HABILITATION À DIRIGER DES RECHERCHES

DE L'UNIVERSITÉ PSL

Présentée à l'École normale supérieure

Algorithmes et structures de données d'approximation et probabilistes pour le traitement des chaînes de caractères

Approximation and Randomised String Processing

Présentation des travaux par

Tatiana **Starikovskaya**

Le 15 septembre 2025

Discipline

Informatique

Acknowledgments

I defended my Ph.D. at Lomonosov Moscow State University in 2013, and at that time including acknowledgements in one's thesis was not permitted there, and I would like to use this opportunity to thank many people I have met along the way. Some names appear more than once in this text, reflecting the fact that I have had the privilege of working with the same people in several different roles. If I have forgotten to mention someone who has supported me along the way, it is entirely unintentional and due only to my memory, not to lack of gratitude.

The journey to this thesis started more than thirty years ago when I was accepted to my school, and I am immensely grateful to its director, Evgenia Grigorievna Ermachkova. She gathered there some of the strongest teachers in math olympiads, and through them and my fellow students I discovered the joy of mathematics. I cannot do justice to all of my teachers by listing them here, but let me name at least a few: Nazar Agahanov, Vladimir Dolnikov, Pavel Kozhevnikov, Boris Trushin, Alexey Poyarkov — thank you!

I would also like to thank my scientific advisors during my studies and my postdoc years, I would not be where I am without them: Maxim Babenko, Raphaël Clifford, Gregory Kucherov, Frédéric Magniez, the late Mikhail Roytberg, whose guidance I will always treasure, Alexei Semenov.

I find myself to be incredibly lucky to be a part of the Combinatorial Pattern Matching community, who welcomed me when I was only starting my research career. I couldn't wish for more supportive, inspiring, and encouraging colleagues — thank you! It is through this community that I got acquainted with two exceptionally bright stringologists, Pawel Gawrychowski and Tomasz Kociumaka, who have become close friends and collaborators, have deeply influenced my work, and accompanied me on many exciting journeys (and I hope there are many more journeys ahead!). I am immensely grateful to them and to all the co-authors with whom I have had the joy to collaborate on the research contained in this document and beyond: Maxim A. Babenko, Gabriel Bathie, Raphaël Clifford, Vincent Cohen-Addad, Anne Driemel, Bartlomiej Dudek, Taha El Ghazi, Jonas Ellert, Laurent Feuilloley, Eldar Fischer, Allyx Fontaine, Dvir Fried, Moses Ganardi, Shay Golan, Garance Gourdel, Adam Górkiewicz, Allan Grønlund Jørgensen, Danny Hucke, Ignat I. Kolesnichenko, Roman Kolpakov, Tsvi Kopelowitz, Tomasz Kociumaka, Gregory Kucherov, Gad M. Landau, Kasper Green Larsen, Markus Lohrey, Konstantinos Mamouras, Frédéric Magniez, Kotaro Matsuda, Masakazu Tateshita, Yakov Nekrich, Ely Porat, Jakub

Radoszewski, Benjamin Sach, Anna Selugina, Teresa Anna Steiner, Michal Svagerka, Przemyslaw Uznanski, Hjalte Wedel Vildhøj.

I would also like to thank all my colleagues with whom I have exchanged ideas but who are not yet my co-authors, and those who have provided crucial support and advice at various stages of my career. Even though they may not appear in my bibliography or curriculum vitae, their words of support and advice have often been as important to my development as new scientific insights.

In addition to the broader international community, I am deeply grateful to DI-ENS for providing me with such an exceptional working environment. I am thankful to all my colleagues there who have contributed to make it a pleasant and inspiring place to work. Teaching at DIENS has also been a delightful experience. I would like to thank all the students who have made my classes lively and interesting, as well as my colleagues, in particular, Pierre Aboulker, Gabriel Bathie, Anne Bouillard, Taha El Ghazi, Jérôme Feret, Chien-Chung Huang, Paul Jeanmaire, Ky Nguyen, Marc Pouzet, Pierre Senellart. I've learned a lot by working by your side, thank you! My special thanks go to Lise-Marie Bivard, who has provided crucial support for many administrative tasks since my arrival at ENS, helping me free up time to do research.

Should this text serve its intended purpose, it will authorize me to direct research. Because this path required me to gain co-supervision experience beforehand, I warmly thank Nathanaël Fijakow, Chien-Chung Huang, and Pierre Peterlongo, who agreed to co-advise PhD students with me, thereby giving me the opportunity to gain valuable experience. In addition to those already mentioned above, I would like to thank all the students, the PhDs and the postdocs who trusted me to accompany them in their research journey, even before my habilitation: Gabriel Bathie, Taha El Ghazi, Jonas Ellert, Garance Gourdel, Benoît Huftier, Wojciech Janczewski, Alexander Loptev, Adrien Mathieu, Anna Selugina.

I gratefully acknowledge the financial support received from the Agence Nationale de Recherche projects ANR-19-CE48-0016 and ANR-20-CE48-0001.

I thank the reviewers of this manuscript Antoine Amarilli, Nadia Pisanti, and Mikaël Salson for accepting to perform this task. I would also like to thank Guillaume Fertin, Chien-Chung Huang, Alantha Newman and Luc Segoufin for accepting to be members of the jury of this manuscript's defence.

Last, but not least, I would like to thank my family and in particular my beloved husband, who has always been supportive of my non-trivial choice of a career, and my daughter, who, at her three years, has already "reviewed" so many papers for me :-)

Contents

1 Research trajectory

In March 2013, I obtained a PhD degree at Lomonosov Moscow State University (Moscow, Russia), for a thesis titled “Effective Algorithms for Specific Word Processing Problems”. My thesis was co-supervised by Alexei Semenov (Lomonosov Moscow State University) and Gregory Kucherov (CNRS and University Paris-Est Marne-la-Vallée). After that, I worked as an Assistant Professor at the Computer Science department of the Higher School of Economics in Moscow, Russia. In January 2015, I moved to Bristol University where I worked as a Research Associate in the group of Raphaël Clifford. In September 2016, I moved to IRIF, where I worked as a postdoc with Frédéric Magniez until, in September 2017, I joined École Normale Supérieure as a Maître de conférences, where I've been working up to date.

My research interests have gradually evolved over time. My initial exposure to stringology came through a lecture on Kolmogorov complexity by A. B. Sosinsky at the summer camp for high school students “Modern Mathematics” in Dubna, Russia. In my third year at Lomonosov State University, I chose Alexey Semenov as my scientific advisor, attracted by his work on related topics. However, I soon realised that I am more interested in the algorithmic aspects of stringology. To better align with these interests, Alexey Semenov introduced me to Andrey Muchnik and Mikhail Roytberg, who proposed an algorithmic problem in RNA folding: improving the classical cubic-time dynamic programming algorithm [119]. Although I achieved no improvement at the time (which is probably no wonder as the first subcubic-time algorithm appeared only in FOCS 2017 [127]), this collaboration introduced me to bioinformatics.

Concurrently, I attended Maxim Babenko's course on string algorithms, which led to a long-term collaboration and my entry into the broader field of algorithms on strings. During my PhD, through Mikhail Roytberg's collaboration with the CNRS Interdisciplinary Scientific Center Poncelet, I met Gregory Kucherov, who became my co-advisor and facilitated a research stay at the Laboratoire d'Informatique Gaspard-Monge.

During the time I worked as an Assistant Professor at the Computer Science department of the Higher School of Economics in Moscow, Russia, I continued working on algorithm in strings. My later postdoctoral work with Raphaël Clifford at the University of Bristol introduced me to streaming algorithms, while discussions with Frédéric Magniez sparked my interest in property testing. Today, my principal research area is algorithms and data structures for efficient processing of strings and, more generally, formal languages, with a focus on small-space algorithms, streaming algorithms, property testing, and lower bounds.

Doctoral and scientific supervision During the course of my career, I have been honoured to (co)-supervise multiple undergraduate-level research internships, as well as three PhD students, Garance Gourdel (defended in October 2023), Gabriel Bathie (defended in June 2025), and Taha El Ghazi (Expected defence date February 2027).

Invited talks It was my greatest pleasure to give invited talks at Jewels of Automata Theory 2024, “Randomness, Information, and Complexity” week 2024, CPM 2023, 18th Montoise Conference on Theoretical Computer Science, Highlights 2022, DAAL Days 2022, ISAAC 2021, IWOCA 2021.

Community service Previously, I have served on the program committees of: SODA 2026, MFCS 2026, STACS 2025, SeqBIM 2024, SPIRE 2024, IWOCA 2024, WADS 2023, SeqBIM 2023, ICALP 2020, SOSA 2020, CPM 2019, CPM 2018, CPM 2017, SPIRE 2017, IWOCA 2016, CSR 2016, SPIRE 2015, CPM 2015, CPM 2014. Together with P. Gawrychowski, I co-chaired CPM 2021.

2 Introduction

"All we have to decide is what to do with the time that is given us."
J.R.R. Tolkien, *The Fellowship of the Ring*

This habilitation thesis focuses on algorithms and data structures for solving the fundamental problem of pattern matching in massive and noisy string data, as well as their applications.

The history of string algorithms originates from the pattern matching problem. In its most basic form, the pattern matching problem assumes that a short string, called the *pattern*, and a longer string, called the *text*, are given, and the goal is to find all substrings of the text that are equal to the pattern. A common application is keyword search in a PDF document. Efficient algorithms and data structures are essential for such routine tasks, as users expect fast responses and minimal resource usage for basic operations. In the seminal 1977 result [97], Knuth, Morris, and Pratt showed the first linear-time and linear-space algorithm for pattern matching. The asymptotic time and space complexities of this algorithm are optimal, since the input must be read and stored. Since then, over 80 pattern matching algorithms have been developed [63, 61, 62], aiming to optimize constants and to address diverse computational settings.

The algorithms discussed above assume that the text cannot be preprocessed, that is, the pattern and the text arrive simultaneously. A complementary formulation assumes that the text arrives first and is preprocessed into a data structure called a *text index*, which supports queries of the following form: given a pattern, find all substrings of the text equal to the pattern. This approach is particularly useful when multiple patterns need to be searched within a fixed text. The most well-known text index is the suffix tree, introduced by Weiner [137]. It requires space linear in the length of the text and supports queries in time linear in the length of the pattern. The suffix tree represents all *suffixes* of the text, i.e. the substrings of the text that start at an arbitrary position and extend till the end of the text, in a tree-like structure. The suffixes are grouped together by their common prefixes, enabling efficient pattern searches.

2.1 Modern-day challenges

Pattern matching has traditionally been applied in domains where data can be represented as a text, such as bioinformatics, information retrieval (e.g., in text and music analysis), and digital security (e.g., in intrusion detection), among others. However, modern data generation in these domains introduces new challenges, such as handling input that is massive, fragmented, noisy, or constantly evolving. Nevertheless, algorithms are still expected to function reliably and produce meaningful results.

2.1.1 Massive string data

One of the primary challenges posed by modern string data is its volume. A particularly voluminous type of data is versioned or archival content. Several illustrative examples follow. In 2022, the metadata from the revision history (excluding article content) of the English Wikipedia pages alone occupied 75 gigabytes. In 2023, the Software Heritage project [3], which aims to archive all software source code ever produced, reported a storage volume of nearly 1 petabyte [140]. Similarly, bioinformatics maintains large-scale archives of string data. As of today, the European Nucleotide Archive (ENA) has accumulated over 50 petabytes [1] of sequencing data. The NCBI Sequence Read Archive (SRA) currently holds over 73 petabases [2] of data, including 38 petabases publicly available, and continues to grow (see Figure 2.1).

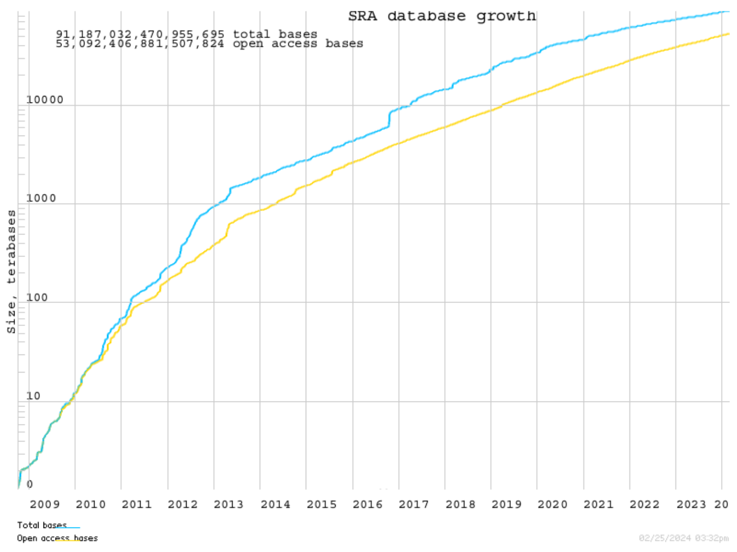


Figure 2.1: Size of the Sequence Read Archive [2].

Classical algorithms and data structures, which assume that data is fully stored in an uncompressed form, become infeasible at such scales. Nevertheless, applications still require effective methods for extracting meaningful information from such data. Two common approaches to storing and processing large-scale data are distributed computing and cloud platforms, but these alone may be insufficient and must be complemented by novel algorithmic and software tools [111]. This is due to two main reasons. First, even the most powerful systems are constrained by the slowing progress of semiconductor technology and the breakdown of Moore's law, making it difficult to keep pace with the exponential growth of data volumes. Second, in the widely adopted "platform as a service" model, users are charged based on resource consumption, which can become prohibitively expensive in the absence of efficient algorithms. Moreover, in scenarios such as disease outbreaks in remote areas (e.g., the 2014 Ebola virus epidemic in West Africa), access to cloud resources may be unavailable, requiring on-site data analysis under severe computational constraints.

This thesis focuses on the particularly restrictive *streaming model*. In this model, data items arrive sequentially, and previously seen items cannot be accessed unless explicitly stored. All memory usage, including any space allocated to store information about the input, contributes to the total space complexity. The overarching goal is to minimize space usage (ideally to a polylogarithmic function of the input size) while processing each item on-the-fly, resulting in highly efficient algorithms.

