 2018.

[373] K. Yuan, T. Sakoparnig, F. Markowetz, and N. Beerenwinkel. Bitphylogeny: a probabilistic framework for reconstructing intra-tumor phylogenies. *Genome Biology*, 16(1):36, 2015.

[374] H. Zafar, A. Tzen, N. Navin, K. Chen, and L. Nakhleh. Sifit: inferring tumor trees from single-cell sequencing data under finite-sites models. *Genome Biology*, 18(1):178, 2017.

[375] M. J. Zaki, W. M. Jr, and W. Meira. *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press, 2014.

[376] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[377] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.