The study of small-space streaming algorithms for pattern matching began in earnest with the work presented at FOCS 2009 [120]. That year, Porat and Porat introduced a randomized algorithm for exact pattern matching in a stream, using only $O(\log m)$ space and requiring $O(\log m)$ worst-case time per arriving symbol [120]. This result was subsequently simplified [60] and later improved in 2011 to achieve constant time per arriving symbol [30]. Since then, streaming pattern matching has received significant attention in the literature. A detailed review is provided in Section 2.2.1.

Sometimes, designing an efficient streaming algorithm for certain pattern matching problems might be computationally intractable. In this case, a less restrictive yet still space-conscious alternative is the read-only model. In this setting, the algorithm is allowed constant-time random access to the input, and the space complexity is measured as the auxiliary memory beyond input storage. A foundational constant-space approach to exact pattern matching was introduced by Karp and Rabin [94], which appeared more than thirty years before the algorithm of Porat and Porat. The algorithm of Karp and Rabin is randomised and uses a so-called fingerprint, a hash function mapping strings to elements of a finite field, which, as we will see later in Section 2.2.1, had a huge impact in streaming algorithms. Following this work, several deterministic constant-space read-only algorithms for exact pattern matching have been developed [32, 29, 55, 56, 57, 58, 79, 80, 126]. All read-only algorithms mentioned above use time linear in the total length of the text and the pattern.

2.1.2 Noisy or scattered data

To remain affordable in terms of time and cost, modern large-scale data generation methods often tolerate noise and fragmentation, posing additional challenges for string processing. In pattern matching, a natural approach to handling noisy or fragmented data is to search for substrings similar (rather than identical) to the pattern. This problem has long been studied in stringology, albeit with a different motivation: Even when the data is carefully collected and relatively noise-free, efficiently identifying approximate matches continues to pose a challenge. For example, given a gene, one may wish to identify whether a variant appears in a given genome. This variant of pattern matching is referred to as *approximate pattern matching*. The two most widely used similarity measures on strings are the Hamming and the edit distances. Recall that the *Hamming distance* between two equal-length strings is the number of mismatching pairs of characters of the strings. The *edit distance* of two strings, not necessarily of equal lengths, is the smallest number of edits (character insertions, deletions, and substitutions) needed to transform one string into the other. Below, we give a brief survey of previous work on approximate pattern matching. Let n and m denote the lengths of the text and pattern, respectively.

Algorithms for the Hamming distance In general, computing Hamming distance is easier and is often regarded as a preliminary step toward solving the edit distance problem. The first solution for approximate pattern matching under the Hamming distance was given by Abrahamson [6] and, independently, Kosaraju [99]; based on the fast Fourier transform, it requires $\mathcal{O}(n\sqrt{m \log m})$ time to compute the Hamming distance between the pattern and all the length-m substrings of the text. Up to date, no algorithm has improved upon this time complexity for the general version of approximate pattern matching under Hamming distance, although more efficient solutions exist when the interest is restricted to distances not exceeding a given threshold k, a variant known as the *k-mismatch pattern matching*. The first algorithm for the k-mismatch problem was given by Landau and Vishkin [102], who improved the running time to $\mathcal{O}(kn)$ using the technique known as “kangaroo jumps”, which uses the suffix tree to compute the longest common prefix of two suffixes in constant time. This bound was further improved by Amir et al. [12] who showed two algorithms: One with running time $\mathcal{O}(n\sqrt{k \log k})$, and another with running time $\tilde{\mathcal{O}}(n + k^3 n/m)$[1]. Continuing this line of research, Clifford et al. [46] presented an $\tilde{\mathcal{O}}(n + k^2 n/m)$-time algorithm, while Gawrychowski and Uznański [78] proposed a smooth trade-off between the latter and the solution of Amir et al. by designing an $\tilde{\mathcal{O}}(n + kn/\sqrt{m})$-time algorithm. Very recently, Chan et al. [39] shaved off most of the polylogarithmic factors and achieved the running time of $\mathcal{O}(n + \min(k^2 n/m, kn\sqrt{\log m}/\sqrt{m}))$ at the cost of Monte-Carlo randomization.

Algorithms for the edit distance For the edit distance, a comprehensive survey of prior work can be found in [115], and here we only discuss the most significant

[1]Hereafter, $\tilde{\mathcal{O}}(\cdot)$ hides a factor of $\text{poly}(\log n)$.

theoretical results. For the general variant of the problem, Sellers introduced the first algorithm [131]. The algorithm was based on dynamic programming and used $\mathcal{O}(nm)$ time. Masek and Paterson [109] improved the running time of the algorithm to $\mathcal{O}(nm/\log n)$ via the Four Russians technique. In terms of lower bounds, it is known that there is no solution with strongly subquadratic time complexity unless the Strong Exponential Time hypothesis [88] is false, even over a binary alphabet [15, 35]. Abboud et al. [5] established a sharper bound under a weaker complexity assumption: Namely, they showed that even shaving an arbitrarily large polylog factor would imply that NEXP does not have non-uniform NC^1 circuits. Finally, Clifford et al. [47] showed that, in the cell-probe model with unit word size $w = 1$, any randomised algorithm that computes the edit distances between the pattern and the text online must spend $\Omega(\frac{\sqrt{\log n}}{(\log \log n)^{3/2}})$ expected amortised time per character of the text.

Similarly to the Hamming distance, one can define the threshold variant of approximate pattern matching, which we refer to as *the k-edit pattern matching*. The first algorithm for this variant of the problem was developed by Landau and Vishkin [103]; this by-now classical algorithm solves the problem in $\mathcal{O}(nk)$ time. The current best result was achieved by a series of works [128, 53] with the running time (for certain parameter ranges) $\mathcal{O}(n + k^4 n/m)$. Very recently, Charalampopoulos et al. [40, 41] studied the problem for both distances in the grammar-compressed setting. Their result, in particular, implies an $\tilde{\mathcal{O}}(n + k^{3.5} n/m)$-time algorithm for approximate pattern matching with k edits.

Approximate text indexing In the approximate text indexing problem, the goal is to preprocess the text into a data structure that supports efficient search for substrings that are similar to a given pattern. As in the pattern matching setting, we fix a similarity threshold k. In the *text indexing with k mismatches* problem, the objective is to report all substrings of the text whose Hamming distance from the pattern is at most k. In the *text indexing with k edits* problem, the goal is to find all substrings of the text at edit distance at most k from the pattern.

The importance of approximate text indexing is witnessed by the extensive use of the program BLAST that serves exactly this purpose for biologists (the original paper on BLAST [11] has been cited 50,000+ times, see also [114]). However, BLAST does not provide theoretical guarantees, and its success is probably due to its efficiency in practice and the simplicity of underlying ideas, which has facilitated its adaptation to a wide range of applications.

The quest for data structures with provable theoretical guarantees for approximate text indexing has led to the development of several efficient solutions. A particularly remarkable result was established by Cole, Gottlieb, and Lewenstein in their foundational work [52]. By combining in a very clever way binary search trees and suffix trees, they proposed a data structure for text indexing with k mismatches with space $\mathcal{O}(n \log^k n)$ and query time $\mathcal{O}(m + \frac{1}{k!}(c \log n)^k \log \log n + \mathsf{occ})$, where $c > 1$ is a constant. In the same work, they also extended their approach to

Space	Query time	
$\mathcal{O}(n(\alpha \log \alpha \log n)^k)$ words	$\mathcal{O}(d + (\log_\alpha n)^k \log\log n + \mathsf{occ})$	[134]
$\mathcal{O}(n \log^k n)$ words	$\mathcal{O}(d + \frac{1}{k!}(c \log n)^k \log\log n + \mathsf{occ})$	[52]
$\mathcal{O}(n \log^{k-1} n)$ words	$\mathcal{O}(d + \frac{1}{k!}(c \log n)^k \log\log n + \mathsf{occ})$	[38]
$\mathcal{O}(n)$ words	$\mathcal{O}(d^2 \min\{n, \lvert\Sigma\rvert^k d^{k+1}\} + \mathsf{occ})$	[136]
	$\mathcal{O}(d \min\{n, \lvert\Sigma\rvert^k d^{k+1}\} \log \min\{n, \lvert\Sigma\rvert^k d^{k+1}\} + occ)$	[136]
	$\mathcal{O}(\min\{n, \lvert\Sigma\rvert^k d^{k+2}\} + \mathsf{occ})$	[50]
	$\mathcal{O}(\min\{(\lvert\Sigma\rvert d)^k \log n + \mathsf{occ})$	[87]
	$\mathcal{O}(d + (c \log n)^{k(k+1)} \log\log n + \mathsf{occ})$	[38]
	$\mathcal{O}(\lvert\Sigma\rvert^k d^{k-1} \log n \log\log n + \mathsf{occ})$	[37]
$\mathcal{O}(n\sqrt{\log n})$ bits	$\mathcal{O}((\lvert\Sigma\rvert d)^k \log\log n + \mathsf{occ})$	[101]
$\mathcal{O}(n)$ bits	$\mathcal{O}((\lvert\Sigma\rvert d)^k \log^2 n + \mathsf{occ} \log\log n)$	[87]
	$\mathcal{O}(((\lvert\Sigma\rvert d)^k \log\log n + \mathsf{occ}) \log^\delta n)$	[101]
	$\mathcal{O}((d + (c \log n)^{k^2+2k} \log\log n + \mathsf{occ}) \log^\delta n)$	[38]

Figure 2.2: Upper bounds for text indexing with k mismatches. Here α is any integer in $[2, n/2]$, $c > 1$, $\delta > 0$ are constants, and occ is the total number of the substrings that we output. For text indexing with k edits, the term occ is replaced with $3^k\mathsf{occ}$.

handle edit distance. Subsequent work has mainly focused on improving the space requirements [37, 38, 87, 101]. An exception is the work of Tsur [134], who presented an index with improved query time at the expense of increased space usage. A summary of these results is provided in Figure 2.2.

2.2 Contributions of the author

The principal area of the author's research is algorithms and data structures for efficient processing of strings and, more generally, formal languages. Her main research interests include small-space algorithms and data structures, streaming algorithms, property testing, and lower bounds. Her interest in the main topics of this proposal, namely, approximate and randomised string processing, text indexing, and applications in formal languages has gradually developed since completing her Ph.D. in 2013. This thesis overviews five papers representing her contribution on these topics. The rest of this section gives a background behind these papers. The papers co-signed by the author are highlighted in blue.

2.2.1 Streaming pattern matching

This section surveys the literature on approximate pattern matching in the streaming model, highlighting the author's contributions. Throughout this section, we assume a pattern P of length m and a text T of length n are given. We focus on the threshold version of approximate pattern matching.

Definition 1 *A substring of the text is called a k-mismatch (respectively, k-edit) occurrence of P if it is at Hamming (respectively, edit) distance at most k from P.*

In the streaming setting, the pattern arrives first. After preprocessing, the algorithm receives the text one character at a time, and reports whenever a suffix of the current prefix of the text forms a k-mismatch (or k-edit) occurrence of the pattern. The resources considered in this setting are the maximum space used by the algorithm after the arrival of the first character of the text, and the time required to process each character, either in the worst-case or amortised sense. The complexity of preprocessing the pattern is not included in the complexity bounds.

All algorithms discussed in this section are randomized and correct with high probability, i.e., with probability at least $1 - 1/n^c$ for some constant $c > 1$. We use $\tilde{\mathcal{O}}(\cdot)$ notation to suppress factors that are polylogarithmic in n.

2.2.1.1 The k-mismatch pattern matching We now present a survey of the literature on streaming algorithms for k-mismatch pattern matching. The results for this problem are summarised in Table 2.1. The currently most space-efficient algorithm, due to Clifford, Kociumaka, and Porat, SODA'19 [48] is the result of a long line of improvements and its space complexity, up to polylogarithmic factors, matches the known lower bound of $\Omega(k)$ bits, as established in [86].

Algorithm	Space	Time (total)
Porat and Porat [120]	$\tilde{\mathcal{O}}(k^3)$	$\tilde{\mathcal{O}}(k^2 n)$
Clifford et al. [46]	$\tilde{\mathcal{O}}(k^2)$	$\tilde{\mathcal{O}}(\sqrt{k}n)$
Radoszewski and Starikovskaya [122, 123]†	$\tilde{\mathcal{O}}(k^2)$	$\tilde{\mathcal{O}}(kn)$
Golan et al. [82]	$\tilde{\mathcal{O}}(k)$	$\tilde{\mathcal{O}}(kn)$
Clifford et al. [48]†	$\tilde{\mathcal{O}}(k)$	$\tilde{\mathcal{O}}(\sqrt{k}n)$
Golan et al. [81]	s for $k \le s \le m$	$\tilde{\mathcal{O}}(n + \min\{\frac{nk^2}{m}, \frac{nk}{\sqrt{s}}, \frac{\sigma nm}{s}\})$

Table 2.1: Streaming algorithms for approximate pattern matching under the Hamming distance. Algorithms, marked with †, provide an error correcting feature, which allows not only to report k-mismatch occurrences, but also the mismatches between them and the pattern. The algorithm of Clifford et al. [48] only does so at request in $\tilde{\mathcal{O}}(k)$ extra time per occurrence.

We now provide more detail about each of the algorithms in Table 2.1.

Porat and Porat, FOCS'09 [120] The study of streaming complexity of k-mismatch pattern matching was initiated by Porat and Porat [120]. Their algorithm runs in $\tilde{\mathcal{O}}(k^3)$ space and $\tilde{\mathcal{O}}(k^2)$ time per arriving symbol. This complexity was achieved through an elegant reduction to exact pattern matching via the Chinese remainder theorem. Namely, they showed that it suffices to partition the pattern and the text into $\tilde{\mathcal{O}}(k)$ subpatterns and subtexts, and to count the number of subpattern-subtext pairs that do not match.

Clifford et al., SODA'16 [46] In this work, we presented an improved streaming algorithm for pattern matching under the Hamming distance that uses $\tilde{\mathcal{O}}(k^2)$ space and $\tilde{\mathcal{O}}(\sqrt{k})$ time per character of the text. The first step toward achieving better complexity is to distinguish between patterns with big approximate period and small approximate period. An approximate period is an integer ρ such that the Hamming distance between the pattern and its copy shifted by ρ is at most k. Intuitively, small approximate period implies repetition in the pattern and in the region of the text containing its k-mismatch occurrences, which allows one to encode both the pattern and the region of the text in small space and process them efficiently. On the other hand, if the approximate period is large, then the k-mismatch occurrences of the pattern are spaced apart. By applying a randomized reduction to streaming exact pattern matching similar to that of [120], we obtained an efficient streaming $(1+\varepsilon)$-approximate pattern matching algorithm. By efficient, we mean that it uses less space and less time than a streaming algorithm computing the Hamming distances exactly. This algorithm is used as a filtering step. Candidate k-mismatch occurrences are then verified using a slower algorithm (since the candidates are spaced apart). **Further details of this result are presented in Chapter 4.**

Radoszewski and Starikovskaya, DCC'17 [122] (see also [123] In this work, we augmented the classical definition of k-mismatch pattern matching with what we called *the error correcting feature*. In short, the error correcting feature allows not only to find k-mismatch occurrences of the pattern in the text, but also the mismatches between these occurrences and the pattern. This feature proved to be a powerful tool. We demonstrated this by using it to develop small-space streaming algorithms for the problem of pattern matching on weighted strings. A weighted string is a sequence of probability distributions on the alphabet and is a commonly used representation of uncertain sequences in molecular biology. In particular, they are commonly used to model motifs and are integral to many software tools for computational motif discovery; see e.g. [129, 138]. In the weighted pattern matching problem, we are given a text and a pattern, both of which are weighted strings, and must find all alignments of the text and of the pattern where there exists a regular string that matches both the text and the pattern with probability larger than a given threshold. As weighted pattern matching is not a primary focus of this thesis, we do not elaborate further. An interested reader can refer to [122, 123]. To implement this feature, we first developed a solution for the case $k = 1$. The main idea is that the indices of the subpatterns that do not match can give the position of the mismatch by the Chinese remainder theorem. For $k > 1$, the solution is similar, but subdivides the pattern into more subpatterns and the text into more subtexts. The complexity of the algorithm is similar to that of [46].

Golan, Kopelowitz, Porat, ICALP'18 [82] In the multiple-patterns, multiple-texts problem one is given a set of patterns and a set of streaming texts, and must detect substrings of the texts that match the patterns. A straightforward solution is to run multiple instances of the streaming exact pattern matching algorithm. However, Golan, Kopelowitz, and Porat [82] demonstrated that by sharing space across streams, an improved solution is achievable. As a corollary, leveraging the reduction by Clifford et al. [46], they obtained an $\tilde{\mathcal{O}}(k)$-space and $\tilde{\mathcal{O}}(k)$-time streaming algorithm for k-mismatch.

Clifford, Kociumaka, Porat, SODA'19 [48] The current state-of-the-art algorithm was presented by Clifford, Kociumaka, Porat, SODA'19 [48]. It uses $\mathcal{O}(k \log \frac{n}{k})$ space and $\tilde{\mathcal{O}}(\sqrt{k})$ time per character of the text. Moreover, similarly to the algorithm of Radoszweski and Starikovskaya [122], it incorporates the error correction feature. The most noteworthy aspect of their work, however, lies in the techniques employed to achieve this result. Instead of using the randomised reduction of [46], they turned back to the idea of the original streaming exact matching algorithms. Namely, they consider prefixes of the pattern of exponentially increasing lengths. If the current text ends with a k-mismatch occurrence of a prefix of the pattern of length 2^i, then 2^{i-1} characters before the algorithm must have detected a k-mismatch occurrence of a prefix of the pattern of length 2^{i-1}. This observation enables the algorithm to use k-mismatch occurrences for the level-$(i-1)$ prefix of length 2^{i-1} to incur k-mismatch occurrences for the level-i prefix of length 2^i. Two main challenges arise: first, demonstrating that occurrences at level i can be stored efficiently, despite the complexity introduced by mismatches. Second, an efficient method is required to verify whether a k-mismatch occurrence of the level-$(i-1)$ prefix can be extended to a k-mismatch occurrence of the level-i prefix. To address this, the authors introduce a novel *k-mismatch sketch*, akin to the Karp–Rabin fingerprint [94]. Their construction is based on the ideas behind Reed–Solomon error correcting codes [124] and yields a sketch that requires only $\tilde{\mathcal{O}}(k)$ space. Moreover, like the Karp–Rabin fingerprint, it enjoys several desirable algorithmic properties, including efficient maintenance under concatenation and prefix removal. Combining these approaches allowed them to obtain the final result. Notably, the algorithm's space complexity is close to optimal, while its time complexity matches that of the offline algorithm [12].

Golan et al., CPM'20 [81] However, in the offline setting, more time-efficient algorithms exist; for instance, the algorithm of Chan et al. [39] runs in $\mathcal{O}(n + \min(k^2 n/m, kn\sqrt{\log m}/\sqrt{m}))$ time. Golan et al. [81] posed the question of whether a streaming algorithm with a time-space trade-off exists for k-mismatch pattern matching. They provided a positive answer by presenting an algorithm that, given an integer parameter $k \le s \le m$, uses s space and $\tilde{\mathcal{O}}(n + \min\{\frac{nk^2}{m}, \frac{nk}{\sqrt{s}}, \frac{\sigma nm}{s}\})$ total time, where σ is the size of the alphabet. The algorithm distinguishes between patterns with small and large approximate periods. For patterns with a small approximate period, it relies on the techniques of Clifford et al. [46]; and for patterns with a large approximate period, it uses the fact that the occurrences are spaced apart.

Gawrychowski and Starikovskaya [77] To conclude this section, we highlight one additional result that somewhat stands apart from the others. In [77], we studied the question of multiple pattern matching under the Hamming distance. In this problem, we are given a set of d patterns with total length m, and must report every position i in the text that corresponds to the endpoint of a k-mismatch occurrence of at least one of these patterns. A straightforward approach runs d independent copies of the algorithm [48], resulting in $\tilde{\mathcal{O}}(dk)$ space and $\tilde{\mathcal{O}}(dk)$ time per character. We demonstrated that by using slightly larger space $\tilde{\mathcal{O}}(dk \log^k d)$, the problem can be solved in $\tilde{\mathcal{O}}(k \log^k d)$ time per character, which is better than $\tilde{\mathcal{O}}(dk)$ for small k. Our approach builds upon a modification of the text index of Cole, Gottlieb, and Lewenstein [52] (see Section 2.1.2). We developed a randomized implementation of this text index. It uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and, assuming access to Hamming distance sketches of the prefixes of a query string, can answer queries in $\tilde{\mathcal{O}}(k \log^k d)$ time. The modified text index is used both for short patterns and for detecting k-mismatch occurrences of aperiodic suffixes of long patterns.

2.2.1.2 The k-edit pattern matching

For the edit distance, no sublinear-space streaming algorithm was known prior to 2017. The primary challenge in developing such algorithms has been (and continues to be) the lack of sketches that can be efficiently updated under concatenation of two strings, analogous to Karp–Rabin fingerprints for exact pattern matching or the Hamming distance sketches [48]. Table 2.2 summarizes existing algorithms.

Algorithm	Space	Time (total)
Starikovskaya, CPM'17 [132]	$\mathcal{O}(k^8\sqrt{m}\log^6 m)$	$\tilde{\mathcal{O}}(n(k^2\sqrt{m}+k^{13}))$
Kociumaka, Porat, and Starikovskaya, FOCS'21 [98]	$\tilde{\mathcal{O}}(k^5)$	$\tilde{\mathcal{O}}(n \cdot k^8)$
Bhattacharya and Koucky [23]	$\tilde{\mathcal{O}}(k^2)$	$\tilde{\mathcal{O}}(n \cdot k^2)$

Table 2.2: Streaming algorithms for approximate pattern matching under the edit distance.

Starikovskaya, CPM'17 [132] In CPM'17, we presented the first sublinear-space algorithm for k-edit pattern matching. The algorithm uses $\mathcal{O}(k^8\sqrt{m}\log^6 m)$ space and $\mathcal{O}((k^2\sqrt{m}+k^{13}) \cdot \log^4 m)$ worst-case time per character of the text. This complexity was achieved thanks to the (non-concatenable) edit distance sketches of Belazzougui and Zhang, FOCS 2016 [20]. They demonstrated the existence of a sketch mapping strings of length at most n to strings of length $\tilde{\mathcal{O}}(k^8)$ that can be efficiently computed in streaming and such that given the sketches of two strings one can compute the edit distance between them if it does not exceed k. The main idea of the algorithm [132] is straightforward: it divides the text into blocks of size $\Theta(\sqrt{m})$, and begins computing the sketch at each block boundary. Simultaneously, the algorithm stores the sketches of the $\Theta(\sqrt{m})$ longest suffixes of the pattern. Notably, for any alignment of the pattern and the text, the starting position of at least one of the suffixes is aligned with a block border, which allows computing the edit distance between the suffix of the pattern and the text. The remaining task is to compute the edit distance between the prefix and the text. For that, we show an efficient encoding of the blocks of the text, which uses a read-only access to the prefix of the text of length $\Theta(\sqrt{m})$.

Kociumaka, Porat, and Starikovskaya, FOCS'21 [98] In this work, we significantly improved the streaming complexity of k-edit pattern matching by presenting a randomized algorithm that uses $\tilde{\mathcal{O}}(k^5)$ space and $\tilde{\mathcal{O}}(k^8)$ amortized time per character of the text. Similarly to the algorithm of [48], our solution considers prefixes of the pattern of exponentially increasing lengths. Furthermore, it distinguishes between non-periodic and periodic prefixes (where periodicity is defined with respect to the edit distance). By [40], the number of k-edit occurrences of non-periodic prefixes is bounded by $\tilde{\mathcal{O}}(\text{poly}(k))$. This means that one can store the edit distance sketch of the suffix of the text starting at each of these occurrences, allowing efficient verification of whether it can be extended to a k-edit occurrence of the next-level prefix. In contrast, the number of k-edit occurrences of periodic prefixes can be large, making it infeasible to compute a sketch for each. However, by [40] such occurrences must lie within a periodic region of the text. This allows us to store the sketch for only the last occurrence and then efficiently reconstruct the sketches for earlier occurrences in the region using the known period. **We provide more details on this result in Chapter 5.**

Bhattacharya and Koucky [22] The current best algorithm was presented in [22]. It uses $\tilde{\mathcal{O}}(k^2)$ space and $\tilde{\mathcal{O}}(k^2)$ time per character of the text. The core of their approach is a grammar-based decomposition technique [22], which partitions a string into a small number of blocks, each encoded by a small grammar. This technique enabled a reduction from k-edit pattern matching to $\tilde{\mathcal{O}}(k^2)$-mismatch pattern matching.

2.2.2 Lower bounds for approximate text indexing

Despite significant progress in approximate text indexing, no text index (not even our randomised implementation [77]) has surpassed the time-space barrier suggested by Navarro in 2010: For k being the maximum allowed number of mismatches or differences, either the index size or the query time must depend on k exponentially.

In Cohen-Addad, Feuilloley, Starikovskaya, SODA'19 [51], we partially confirmed Navarro's conjecture by giving lower bounds, showing that indeed getting over this barrier is beyond the current techniques. We focused on two models of computation:

the classical RAM model and the pointer machine model. In the RAM model, we established conditional lower bounds based on the now-standard Strong Exponential Time Hypothesis (SETH), and in the pointer machine model, we provided unconditional lower bounds. The pointer machine model is more restricted than the RAM model, but it is particularly relevant for approximate text indexing, as all known solutions are pointer-machine data structures.

Approximate text indexing is closely related to approximate dictionary look-up, where one is given a dictionary of patterns of length n, and given a query string must decide whether there is a pattern at Hamming (edit) distance at most k from any pattern.

The earliest lower bounds for this problem were shown in [17, 28]. They showed that for some constant $c > 0$, $m \in w(\log n \cap n^{o(1)})$, and $k = m/2 - c\sqrt{m}\log n$, any randomised two-sided error cell-probe algorithm for dictionary look-ups with k mismatches that is restricted to use $\text{poly}(n, m)$ cells of size $\text{poly}(\log n, m)$ each, must probe at least $\Omega(m/\log n)$ cells. Since the cell-probe model is stronger than the RAM model, this lower bound implies $\Omega(m/\log n)$ query time lower bound for the classic RAM data structures. We note, however, that the lower bound is not that high while the value of k is rather large, $k = w(\log n \cap n^{o(1)})$.

Rubinstein [125] established a lower bound for approximate dictionary look-up under the Hamming distance by reduction from bichromatic closest pair under the Hamming distance. In this problem, one is given two sets of n binary strings, blue and red, and must find one blue string and one red string with the smallest Hamming distance between them. It is known [10, 125] that solving this problem requires $\Omega(n^{2-\delta})$ time, for any constant $\delta > 0$, conditional on the Strongly Exponential Time Hypothesis (SETH).

Conjecture 2.1 (SETH [88]) *For any $\delta > 0$, there exists $m = m(\delta)$ such that SAT on m-CNF formulas with n variables cannot be solved in time $\mathcal{O}(2^{(1-\delta)n})$.*

In [51], we extend the lower bound to approximate text indexing under the Hamming distance. Next, we introduce a transform that, applied to strings of length n with Hamming distance k between them, constructs two strings of length $\mathcal{O}(n \log n)$ with edit distance k between them. We refer to this transform as the *stoppers transform*. The stoppers transform allowed us extending the lower bounds to the edit distance as well. (See Figure 2.3 for a summary.)

In the second part of the paper, we establish lower bounds in the pointer machine model. We begin by proving a lower bound for approximate dictionary look-up under the Hamming distance.

To this end, we use the framework introduced by Afshani [7], initially introduced to show lower bounds for simplex range reporting and related problems. This framework was later used by Afshani and Nielsen [8] to show a lower bound for dictionary look-up with k don't cares. In this problem, the query strings may contain up to k don't care symbols, that is, special symbols that match any character of the alphabet, and the task is to retrieve all dictionary strings that match the

$$\begin{tikzcd}
\begin{array}{c}\textbf{Bichrom. closest pair}\\ \textbf{(Ham.)}\\ \Omega(n^{2-\delta})\text{ time ([10, 125])}\end{array} \arrow[r] \arrow[d, "\text{Stoppers transform, [125]}"] & \begin{array}{c}\textbf{Dictionary look-up}\\ \textbf{(Ham.)}\\ \Omega(n^{1-\delta})\text{ query time ([125])}\end{array} \arrow[r] & \begin{array}{c}\textbf{Approximate}\\ \textbf{text indexing}\\ \textbf{(Ham.)}\\ \Omega(n^{1-\delta})\text{ query time}\\ \text{([51])}\end{array} \\
\begin{array}{c}\textbf{Bichrom. closest pair}\\ \textbf{(edit)}\\ \Omega(n^{2-\delta})\text{ time}\\ \text{([125], [51])}\end{array} \arrow[r] & \begin{array}{c}\textbf{Dictionary look-up}\\ \textbf{(edit)}\\ \Omega(n^{1-\delta})\text{ query time}\\ \text{([125], [51])}\end{array} \arrow[r] & \begin{array}{c}\textbf{Approximate}\\ \textbf{text indexing}\\ \textbf{(edit)}\\ \Omega(n^{1-\delta})\text{ query time}\end{array}
\end{tikzcd}$$

Figure 2.3: Summary of lower bounds conditional on SETH [51]. These bounds hold for all data structures that can be constructed in polynomial time.

query string. Essentially, it is a parameterized variant of the *partial match* problem (see [121] and references therein). The structure of this problem is very similar to that of dictionary look-up with mismatches, as each don't care symbol in the query string gives $|\Sigma|$ possibilities for the corresponding symbol in the dictionary strings. On the other hand, it is simpler, as the positions of the don't care symbols are fixed. We extend their framework to the problem of dictionary look-up with mismatches, and then apply the stoppers transform to show lower bounds for the edit distance as well.

Finally, we show a reduction from dictionary look-up to text indexing which gives us lower bounds for text indexing. The main idea of the reduction is quite simple: We define the text as the concatenation of the dictionary strings interleaved with a special gadget string. The gadget string must guarantee that if we align the pattern with a substring that is not in the dictionary, the Hamming distance will be much larger than k. Via this reduction, the lower bounds for approximate dictionary look-up under the Hamming distance imply that for some value of k, there is no data structure for text indexing with k mismatches with sublinear query time unless SETH is false. This lower bound holds even if approximation is allowed. Similar bounds hold in the pointer machine model, see Figure 2.4 for an overview.

We give more details of this contribution in Chapter 6.

2.2.3 Formal languages membership

A formal language is, essentially, a set of strings. Pattern matching can be stated as a problem of recognition of a formal language, where the language is a singleton containing only the pattern. We have already seen that even this problem is non-trivial in streaming. An independent line of research has examined the problem of the recognition of classes of formal languages in streaming, such as regular and context-free languages, often yielding coarse-grained or even negative results [108, 70, 71, 68, 69, 74, 14, 66, 73, 19, 72]. In this section, we focus on applications of streaming (approximate) pattern matching. We consider languages

$$\begin{tikzcd}
\begin{array}{c}\textbf{Dictionary look-up (Ham.)}\\ \mathcal{O}(m+(\frac{\log n}{k})^k+occ)\text{ query time}\\ \implies \Omega(c^k n)\text{ space}\end{array} \arrow[r] \arrow[d, "\text{Stoppers transform}"] & \begin{array}{c}\textbf{Text indexing (Ham.)}\\ \mathcal{O}(m+(\frac{\log n}{2k})^k+occ)\text{ query time}\\ \implies \Omega(c^k n)\text{ space}\end{array} \\
\begin{array}{c}\textbf{Dictionary look-up (edit)}\\ \mathcal{O}(m/\log m+(\frac{\log n}{2k})^k+occ)\text{ query time}\\ \implies \Omega(c^k n)\text{ space}\end{array} &
\end{tikzcd}$$

Figure 2.4: Summary of pointer-machine lower bounds [51]. Here m is the length of the query string (pattern) and $c > 1$ is some constant. The bounds hold for all even k such that $\frac{8}{\sqrt{3}}\sqrt{\log n} \le k = o(\log n)$.

traditionally studied in string processing: regular languages, the language of palindromes, which is context-free but not regular, and the language of squares, which is not context-free. Although these languages belong to different levels of the Chomsky hierarchy [118, 43], these languages have a common feature: for any of them, one can fix a set of patterns such that any string in the language can be represented as a concatenation of the patterns. In this sense, these languages are particularly amenable to recognition via streaming pattern matching.

2.2.3.1 Regular languages membership

The fundamental notion of regular expressions was introduced back in the 1951 by Kleene [96]. Regular expression search is one of the key primitives in diverse areas of large scale data analysis: computer networks [100], databases and data mining [75, 106, 112], human-computer interaction [95], internet traffic analysis [91, 139], protein search [116], and many others. As such, this primitive is often the main computational bottleneck in these areas and in the pursuit for efficiency has been implemented in many programming languages: Perl, Python, JavaScript, Ruby, AWK, Tcl and Google RE2, to name just a few.

A regular expression R is a sequence containing characters of a specified alphabet Σ and three special symbols (operators): concatenation $(\cdot)$, union $(|)$, and Kleene star $(*)$, and it describes a set of strings $L(R)$ on Σ. For example, a regular expression $R = (a|b)^*c$ specifies a set of strings $L(R)$ on the alphabet $\Sigma = \{a, b, c\}$ such that their last character equals c, and all other characters are equal to a or b. There exist two classical formalisations of regular expressions search, regular expression membership and pattern matching. In the regular expression membership problem, we are given a string T of length n, and must decide whether $T \in L(R)$ for a given regular expression R. In the regular expression pattern matching problem, we must find all positions $1 \le r \le n$ such that for some $1 \le \ell \le r$, the substring $T[\ell\,..\,r] \in L(R)$.

Assume that T is read-only, and let m be the length of the regular expression. The classical algorithm by Thompson [133] allows to solve both problems in $\mathcal{O}(nm)$

time and $\mathcal{O}(m)$ space by constructing a non-deterministic finite automaton accepting $L(R)$. Galil [67] noted that while the space bound of Thompson's algorithm is optimal in the deterministic setting, the time bound could probably be improved. Since then, the effort has been mainly focused on improving the time complexity of regular expression search. The first breakthrough was achieved by Myers [113], who showed that both problems can be solved in $\mathcal{O}(mn/\log n + (n+m)\log n)$ time and $\mathcal{O}(mn/\log n)$ space. Bille and Farach-Colton [25] reduced the space complexity down to $\mathcal{O}(n^\varepsilon + m)$, for an arbitrary constant $\varepsilon > 0$. This result was further improved by Bille and Thorup [26] who showed an algorithm with running time $\mathcal{O}(nm(\log\log n)/\log^{3/2} n + n + m)$ time that uses $\mathcal{O}(n^\varepsilon + m)$ space. The idea of the algorithms by Myers [113], Bille and Farach-Colton [25], and Bille and Thorup [26] is to decompose Thompson's automaton into small non-deterministic finite automata and tabulate information to speed up simulating the behaviour of the original automaton when reading T. A slightly different approach was taken by Bille [24] who showed that the small non-deterministic finite automata can be simulated directly using the parallelism built-in in the RAM model. For w being the size of the machine word, Bille showed $\mathcal{O}(m)$-space algorithms with running times $\mathcal{O}(n\frac{m\log w}{w} + m\log w)$ for $m > w$, $\mathcal{O}(n\log m + m\log m)$ for $\sqrt{w} < m \le w$, and $\mathcal{O}(\min\{n+m^2, n\log m + m\log m\})$ for $m \le \sqrt{w}$. Finally, Bille and Thorup [27] identified a new parameter affecting the complexity of regular expression search, which is particularly relevant to this paper. Namely, they noticed that in practice a regular expression contains $d \ll m$ occurrences of the union symbol and Kleene stars, and showed that regular expression membership and pattern matching can be solved in $\mathcal{O}(m)$ space and $\mathcal{O}(n\cdot(\frac{d\log w}{w} + \log d))$ time.

It is easy to see, however, that in the general case the time complexity of all the algorithms above remains close to "rectangular", with some polylogarithmic factors shaved. Recently, fine-grained complexity provided an explanation for this. Backurs and Indyk [16] followed by Bringmann, Grønlund, and Larsen [34] considered a subclass of regular expressions which they refer to as "homogeneous". Intuitively, a regular expression is homogeneous, if the operators at the same level of the expression are equal. Assume that the alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. To give a few examples, the following regular expressions are homogeneous: $R_1 = (P_1|P_2|\ldots|P_d)$, $R_2 = P_1(1|2|\ldots|\sigma)P_2(1|2|\ldots|\sigma)\ldots(1|2|\ldots|\sigma)P_d$, and $R_3 = (P_1|P_2|\ldots|P_d)^*$, where P_i, $1 \le i \le d$, are strings on Σ, i.e. concatenations of characters in Σ. [16, 34] considered both the membership and the pattern matching problems. A careful reader might notice that in the pattern matching setting the expression R_1 corresponds to the classical dictionary matching problem [9] and R_2 to pattern matching with wildcards (don't cares) [65, 92, 54, 89, 44]. In the membership setting, R_3 corresponds to the word break problem [135, 104]. As such, a seemingly simple class of homogeneous regular expressions covers many classical problems in stringology. The authors of [16, 34] provided a complete dichotomy of the time complexities for homogeneous regular expressions in both settings. Namely, they showed that in both settings, every regular expression either allows a solution in near-linear time, or requires $\Omega((nm)^{1-\alpha})$ time, conditioned on SETH. The only exception is the word break problem in the membership setting, for which [34] showed an $\mathcal{O}(n(m\log m)^{1/3}+m)$-time algorithm and a matching combinatorial lower bound (up to polylogarithmic factors). Later, Abboud and Bringmann [4] took an even more fine-grained approach and showed that in general, regular expression pattern matching and membership cannot be solved in time $\mathcal{O}(nm/\log^{7+\alpha} n)$ for any constant $\alpha > 0$ under the Formula-SAT Hypothesis. Schepper [130] extended their result by revisiting the dichotomy for homogeneous regular expressions, and showed an $\mathcal{O}(nm/2^{\Omega(\sqrt{\log\min\{n,m\}})})$ time bound for some regular expressions, and for the remaining ones an improved lower bound of $\Omega(nm/\text{polylog } n)$.

In [59], we study the streaming complexity of regular expression membership and pattern matching, motivated by multiple practical applications. For a general regular expression membership and pattern matching, it is not hard to see that $\Omega(m)$ bits of space are required by a reduction from the set intersection problem. However, there are at least two interesting special cases of regular expression pattern matching that admit better streaming algorithms. In the dictionary matching, we are given a dictionary of d strings of length at most m over an alphabet Σ and for each position r in T must decide whether there is a position $\ell \leq r$ such that $T[\ell\,..\,r]$ matches a dictionary string. A series of work [120, 31, 45, 84, 82] showed that this problem can be solved by a streaming algorithm in $\mathcal{O}(d\log m)$ space and $\mathcal{O}(\log\log|\Sigma|)$ time per character of the text. In the d-wildcard pattern matching the expression is $R = P_1(1|2|\dots|\sigma)P_2(1|2|\dots|\sigma)\dots(1|2|\dots|\sigma)P_{d+1}$, where P_i, $1 \leq i \leq d+1$ are strings of total length at most m over an alphabet $\Sigma = \{1, 2, \dots, \sigma\}$. Golan, Kopelowitz, and Porat [83] showed that this problem can be solved by a streaming algorithm in $\mathcal{O}(d\log m)$ space and $\mathcal{O}(d + \log m)$ time per character. Note further that the d-wildcard pattern matching problem is a special case of the k-mismatch problem, and hence can be solved by an application of the algorithm of Clifford, Kociumaka and Porat [48] in $\mathcal{O}(d\log\frac{m}{d})$ space and $\mathcal{O}(\log\frac{m}{d}(\sqrt{d\log d}+\log^3 m))$ time per character.

As mentioned earlier, Bille and Thorup [27] observed that in practice the number d of occurrences of the union symbol and Kleene stars is significantly smaller than the size m of the expression R. Furthermore, both the dictionary matching and the wildcard pattern matching can be casted as instances of the regular expression pattern matching, and streaming algorithms with space complexity of the form $\text{poly}(d, \log n)$ are known. In [59], we showed that this is also the case for the general regular expression membership and pattern matching. More specifically, we design streaming algorithms that solve both problems using $\mathcal{O}(d^3\text{polylog } n)$ space and $\mathcal{O}(nd^5\text{polylog } n)$ time per character of the text.

On a very high level, our approach is based on storing carefully chosen subsets of occurrences of the strings appearing in R. Similar to streaming pattern matching algorithms, our approach is to treat periodic and aperiodic strings separately. The technical novelty of our algorithms is that we apply this reasoning on $\mathcal{O}(\log n)$ levels, thus obtaining a hierarchical decomposition of a periodic string. Next, because not all occurrences are stored we need to recover the omitted information. Very informally, we need to decide whether a substring of T sandwiched between two occurrences of strings A_1, A_2 is a label of some run from A_1 to A_2 in the compact Thompson automaton for R, where the period of the substring is equal to the period of some prefix of length 2^k of one of the strings. The difficulty is that, while the substring has a simple structure, it could be very long, and it is not clear how to implement this computation in a space-efficient manner. We overcome this difficulty by recasting the problem in the language of evaluating a circuit with addition and convolution gates, following and improving on the technique initially introduced in [107, 33] for the subset sum problem.

We provide more details in Chapter 7.

2.2.3.2 Palindromes and squares

In [18], we study two classical formal languages, the language of palindromes and the language of squares. The language of palindromes contains all strings that are equal to their reversed copies (in particular, even-length palindromes can be represented as the concatenation of a string and its reversed copy). The language of squares contains all strings that are the concatenation of two copies of a string. These two languages are very similar yet very different in nature: the language of palindromes is not regular but is context-free, whereas the language of squares is not even context-free.

While these languages are not regular, deciding whether a sting belongs to one of them is easy, even in streaming: for the languages of palindromes, it is enough to compute the Karp–Rabin fingerprint of the input and its reverse, and for squares the Karp–Rabin fingerprint of the first half of the input and of the input. Both problems can therefore be solved in streaming in $\mathcal{O}(1)$ space and $\mathcal{O}(1)$ time per character of the input.

In [18], we studied the complexity of a strictly harder language distance problem, focusing on the *online* and *low-distance* regime. In this regime, we are given a string T of length n, and the task is to compute the minimum distance from *every* prefix of T to a formal language if it does not exceed a given threshold k. [13] showed that there is a streaming algorithm that solves the problem in $\tilde{\mathcal{O}}(k)$ space and $\tilde{\mathcal{O}}(k^2)$ time per input character. We continue their line of research and give streaming algorithms using $\mathrm{poly}(k, \log n)$ time per character and $\mathrm{poly}(k, \log n)$ space for language distance membership for palindromes and squares under the Hamming and the edit distances.

As a corollary, we obtain new streaming algorithms for $(1+\varepsilon)$-approximation of the maximal length of a substring of the input string that is within Hamming (edit) distance at most k to a palindrome. The previous best algorithm for this problem under the Hamming distance is by [85], who extended the works [21, 76] and showed a solution that uses $\mathcal{O}(\frac{k \log^9 n}{\varepsilon \log(1+\varepsilon)})$ time per character and $\mathcal{O}(\frac{k \log^7 n}{\varepsilon \log(1+\varepsilon)})$ space. We significantly improve their result and give an algorithm that uses $\mathcal{O}((k/\varepsilon) \log^4 n)$ time per character and $\mathcal{O}((k/\varepsilon) \log^2 n)$ bits of space. We also give the first streaming algorithm for a similar problem under the edit distance which uses $\tilde{\mathcal{O}}(k^2/\varepsilon)$ time per character and space.

We provide further details in Chapter 8.

2.3 Organisation of the thesis

The remainder of the thesis is comprised of preliminaries, where we give basic definitions and notation, and of five chapters, each dedicated to an article co-signed by the author. The names of PhD students co-advised by the applicant are highlighted.

- Chapter 4: R. Clifford, A. Fontaine, E. Porat, B. Sach, T. Starikovskaya, *The k-mismatch problem revisited.* Proceedings of the 2016 ACM-SIAM Symposium on Discrete Algorithms (SODA 2016), pages 2039–2052 [46].
- Chapter 5: T. Kociumaka, E. Porat, T. Starikovskaya, *Small-space and streaming pattern matching with k edits.* Proceedings of the 62nd IEEE Annual Symposium on Foundations of Computer Science (FOCS 2021), pages 885–896 [98].
- Chapter 6: V. Cohen-Addad, L. Feuilloley, T. Starikovskaya, *Lower bounds for text indexing with mismatches and differences.* Proceedings of the 2019 ACM-SIAM Symposium on Discrete Algorithms (SODA 2019), pages 1146–1164 [51].
- Chapter 7: B. Dudek, P. Gawrychowski, G. Gourdel, T. Starikovskaya, *Streaming Regular Expression Membership and Pattern Matching.* Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms (SODA 2022), pages 670–694 [59].
- Chapter 8: G. Bathie, T. Kociumaka, T. Starikovskaya, *Small-Space Algorithms for the Online Language Distance Problem for Palindromes and Squares.* Proceedings of the 34th International Symposium on Algorithms and Computation (ISAAC 2023). LIPIcs, vol. 283, pages 10:1–10:17 [18].

3 Preliminaries

"A word after a word after a word is power."

Margaret Atwood, *Spelling*.

In this section, we introduce basic notation and definitions necessary for the subsequent chapters.

We assume an integer alphabet $\Sigma = \{1, 2, \dots, \sigma\}$ with σ *characters*. A *string* (or a word) Y is a sequence of characters numbered from 1 to $n = |Y|$. By $Y[i]$ we denote the i-th symbol of Y. For a string Y of length n, we denote its *reverse* $Y[n]Y[n-1]\dots Y[1]$ by $\mathsf{rev}Y$. We define $Y[i\,..\,j]$ to be equal to $Y[i]\dots Y[j]$ which we call a *fragment* of Y if $i \le j$ and to the empty string ε otherwise. We also use notations $Y[i\,..\,j)$ and $Y(i\,..\,j]$ which naturally stand for $Y[i]\dots Y[j-1]$ and $Y[i+1]\dots Y[j]$, respectively. We call a fragment $Y[1]\dots Y[j]$ *a prefix* of Y and use a simplified notation $Y[\,..\,i]$, and a fragment $Y[i]\dots Y[n]$ *a suffix* of Y denoted by $Y[i\,..\,]$. We say that X is a *substring* of Y if $X = Y[i\,..\,j]$ for some $1 \le i \le j \le |Y|$. The fragment $Y[i\,..\,j]$ is called an *occurrence* of X.

For a string Y, we define Y^m to be the concatenation of m copies of Y. We call Y^m a *power* of Y. We also define Y^∞ to be an infinite string obtained by concatenating infinitely many of copies of Y. We say that a string X of length x is a *period* of a string T if $X = T[1\,..\,x]$ and $T[i] = T[i+x]$ for all $i = 1, \dots, |T| - x$. By $\mathsf{per}(T)$ we denote the length of the shortest period of T. The string T is called *periodic* if $2\,\mathsf{per}(T) \le |T|$. For a string $Y \in \Sigma^n$, we define a *forward rotation* $\mathsf{rot}(Y) = Y[2]\cdots Y[n]Y[1]$. In general, a *cyclic rotation* $\mathsf{rot}^s(Y)$ with *shift* $s \in \mathbb{Z}$ is obtained by iterating rot or the inverse operation rot^{-1}. A non-empty string $X \in \Sigma^n$ is *primitive* if it is distinct from its non-trivial rotations, i.e., if $X = \mathsf{rot}^s(X)$ holds only when s is a multiple of n.

We say that a fragment $X[i\,..\,i+\ell)$ is a *previous factor* if $X[i\,..\,i+\ell) = X[i'\,..\,i'+\ell)$ holds for some $i' \in [1\,..\,i)$. The *LZ77 factorization* of X is a factorization $X = F_1 \cdots F_z$ into non-empty *phrases* such tht the jth phrase F_j is the longest previous factor starting at position $1 + |F_1 \cdots F_{j-1}|$; if no previous factor starts there, then F_j consists of a single character. In the underlying *LZ77 representation*, every phrase $F_j = T[j\,..\,j+\ell)$ that is a previous fragment is encoded as (i', ℓ), where $i' \in [1\,..\,i)$ satisfies $X[i\,..\,i+\ell) = X[i'\,..\,i'+\ell)$. The remaining length-1 phrases are represented by the underlying character. We use $\mathsf{LZ}(X)$ to denote the underlying *LZ77 representation* and $|\mathsf{LZ}(X)|$ to denote its size (the number of phrases).

3.1 Distances

The *Hamming distance* between two strings S, T (denoted $\mathsf{hd}(S,T)$) is defined to be equal to infinity if S and T have different lengths, and otherwise to the number of positions where the two strings differ (mismatches).

The *edit distance* $ed(X,Y)$ between two strings X and Y is defined as the smallest number of character insertions, deletions, and substitutions required to transform X to Y.

Definition 2 *A sequence $(x_t, y_t)_{t=1}^{m}$ is an alignment of $X, Y \in \Sigma^*$ if $(x_1, y_1) = (1,1)$, $(x_m, y_m) = (|X|+1, |Y|+1)$, and $(x_{t+1}, y_{t+1}) \in \{(x_t+1, y_t+1), (x_t+1, y_t), (x_t, y_t+1)\}$ for $t \in [1\,..\,m)$.*

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=1}^{m}$ of strings $X, Y \in \Sigma^*$, for every $t \in [1\,..\,m)$:

- If $(x_{t+1}, y_{t+1}) = (x_t+1, y_t)$, we say that $\mathcal{A}$ *deletes* $X[x_t]$,
- If $(x_{t+1}, y_{t+1}) = (x_t, y_t+1)$, we say that $\mathcal{A}$ *deletes* $Y[y_t]$,
- If $(x_{t+1}, y_{t+1}) = (x_t+1, y_t+1)$, we say that $\mathcal{A}$ *aligns* $X[x_t]$ and $Y[y_t]$, denoted $X[x_t] \sim_{\mathcal{A}} Y[y_t]$. If additionally $X[x_t] = Y[y_t]$, we say that $\mathcal{A}$ *matches* $X[x_t]$ and $Y[y_t]$, denoted $X[x_t] \simeq_{\mathcal{A}} Y[y_t]$. Otherwise, we say that $\mathcal{A}$ *substitutes* $X[x_t]$ for $Y[y_t]$.

The *cost* of an edit distance alignment $\mathcal{A}$ is the total number characters that $\mathcal{A}$ deletes or substitutes. We denote the cost by $\mathsf{cost}_{X,Y}(\mathcal{A})$, omitting the subscript if X, Y are clear from context. The cost of an alignment $\mathcal{A} = (x_t, y_t)_{t=1}^{m}$ is at least its width $\mathsf{width}(\mathcal{A}) = \max_{t=1}^{m} |x_t - y_t|$. Observe that $ed(X,Y)$ can be defined as the minimum cost of an alignment of X and Y. An alignment of X and Y is *optimal* if its cost is equal to $ed(X,Y)$.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=1}^{m}$ of $X, Y \in \Sigma^+$, we partition the elements (x_t, y_t) of $\mathcal{A}$ into *matches* (for which $X[x_t] \simeq_{\mathcal{A}} Y[y_t]$) and *breakpoints* (the remaining elements). We denote the set of matches and breakpoints by $\mathcal{M}_{X,Y}(\mathcal{A})$ and $\mathcal{B}_{X,Y}(\mathcal{A})$, respectively, omitting the subscripts if the strings X, Y are clear from context. Observe that $|\mathcal{B}_{X,Y}(\mathcal{A})| = 1 + \mathsf{cost}(\mathcal{A})$.

We call $M \subseteq [1\,..\,|X|] \times [1\,..\,|Y|]$ a *non-crossing matching* of $X, Y \in \Sigma^*$ if $X[x] = Y[y]$ holds for all $(x,y) \in M$ and there are no distinct pairs $(x,y), (x',y') \in M$ with $x \le x'$ and $y \ge y'$. Note that, for every alignment $\mathcal{A}$ of X, Y, the set $\mathcal{M}(\mathcal{A})$ is a non-crossing matching of X, Y.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=1}^{m}$ of X and Y, for every $\ell, r \in [1\,..\,m]$ with $\ell \le r$, we say that $\mathcal{A}$ *aligns* $X[x_\ell\,..\,x_r)$ and $Y[y_\ell\,..\,y_r)$, denoted $X[x_\ell\,..\,x_r) \sim_{\mathcal{A}} Y[y_\ell\,..\,y_r)$. If there is no breakpoint (x_t, y_t) with $t \in [\ell\,..\,r)$, we further say that $\mathcal{A}$ *matches* $X[x_\ell\,..\,x_r)$ and $Y[y_\ell\,..\,y_r)$, denoted $X[x_\ell\,..\,x_r) \simeq_{\mathcal{A}} Y[y_\ell\,..\,y_r)$.

An alignment $\mathcal{A} = (x_t, y_t)_{t=1}^{m}$ of $X, Y \in \Sigma^*$ naturally induces a unique alignment of any two fragments $X[x\,..\,x')$ and $Y[y\,..\,y')$. Formally, the *induced alignment* $\mathcal{A}_{[x\,..\,x'),[y\,..\,y')}$ is obtained by removing repeated entries from $(\max(x, \min(x', x_t)) - x + 1, \max(y, \min(y', y_t)) - y + 1)_{t=1}^{m}$.

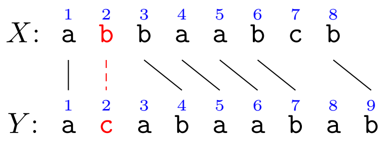


Figure 3.1: Consider strings $X = \texttt{abbaabcb}$ and $Y = \texttt{acabaabab}$ and a cost-4 alignment $\mathcal{A} : (1,1),(2,2),(3,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,8),(8,9),(9,10)$. The breakpoints are $\mathcal{B}(\mathcal{A}) = \{(2,2),(3,3),(7,8),(8,8),(9,10)\}$; the first 4 breakpoints correspond to a substitution of $X[2]$ for $Y[2]$, a deletion of $Y[3]$, a deletion of $X[7]$, and a deletion of $Y[8]$, respectively. Graphically, the alignment is depicted on the right; the aligned pairs of characters are connected with an edge, and the substituted pair is highlighted.

Fact 3.1 *If an alignment $\mathcal{A}$ satisfies $X[x\,..\,x') \sim_{\mathcal{A}} Y[y\,..\,y')$, then $|x-x'|, |y-y'| \le \mathsf{width}(\mathcal{A})$ and*

$$|(x'-x)-(y'-y)| \le ed(X[x\,..\,x'), Y[y\,..\,y')) \le \mathsf{cost}(\mathcal{A}_{[x\,..\,x'),[y\,..\,y')}) \le \mathsf{cost}(\mathcal{A}).$$

4 Streaming algorithm for k-mismatch pattern matching

"We live by patterns. We die when patterns break."

Frank Herbert, *God Emperor of Dune*

In this section, we give an overview of the main ideas of our streaming algorithm for k-mismatch pattern matching [46].

Let us start with a formal statement of the pattern matching with k mismatches problem. Let $\mathsf{hd}(P, S)$ be the Hamming distance between length m strings P and S. We say that $T[\ell \mathinner{..} r]$ is a *k-mismatch occurrence* of P if $\mathsf{hd}(P, T[\ell \mathinner{..} r]) \leq k$, and we denote the set of the *right* endpoints of the k-mismatch occurrences of P in T by $\mathsf{occ}_k^H(P, T)$.

Problem 4.1 (k-mismatch pattern matching) *Given a pattern P of length m over an alphabet Σ, a text T of length n over Σ, and an integer k, compute $\mathsf{occ}_k^H(P, T)$.*

Next, we introduce the notion of the approximate period, or x-period of a string, which we use to separate our problems into two cases. Let $\mathsf{hd}(P, T)[i]$ be $\mathsf{hd}(P, T[i - m + 1 \mathinner{..} i])$.

Definition 3 *The x-period of a string P of length m is the smallest integer $\pi > 0$ such that $\mathsf{hd}(P[\pi \mathinner{..} m-1], P[0 \mathinner{..} m-1-\pi]) \leq x$. (For example, the 1-period of a string babaa is 2.)*

Let ℓ be the $3k$-period of the pattern P and as our first of two cases, consider when $\ell \leq k$. We call this the small approximate period case and as we will see, the solution for this case contains some of the main ideas on which our other results will rely.

Fact 4.1 *If the $3k$-period of the pattern is ℓ, then each two $(3k/2)$-mismatch occurrences of the pattern in the text must be at least ℓ characters apart.*

4.1 Small approximate period ($\ell \leq k$) case.

Before we explain the idea of our solution in this case, let us remind the definition of a *run length encoding*. A run of a string is simply a maximal substring equal to a power of a character. For example, the runs of a string $S = aaabbbbbbcccc$ are aaa, $bbbbbb$, and $cccc$. Each run can be encoded as a character and the number of times it repeats (the *power*). The run length encoding of a string is the concatenation of the encodings of the runs. For example, the run length encoding of S is $(a, 3)(b, 6)(c, 4)$.

The main new idea of our solution in the small approximate period case is to reduce the k-mismatch pattern matching problem to $\mathcal{O}(k^2)$ small instances of run length encoded pattern matching. There are a number of surprising elements to our solution. The first one is that in any substring of the text of length $2m$ we can find a compressible region that contains all the alignments of the pattern and text with Hamming distance at most k. The second is that by choosing a suitable partitioning of the pattern and of this compressible region into $\mathcal{O}(k)$ subpatterns and $\mathcal{O}(k)$ subtexts respectively and then run length encoding those, we can ensure that the total number of runs, summed across all subpatterns and subtexts is only $\mathcal{O}(k)$. The third is that despite there being $\mathcal{O}(k)$ subpatterns and $\mathcal{O}(k)$ subtexts giving $\mathcal{O}(k^2)$ instances of the run length encoded Hamming distance problem, each of which can take $\mathcal{O}(k^2 \log k)$ time, we show that the time complexity of all the instances sums to only $\mathcal{O}(k^2 \log k)$. By the same approach, we demonstrate that the working space of all the instances sums to $\mathcal{O}(k^2)$. We also need to be careful when recovering the final Hamming distances because, in the worst case, each final distance is the sum of k outputs of the run length encoded Hamming distance problem. A naive summation would therefore result in an additive $\Omega(k)$ term per Hamming distance in the time complexity. To overcome this bottleneck, we take advantage of the compressed output to reduce the time taken to recover the final distances to $\mathcal{O}(m + k^2 \log k)$ per text substring of length $2m$.

Using a standard trick, we run our algorithm independently on $\mathcal{O}(n/m)$ substrings of the text of length $2m$, each overlapping the next by m characters. This results in an offline algorithm (that is, when both the pattern and the text are given simultaneously), the main steps of which are summarised in Algorithm 1. We then show that it can be implemented online by running the five steps of the offline algorithm in parallel.

Algorithm 1 Deterministic offline algorithm for k-mismatch when the pattern has small approximate period

```
Require: Pattern $P$ of length $m$ and text $T$ of length $2m$
1: Identify a compressible region of $T$ containing all $k$-mismatch occurrences
2: Partition this region into $\mathcal{O}(k)$ subtexts and $P$ into $\mathcal{O}(k)$ subpatterns
3: Run-length encode all subpatterns and subtexts
4: Compute run-length encoded Hamming distances for each subpattern/subtext
   pair
5: Sum the Hamming distances from previous line
```

Lemma 4.1 *Consider a pattern P of length m, and a text T of length n arriving online. If the $3k$-period of P is smaller than k, then the k-mismatch pattern matching* *problem can be solved in* $\mathcal{O}(k^2)$ *space and* $\mathcal{O}(nk^2 \log k/m + n)$ *time.*

4.2 Large approximate period ($\ell > k$) case.

For this case, we give two algorithms: a deterministic offline one and a randomised streaming algorithm. The overall structure of the algorithms is the same:

1. Filter out all alignments of the pattern and text with Hamming distance greater than $3k/2$.
2. Verify whether the Hamming distance is at most k at those positions.

Offline algorithm In the deterministic offline algorithm, the filtering step is implemented by running Karloff's $(1+\varepsilon)$-approximation algorithm [93] with $\varepsilon = 1/2$, excluding all positions which are reported to have Hamming distance greater than $3k/2$. This takes $\mathcal{O}(\log^3 m)$ time per character of the text and space $\mathcal{O}(n)$. The verification step is implemented via the kangaroo jumps technique, which takes $\mathcal{O}(k)$ time per character of the text and space $\mathcal{O}(n)$ [102]. We need only run the verification step at alignments that have not been filtered out by the filtering step. By Fact 4.1 there can be no more than one such alignment for every k consecutive text characters that arrive. It follows that the total amortised time for the large approximate period case is $\mathcal{O}(n\text{polylog } m)$. This completes the description of the deterministic algorithm. Combined with Lemma 4.1, this gives a deterministic offline algorithm with runtime $\mathcal{O}(nk^2 \log k/m + n\text{polylog } m)$ and space $\mathcal{O}(n)$.

Streaming algorithm To establish a randomised streaming algorithm for k-mismatch pattern matching for the large period case, we need small-space versions of both the filtering and verification steps. Let us first discuss the verification step. In the same way as in the deterministic case, after filtering the verification step will only need to verify at most one potential k-mismatch per k consecutive text characters. To do this efficiently we maintain a dynamic data structure that allows us to query the Hamming distance between P and the latest m-length suffix of the text and will output the exact distance if it is at most k and "No" otherwise. Each time a new character of the text arrives we perform an update.

Lemma 4.2 *For a given pattern P of length m, and an online text T of length n there is a data structure which answers Hamming distance queries as described above and uses $\mathcal{O}(k^2\text{polylog } m)$ space, update time $O(\text{polylog } m)$, and query time $\mathcal{O}(k\text{polylog } m)$. If the Hamming distance does not exceed $2k$, the probability of error is at most $1/m^2$.*

The key technical innovation, which is set out in Lemma 4.2 is that our data structure takes only polylog m time to perform an update when a new text character arrives if no query is performed at that time. We use this asymmetry in query and update times combined with Fact 4.1 to show that the $n-m$ verification steps can be performed in $\mathcal{O}(n\text{polylog } m)$ time and $\mathcal{O}(k^2\text{polylog } m)$ space overall. Our solution for Lemma 4.2 works by first reducing the problem to repeated application of 1-mismatch, in a similar fashion to Porat and Porat [120] and then in turn reducing the 1-mismatch problem to the streaming dictionary matching problem. However, our method differs significantly in technique from [120] both by randomising the first reduction step and then in our second reduction step which allows us to perform updates much more quickly than queries.

We now explain how to implement the filtering step. The main new ideas for our approximation algorithm are a novel randomised length reduction scheme and a two stage approximation scheme. The general idea is as follows. First, during preprocessing we reduce the length of the pattern to be only $O(k \log^2 m)$. We then overcome a particularly significant technical hurdle by showing how to transform the text in such a way that any Hamming distance between the reduced length pattern and the transformed text provides a reasonable approximation of the corresponding Hamming distance in the original input. Finally we apply an existing linear space online $(1+\varepsilon)$-approximation algorithm to the reduced length pattern and the transformed text to give the final approximate answer. The entire process is repeated independently in parallel a logarithmic number of times to improve the error probability. We argue that this approximation of an approximation still gives us a $(1+\varepsilon)$-approximation to the true Hamming distance at each alignment with good probability.

Deamortisation using the tail trick We finally explain how to deamortise our streaming k-mismatch algorithm with $O(nk^2 \log k/m + n\text{polylog } m)$ run-time to give us a fast worst-case time streaming algorithm. We first observe that if the pattern length m is at most $2k^2$, we can run an existing algorithm [49] which will take $O(\sqrt{k} \log k)$ time per character and uses linear space, which in this case is $O(k^2)$. We now proceed under the assumption that $m > 2k^2$.

To deamortise the algorithm, we use a two part partitioning that we call the *tail trick*. Similar ideas were also used to deamortise streaming pattern matching algorithms in [45, 49]. We partition the pattern into two parts: the *tail*, P_t — the suffix of P of length $2k^2$, and the *head*, P_h — the prefix of P length $(m - 2k^2)$. We will compute the current Hamming distance, $\mathsf{hd}(P,T)[i]$ by summing $\mathsf{hd}(P_t,T)[i]$ and $\mathsf{hd}(P_h,T)[i-2k^2]$. To compute $\mathsf{hd}(P_t,T)[i]$ we again use the existing linear space online k-mismatch algorithm from [49] taking $O(\sqrt{k} \log k)$ time per character and $O(k^2)$ space.

We also need to make sure that when the i-th character of the text, $T[i]$, arrives, we will have computed $\mathsf{hd}(P_h,T)[i-2k^2]$ in time. To this end we run the amortised algorithm using pattern P_h. However, we cap the run-time at $O(\text{polylog } m)$ per character. That is, when $T[i]$ arrives we run polylog m steps of the algorithm. Because the algorithm is amortised, it may lag behind the text stream: When $T[i]$ arrives, it may still be processing $T[i']$ for some $i' < i$. Fortunately, the lag cannot exceed $2k^2$, that is at all times $i - i' \leq 2k^2$. This is because we are able to show that while processing any k^2 consecutive text characters the total time complexity of the algorithm, summed over those consecutive characters is upper bounded by $O(k^2 \log k) = O(k^2\text{polylog } m)$. To allow for the lag in the deamortisation process we also maintain a buffer containing the most recently arrived $2k^2$ text characters and the most recent $2k^2$ outputs.

The algorithm uses $O(k^2\text{polylog } m)$ space. The time complexity is the sum of the complexities for processing P_t and P_h which is $O(\sqrt{k}\log k + \text{polylog } m)$ per arriving character of the text.

5 Streaming algorithm for k-edit pattern matching

"The difference between the almost right word and the right word is really a large matter — 'tis the difference between the lightning bug and the lightning."

Mark Twain, *in a letter to George Bainton, 1888*

In this chapter, we provide an overview of our k-edit pattern matching algorithm presented in [98].

Let us start with a formal statement of the pattern matching with k edits problem. We say that $T[\ell\,..\,r]$ is a *k-edit occurrence* of P if $ed(P, T[\ell\,..\,r]) \le k$, and we denote the set of the *right* endpoints of the k-edit occurrences of P in T by $\mathsf{occ}^E_k(P,T)$.

Problem 5.1 (k-edit pattern matching) *Given a pattern P of length m over an alphabet Σ, a text T of length n over Σ, and an integer k, compute $\mathsf{occ}^E_k(P,T)$.*

We consider the problem in the streaming setting, and, in addition, in the asymmetric streaming setting, where the algorithm can no longer access $T[1\,..\,r)$ while processing $T[r]$, but it is given an oracle providing read-only constant-time access to individual characters of P. This oracle is not counted towards the space complexity of the algorithm. For the semi-streaming setting, we provide a deterministic solution, whereas our solution for the streaming setting is Monte-Carlo randomized.

5.1 (Semi-)Streaming Algorithm for Pattern Matching with k Edits

Our algorithms solve a slightly stronger problem: every element $r \in \mathsf{occ}^E_k(P,T)$ is augmented with the smallest integer $k' \in [0\,..\,k]$ such that $r \in \mathsf{occ}^E_{k'}(P,T)$. At a very high-level, we reuse the structure of existing streaming algorithms for exact pattern matching and the k-mismatch problem [120, 31, 46, 48]. Namely, we consider $\mathcal{O}(\log m)$ prefixes $P_i = P[1\,..\,\ell_i]$ of exponentially increasing lengths ℓ_i. The algorithms are logically decomposed into $\mathcal{O}(\log m)$ levels, with the ith level receiving $\mathsf{occ}^E_k(P_{i-1},T)$ and producing $\mathsf{occ}^E_k(P_i,T)$. In other words, the task of the ith level is determine which k-edit occurrences of P_{i-1} can be extended to k-edit occurrences of P_i. When the algorithm processes $T[r]$, the relevant positions $p \in \mathsf{occ}^E_k(P_{i-1}, T)$ are those satisfying $|r - p - (\ell_i - \ell_{i-1})| \leq k$. Since each $p \in \mathsf{occ}^E_k(P_{i-1}, T)$ is reported when the algorithm processes $T[p]$, we need a buffer storing the *active* k-edit occurrences of P_{i-1} (i.e. those that can still be extended into k-edit occurrences of P_i). We implement it using a recent combinatorial characterization of k-edit occurrences [40], which classifies strings based on the following notion of approximate periodicity:

Definition 4 *A string X is k-periodic if there exists a primitive string Q with $|Q| \leq |X|/128k$ such that the edit distance between X and a prefix of Q^∞ is at most $2k$. We call Q a k-period of X.*

The main message of [40] is that only k-periodic strings may have many k-edit occurrences.

Corollary 5.1 (of [40, Theorem 5.1]) *Let $X \in \Sigma^m$, $k \in [1\mathinner{.\,.} m]$, and $Y \in \Sigma^n$ with $n \leq 2m$. If X is not k-periodic, then $|\mathsf{occ}^E_k(X, Y)| = \mathcal{O}(k^2)$.*

In particular, if P_{i-1} is not k-periodic, then it has $\mathcal{O}(k^2)$ active k-edit occurrences. For each active occurrence $p \in \mathsf{occ}^E_k(P_{i-1}, T)$, we maintain an edit-distance sketch $\mathsf{sk}^E_k(T(p\mathinner{.\,.} r])$ and combine it with a sketch $\mathsf{sk}^E_k(P(\ell_{i-1}\mathinner{.\,.}\ell_i])$ (constructed at preprocessing) in order to derive $ed(T(p\mathinner{.\,.} r], P(\ell_{i-1}\mathinner{.\,.}\ell_i])$ or certify that this distance exceeds k. Since we have stored the smallest $k' \in [0\mathinner{.\,.} k]$ such that $p \in \mathsf{occ}^E_{k'}(P, T)$, this lets us check whether any k-edit occurrence of P_{i-1} ending at position p extends to a k-edit occurrence of P_i ending at position r. With previously known k-edit sketches [20, 90], this already yields an $\mathrm{poly}(k \log n)$-space implementation in this case.

The difficulty lies in k-periodic strings whose occurrences form *chains*.

Definition 5 (Chain of occurrences) *Consider strings $X, Y \in \Sigma^*$ and an integer $k \in \mathbb{Z}_{\geq 0}$. An increasing sequence of positions $p_1, \ldots, p_c$ forms a chain of k-edit occurrences of X in Y if:*

1. *There is a difference string $D \in \Sigma^*$ such that $D = Y(p_j\mathinner{.\,.} p_{j+1}]$ for $j \in [1\mathinner{.\,.} c)$;*
2. *There is an integer $k' \in [0\mathinner{.\,.} k]$ such that $p_j \in \mathsf{occ}^E_{k'}(X, Y) \setminus \mathsf{occ}^E_{k'-1}(X, Y)$ for $j \in [1\mathinner{.\,.} c]$.*

Corollary 5.2 (of [40, Theorem 5.2, Claim 5.16, Claim 5.17]) *Let $X \in \Sigma^m$, $k \in [1\mathinner{.\,.} m]$, and $Y \in \Sigma^n$ with $n \leq 2m$. If X is k-periodic with period Q, then $\mathsf{occ}^E_k(X, Y)$ can be decomposed into $\mathcal{O}(k^3)$ chains whose differences are of the form $\mathsf{rot}^s(Q)$ with $|m - s| \leq 10k$.*

In the following discussion, assume that P_{i-1} is k-periodic with period Q_{i-1}. Compared to the previous algorithm, we cannot afford maintaining $\mathsf{sk}^E_k(T(p\mathinner{.\,.} r])$

for all active $p \in \mathsf{occ}_k^E(P,T)$. If sk_k^E were a concatenatable sketch (like the k-mismatch sketches of [48]), we would compute $\mathsf{sk}_k^E(D)$ at preprocessing time for all $\mathcal{O}(k)$ feasible chain differences D and then, for any two subsequent positions p_j, p_{j+1} in a chain with difference D, we could use $\mathsf{sk}_k^E(D) = \mathsf{sk}_k^E(T(p_j \mathinner{.\,.} p_{j+1}])$ to transform $\mathsf{sk}_k^E(T(p_j \mathinner{.\,.} r])$ into $\mathsf{sk}_k^E(T(p_{j+1} \mathinner{.\,.} r])$. However, despite extensive research, no such edit-distance sketch is known, which remains the main obstacle in designing streaming algorithms for k-edit pattern matching.

Our workaround relies on a novel *encoding* $\mathsf{qGR}(X,Y)$ that, for a pair of strings $X, Y \in \Sigma^*$, represents a large class of low-distance edit distance alignments between X, Y. In the preprocessing phase of our algorithm, we build, for every feasible chain difference D, $\mathsf{qGR}(P(\ell_{i-1} \mathinner{.\,.} \ell_i], D^\infty[1 \mathinner{.\,.} \ell_i - \ell_{i-1}])$. In the main phase, for subsequent positions $p_j \in \mathsf{occ}_k^E(P_{i-1}, T)$ in a chain with difference D, we aim to build $\mathsf{qGR}(T(p_j \mathinner{.\,.} r], D^\infty[1 \mathinner{.\,.} \ell_i - \ell_{i-1}])$ when necessary, i.e., $|r - p_j - (\ell_i - \ell_{i-1})| \le k$. We then combine the two encodings to derive $\mathsf{qGR}(P(\ell_{i-1} \mathinner{.\,.} \ell_i], T(p_j \mathinner{.\,.} r])$ and $ed(P(\ell_{i-1} \mathinner{.\,.} \ell_i], T(p_j \mathinner{.\,.} r])$. Except for such *products* (transitive compositions), our encoding supports *concatenations*, i.e., $\mathsf{qGR}(X_1, Y_1)$ and $\mathsf{qGR}(X_2, Y_2)$ can be combined into $\mathsf{qGR}(X_1X_2, Y_1Y_2)$. Consequently, it suffices to maintain an encoding $\mathsf{qGR}(T(p_c \mathinner{.\,.} r], D^\infty[1 \mathinner{.\,.} r - p_c - k])$ (where p_c is the rightmost element of the chain). When necessary, we prepend $(j - c)$ copies of $\mathsf{qGR}(D, D)$ (merged by doubling) and append $\mathsf{qGR}(\varepsilon, D^\infty(r - p_j - k \mathinner{.\,.} \ell_i - \ell_{i-1}])$ to derive $\mathsf{qGR}(P(\ell_{i-1} \mathinner{.\,.} \ell_i], D^\infty[1 \mathinner{.\,.} \ell_i - \ell_{i-1}])$.

In the semi-streaming setting, we extend $\mathsf{qGR}(T(p_c \mathinner{.\,.} r], D^\infty[1 \mathinner{.\,.} r - p_c - k])$ one character at a time using read-only random access to D^∞. In the streaming setting, we cannot afford storing D, so we append the entire difference D in a single step and utilize a new edit-distance sketch sk^{q} that allows retrieving $\mathsf{qGR}(T(r - |D| \mathinner{.\,.} r], D)$ from $\mathsf{sk}^{\mathsf{q}}(T(r - |D| \mathinner{.\,.} r])$ and $\mathsf{sk}^{\mathsf{q}}(D)$. The sketch $\mathsf{sk}^{\mathsf{q}}(D)$ is constructed in the preprocessing phase, whereas $\mathsf{sk}^{\mathsf{q}}(T(r - |D| \mathinner{.\,.} r])$ is built as the algorithm scans T. Similarly, we can (temporarily) append any of the $\mathcal{O}(k)$ prefixes of D that may arise when $\mathsf{qGR}(T(p_c \mathinner{.\,.} r], D^\infty[1 \mathinner{.\,.} r - p_c - k])$ is necessary. Below, we outline the ideas behind our two main conceptual and technical contributions: the encoding $\mathsf{qGR}(\cdot,\cdot)$ and the sketch $\mathsf{sk}^{\mathsf{q}}(\cdot)$.

5.1.1 Greedy Alignments and Encodings

Recall that the encoding $\mathsf{qGR}(\cdot,\cdot)$ needs to support the following three operations:

1. **Capped edit distance:** given $\mathsf{qGR}(X,Y)$, compute $ed(X,Y)$ or certify that $ed(X,Y)$ is large;
2. **Product:** given $\mathsf{qGR}(X,Y)$ and $\mathsf{qGR}(Y,Z)$, retrieve $\mathsf{qGR}(X,Z)$;
3. **Concatenation:** given $\mathsf{qGR}(X_1,Y_1)$, $\mathsf{qGR}(X_2,Y_2)$, retrieve $\mathsf{qGR}(X_1X_2, Y_1Y_2)$.

Our encoding is parameterized with a threshold $k \in \mathbb{Z}_+$ such that $ed(\cdot,\cdot) > k$ is considered large, and the goal is to achieve $\tilde{\mathcal{O}}(k^{\mathcal{O}(1)})$ encoding size. In fact, whenever $ed(X,Y) > k$, we shall simply assume that $\mathsf{qGR}_k(X,Y)$ is undefined (formally,

$\mathsf{qGR}_k(X,Y) = \bot$). Consequently, products and concatenations will require sufficiently large thresholds in the input encodings so that if either of them is undefined, the output encoding is also undefined.

In order to support concatenations alone, we could use so-called *semi-local* edit distances. For now, suppose that we only need to encode pairs of equal-length strings.[1] Through a sequence of concatenations, we may only extend $\mathsf{qGR}_k(X,Y)$ to $\mathsf{qGR}_k(X',Y')$ so that $X = X'[\ell\,..\,r)$ and $Y = Y'[\ell\,..\,r)$. For any alignment $\mathcal{A}'$ of X', Y' with $\mathsf{cost}(\mathcal{A}') \le k$, consider the induced alignment $\mathcal{A} := \mathcal{A}'_{[\ell..r),[\ell..r)}$. Note that $\mathcal{A}$ mimics the behaviour of $\mathcal{A}'$ except that it deletes some characters at the extremes of X and Y (which $\mathcal{A}'$ aligns outside Y and X, respectively). By Fact 3.1, we have $\mathsf{cost}(\mathcal{A}) \le k$ and, in particular, $\mathcal{A}$ deletes at most k characters at the extremes of X and Y. If, after performing these deletions, we replace $\mathcal{A}$ with an optimal alignment between the remaining fragments of X and Y, this modification may only decrease $\mathsf{cost}(\mathcal{A})$ and $\mathsf{cost}(\mathcal{A}')$. Consequently, it suffices to store the $\mathcal{O}(k)$ characters at the extremes of X, Y and the $\mathcal{O}(k^4)$ edit distances between long fragments of X and Y. In a sense, this encoding represents $\mathcal{O}(k^4)$ alignments between X, Y that are sufficient to derive an optimal alignment of any extension.

The main challenge is to handle products, for which we develop a *greedy encoding* $\mathsf{GR}_k(X,Y)$ that compactly represents the following family $\mathsf{GA}_k(X,Y)$ of *greedy alignments* of X, Y.

Definition 6 (Greedy alignment) *We say that an alignment $\mathcal{A}$ of two strings $X, Y \in \Sigma^*$ is greedy if $X[x] \ne Y[y]$ holds for every $(x,y) \in \mathcal{B}(\mathcal{A}) \cap ([1\,..\,|X|] \times [1\,..\,|Y|])$. Given $k \ge 0$, we denote by $\mathsf{GA}_k(X,Y)$ the set of all greedy alignments $\mathcal{A}$ of X, Y satisfying $\mathsf{cost}(\mathcal{A}) \le k$.*

Intuitively, whenever a greedy alignment encounters a pair of matching characters $X[x]$ and $Y[y]$, it must (greedily) match these characters (it cannot delete $X[x]$ or $Y[y]$). As stated below, this restriction does not affect the optimal cost.

Fact 5.1 *For two strings $X, Y \in \Sigma^*$, there is an optimal greedy alignment of X, Y.*

For strings $X, Y \in \Sigma^*$ and an integer $k \ge ed(X,Y)$, we define a set $\mathcal{M}_k(X,Y)$ of *common matches* of all alignments $\mathcal{A} \in \mathsf{GA}_k(X,Y)$; formally $\mathcal{M}_k(X,Y) = \bigcap_{\mathcal{A}\in\mathsf{GA}_k(X,Y)} \mathcal{M}_{X,Y}(\mathcal{A})$. In our greedy encoding, we shall mask out all the characters involved in the common matches. Below, this transformation is defined for an arbitrary non-crossing matching of X, Y.

Definition 7 *Let M be a non-crossing matching of strings $X, Y \in \Sigma^*$. We define X^M, Y^M to be the strings obtained from X, Y by replacing $X[x]$ and $Y[y]$ with $\# \notin \Sigma$ for every $(x,y) \in M$. We refer to $\#$ as a dummy symbol and to maximal blocks of $\#$'s as dummy segments.*

[1] Pairs of strings of any lengths can be supported in the same way provided that concatenations require larger input thresholds (compared to the output threshold) to accommodate length differences.

The following lemma proves that masking out common matches does not affect $ed(X, Y)$ or $\mathcal{M}_k(X, Y)$ provided that we *enumerate* the dummy symbols, that is, any string Z is transformed to $\mathsf{num}(Z)$ by replacing the ith leftmost occurrence of $\#$ with a unique symbol $\#_i \notin \Sigma$.

Lemma 5.1 *Consider strings $X, Y \in \Sigma^*$, an integer $k \geq ed(X, Y)$, and a set $M \subseteq \mathcal{M}_k(X, Y)$. Then, $ed(X, Y) = ed(\mathsf{num}(X^M), \mathsf{num}(Y^M))$ and $\mathcal{M}_k(X, Y) = \mathcal{M}_k(\mathsf{num}(X^M), \mathsf{num}(Y^M))$.*

At the same time, after masking out the common matches, the strings become compressible. Intuitively, this is because once two greedy alignments converge, they stay together until they encounter a mismatch. Moreover, when two alignments proceed in parallel without any mismatch, this incurs a small period (at most $2k$) that is captured by the LZ factorization.

Lemma 5.2 *Let $M = \mathcal{M}_k(X, Y)$ for strings $X, Y \in \Sigma^*$ and a positive integer $k \geq ed(X, Y)$. Then, $|\mathsf{LZ}(X^M)|, |\mathsf{LZ}(Y^M)| = \mathcal{O}(k^2)$, and X^M, Y^M contain $\mathcal{O}(k)$ dummy segments.*

Consequently, for $k \geq ed(X, Y)$, we could define the *greedy encoding* $\mathsf{GR}_k(X, Y)$ so that it consists of $\mathsf{LZ}(X^M)$ and $\mathsf{LZ}(Y^M)$. Instead, we use a more powerful compressed representation that supports more efficient queries concerning X^M and Y^M.

Even though $\mathsf{GR}_k(X, Y)$ is small, $\mathsf{GA}_k(X, Y)$ may consist of $2^{\Theta(k)}$ alignments, which is why constructing $\mathsf{GR}_k(X, Y)$ in poly(k) time is far from trivial. The following combinatorial lemma lets us obtain an $\mathcal{O}(k^5)$-time algorithm. Intuitively, the alignments in $\mathsf{GA}_k(X, Y)$ can be interpreted as paths in a directed acyclic graph with $\mathcal{O}(k^5)$ branching vertices.

Lemma 5.3 *For all $X, Y \in \Sigma^*$ and $k \in \mathbb{Z}_+$, the set $\mathcal{B}_k(X, Y) = \bigcup_{\mathcal{A} \in \mathsf{GA}_k(X,Y)} \mathcal{B}(\mathcal{A})$ is of size $\mathcal{O}(k^5)$.*

The reason why $\mathsf{GR}_k(X, Y)$ supports products is that every greedy alignment of X, Z can be interpreted as a product of a greedy alignment of X, Y and a greedy alignment of Y, Z.

Definition 8 *Consider strings $X, Y, Z \in \Sigma^*$, an alignment $\mathcal{A}^{X,Y}$ of X, Y, an alignment $\mathcal{A}^{Y,Z}$ of Y, Z, and an alignment $\mathcal{A}^{X,Z}$ of X, Z. We say that $\mathcal{A}^{X,Z}$ is a product of $\mathcal{A}^{X,Y}$ and $\mathcal{A}^{Y,Z}$ if, for every $(x, z) \in \mathcal{A}^{X,Z}$, there is $y \in [1\,.\,.\,|Y| + 1]$ such that $(x, y) \in \mathcal{A}^{X,Y}$ and $(y, z) \in \mathcal{A}^{Y,Z}$.*

Lemma 5.4 *Consider strings $X, Y, Z \in \Sigma^*$ and $k \in \mathbb{Z}_{\geq 0}$. Every alignment $\mathcal{A}^{X,Z} \in \mathsf{GA}_k(X, Z)$ is a product of alignments $\mathcal{A}^{X,Y} \in \mathsf{GA}_d(X, Y)$ and $\mathcal{A}^{Y,Z} \in \mathsf{GA}_d(Y, Z)$, where $d = 2k + ed(X, Y)$.*

Hence, $\mathsf{GR}_d(X,Y)$ and $\mathsf{GR}_d(Y,Z)$ contain enough information to derive $\mathsf{GR}_k(X,Z)$. The underlying algorithm propagates the characters of Y stored in $\mathsf{GR}_d(X,Y)$ and $\mathsf{GR}_d(Y,Z)$ along the matchings $\mathcal{M}_d(Y,Z)$ and $\mathcal{M}_d(X,Y)$, respectively. Then, $\mathsf{GR}_k(X,Z)$ is obtained by masking out all the characters corresponding to $\mathcal{M}_k(X,Y)$.

To support concatenations, we extend the family $\mathsf{GA}_k(X,Y)$ of greedy alignments to a family $\mathsf{qGA}_k(X,Y)$ of *quasi-greedy alignments*, which are allowed to delete a prefix of X or Y in violation of Definition 6. The *quasi-greedy encoding* $\mathsf{qGR}_k(X,Y)$ is defined analogously to $\mathsf{GR}_k(X,Y)$. Equivalently, $\mathsf{qGA}_k(X,Y)$ can be derived from $\mathsf{GA}_{k+1}(\$_1X,\$_2Y)$, where $\$_1 \neq \$_2$ are sentinel symbols outside Σ, by taking the alignments induced by X,Y. The latter characterization makes all our claims regarding GA and GR easily portable to qGA and qGR. In particular, this is true for the sketches, described in the following subsection for GR only.

5.1.2 Edit Distance Sketches

Recall that we need an edit-distance sketch sk^E allowing to retrieve $\mathsf{GR}_k(X,Y)$ from $\mathsf{sk}^E_k(X)$ and $\mathsf{sk}^E_k(Y)$ for any strings $X,Y \in \Sigma^{\le n}$ and any threshold $k \in [0\mathinner{.\,.} n]$. Furthermore, we need to make sure that $\mathsf{sk}^E_k(S)$ can be computed given streaming access to $S \in \Sigma^{\le n}$, and that the encoding and decoding procedures use $\mathrm{poly}(k,\log n)$ space. We devise a novel $\tilde{\mathcal{O}}(k^2)$-size sketch specifically designed to output $\mathsf{qGR}_k(X,Y)$. We note that the $\tilde{\mathcal{O}}(k^2)$ size is optimal for $\mathsf{qGR}_k(X,Y)$, but we are not aware of a matching lower bound for retrieving $ed(X,Y)$ capped with k.

In a manner similar to the sketches of [20, 90], ours relies on the embedding of Chakraborty, Goldenberg, and Koucký [36]. The CGK algorithm performs a random walk over the input string (with forward and stationary steps only). In abstract terms, such walk can be specified as follows:

Definition 9 (Complete walk) *For a string $S \in \Sigma^*$, we say that $(s_t)_{t=1}^{m+1}$ is an m-step complete walk over S if $s_1 = 1$, $s_{m+1} = |S|+1$, and $s_{t+1} \in \{s_t, s_t+1\}$ for $t \in [1\mathinner{.\,.} m]$.*

For any two strings $X,Y \in \Sigma^*$, the two walks underlying $\mathsf{CGK}(X)$ and $\mathsf{CGK}(Y)$ can be interpreted as an edit distance alignment using the following abstract definition:

Definition 10 *The zip alignment of m-step complete walks $(x_t)_{t=1}^{m+1}$ and $(y_t)_{t=1}^{m+1}$ over $X,Y \in \Sigma^*$ is obtained by removing repeated entries in $(x_t,y_t)_{t=1}^{m+1}$.*

The key result of [36] is that the cost of the zip alignment of CGK walks over $X,Y \in \Sigma^*$ is $\mathcal{O}(ed(X,Y)^2)$ with good probability, which is then exploited to derive a metric embedding (mapping edit distance to Hamming distance) with quadratic distortion. In our sketch, we also need to observe that the CGK alignment is greedy and that its width is $\mathcal{O}(ed(X,Y))$ with good probability. The following lemma provides a complete black-box interface of the properties of the CGK algorithm utilized in our sketches. It also encapsulates Nisan's pseudorandom generator [117] that reduces the number of (shared) random bits.

Lemma 5.5 *For every constant $\delta \in (0,1)$, there exists a constant c and an algorithm W that, given an integer n, a seed r of $\mathcal{O}(\log^2 n)$ random bits, and a string $S \in \Sigma^{\le n}$, outputs a $3n$-step complete walk $\mathsf{W}(n,r,S)$ over S satisfying the following property for all $X, Y \in \Sigma^{\le n}$ and the zip alignment $\mathcal{A}_\mathsf{W}$ of $\mathsf{W}(n,r,X)$ and $\mathsf{W}(n,r,Y)$:*

$$\Pr_r\begin{bmatrix}\mathcal{A}_\mathsf{W} \in \mathsf{GA}_{c\cdot ed(X,Y)^2}(X,Y)\\ \textit{and}\\ \mathsf{width}(\mathcal{A}_\mathsf{W}) \le c \cdot ed(X,Y)\end{bmatrix} \ge 1-\delta.$$

Moreover, W is an $\mathcal{O}(\log^2 n)$-bit streaming algorithm that costs $\mathcal{O}(n \log n)$ time and reports any element $s_t \in [1\,..\,|S|]$ of $\mathsf{W}(n,r,S)$ while processing the corresponding character $S[s_t]$.

Next, we analyze the structural similarity between $\mathcal{A}_\mathsf{W}$ and any alignment $\mathcal{A} \in \mathsf{GA}_k(X,Y)$. Based on Lemma 5.5, we may assume that $\mathcal{A}_\mathsf{W} \in \mathsf{GA}_{\mathcal{O}(k^2)}(X,Y)$ and $\mathsf{width}(\mathcal{A}_\mathsf{W}) = \mathcal{O}(k)$. Consider the set $M = \mathcal{M}(\mathcal{A}) \cap \mathcal{M}(\mathcal{A}_\mathsf{W})$ of the common matches of $\mathcal{A}$ and $\mathcal{A}_\mathsf{W}$ and the string X^M obtained by masking out the underlying characters of X. Whereas Lemma 5.2 immediately implies that the LZ factorization of X^M consists of $\mathcal{O}(k^4)$ phrases, a more careful application of the same technique provides a refined bound of $\mathcal{O}(k^2)$ phrases. Furthermore, there are $\mathcal{O}(k)$ dummy segments in X^M and, if $X[x]$ is not masked out in X^M (for some $x \in [1\,..\,|X|]$), then there is a breakpoint $(x', y') \in \mathcal{B}_{X,Y}(\mathcal{A}_\mathsf{W})$ with $x' \in [1\,..\,x]$ and $|\mathsf{LZ}(X[x'\,..\,x])| = \mathcal{O}(k)$. Intuitively, this means that $\mathcal{A}$ and $\mathcal{A}_\mathsf{W}$ diverge only within highly compressible regions following the breakpoints $\mathcal{B}_{X,Y}(\mathcal{A}_\mathsf{W})$. We call these regions *forward contexts* (formally, a forward context is the longest fragment starting at a given position and satisfying certain compressibility condition). Since our choice of $\mathcal{A} \in \mathsf{GA}_k(X,Y)$ was arbitrary, any two alignments $\mathcal{A}, \mathcal{A}' \in \mathsf{GA}_k(X,Y)$ diverge only within these forward contexts. Hence, in order to reconstruct $\mathsf{GR}_k(X,Y)$ and, in particular, $X^{\mathcal{M}_k(X,Y)}$, the sketch should be powerful enough to retrieve all characters in forward contexts of breakpoints $\mathcal{B}_{X,Y}(\mathcal{A}_\mathsf{W})$. Even though $\mathcal{B}_{X,Y}(\mathcal{A}_\mathsf{W})$ could be of size $\Theta(k^2)$, due to the aforementioned bounds on $|\mathsf{LZ}(X^M)|$ and the number of dummy segments in X^M, it suffices to take $\mathcal{O}(k)$ among these forward contexts to cover the unmasked regions of X^M and $X^{\mathcal{M}_k(X,Y)}$. Each context can be encoded in $\tilde{\mathcal{O}}(k)$ bits, so this paves a way towards sketches of size $\tilde{\mathcal{O}}(k^2)$.

Nevertheless, while processing a string $X \in \Sigma^{\le n}$, we only have access to the string X and the m-complete walk $(x_t)_{t=1}^m = \mathsf{W}(n,r,X)$ over X. In particular, depending on Y, any position in X could be involved in a breakpoint. A naive strategy would be to build a *context encoding* $\mathsf{CE}(X)[1\,..\,m]$ that stores at $t \in [1\,..\,m]$ (a compressed representation of) the forward context starting at $X[x_t]$, and then post-process it using a Hamming-distance sketch. This is sufficiently powerful because $X[x_t] \ne Y[y_t]$ holds for any $(x_t, y_t) \in \mathcal{B}(\mathcal{A}_\mathsf{W})$ (recall that $\mathcal{A}_\mathsf{W}$ is greedy). Unfortunately, this construction does not guarantee any upper bound on $\mathsf{hd}(\mathsf{CE}(X), \mathsf{CE}(Y))$ in terms of k. (For example, if X is compressible, modifying its final character affects the entire $\mathsf{CE}(X)$.) Hence, we sparsify $\mathsf{CE}(X)$ by placing a blank symbol $\perp$

at some positions $\mathsf{CE}(X)[t]$ so that just a few forward contexts stored in $\mathsf{CE}(X)[t]$ cover any single position in X.

This brings two further challenges. First, if $X[x]$ is involved in a breakpoint, then we are only guaranteed that it is covered by some forward context $X[p\mathinner{.\,.} q)$ of $\mathsf{CE}(X)[t]$ (i.e., $x \in [p\mathinner{.\,.} q)$). In particular, the forward context starting at position x could extend beyond $X[x\mathinner{.\,.} q)$. Hence, the string $\mathsf{CE}(X)[t]$ actually stores *double forward contexts* $X[p\mathinner{.\,.} r) = X[p\mathinner{.\,.} q)X[q\mathinner{.\,.} r)$ defined as the concatenation of the forward contexts of $X[p]$ and $X[q]$. We expect this double forward context $X[p\mathinner{.\,.} r)$ to cover the entire forward context of $X[x]$. Unfortunately, this is not necessarily true if we use the Lempel–Ziv factorization to quantify compressibility: we could have $|\mathsf{LZ}(X[x\mathinner{.\,.} r))| < |\mathsf{LZ}(X[q\mathinner{.\,.} r))|$ because $\mathsf{LZ}(\cdot)$ is not monotone. Instead, we use an ad-hoc compressibility measure defined as $\mathsf{maxLZ}(S) = \max_{[\ell\mathinner{.\,.} r)\subseteq[1\mathinner{.\,.}|S|]} \mathsf{LZ}(\mathsf{rev}(S[\ell\mathinner{.\,.} r)))$. Maximization over substrings guarantees monotonicity, whereas reversal helps designing an efficient streaming algorithm constructing contexts (beyond the scope of this overview).

Another challenge is that the sparsification needs to be consistent between $\mathsf{CE}(X)$ and $\mathsf{CE}(Y)$: assuming $ed(X,Y) \le k$, we should have $\mathsf{hd}(\mathsf{CE}(X), \mathsf{CE}(Y)) = \tilde{\mathcal{O}}(k)$, which also accounts for mismatches between $\perp$ and a stored double forward context. This rules out a naive strategy of covering X from left to right using disjoint forward contexts: any substitution at the beginning of X could then have a cascade of consequences throughout $\mathsf{CE}(X)$. Hence, we opt for a memory-less strategy that decides on $\mathsf{CE}(X)[t]$ purely based on the forward contexts $X[x_{t-1}\mathinner{.\,.} x'_{t-1})$ and $X[x_t\mathinner{.\,.} x'_t)$. For example, we could set $\mathsf{CE}(X)[t] = \perp$ unless the smallest dyadic interval containing $[x_{t-1}\mathinner{.\,.} x'_{t-1})$ differs from the smallest dyadic interval containing $[x_t\mathinner{.\,.} x'_t)$ (a dyadic interval is of the form $[i2^j\mathinner{.\,.}(i+1)2^j)$ for some integers $i, j \ge 0$). With this approach, each position of X is covered by at least one and at most $\mathcal{O}(\log|X|)$ forward contexts. Furthermore, substituting any character in X does not have far-reaching knock-on effects. Unfortunately, insertions and deletions are still problematic as they shift the positions x_t. Thus, the decision concerning $\mathsf{CE}(X)[t]$ should be independent of the numerical value of x_t. Consequently, instead of looking at the smallest dyadic interval containing $[x_t\mathinner{.\,.} x'_t)$, we choose the largest t' such that $[x_t\mathinner{.\,.} x'_t) = [x_t\mathinner{.\,.} x_{t'})$, and we look at the smallest dyadic interval containing $[t\mathinner{.\,.} t')$.

With each forward context $X[x_t\mathinner{.\,.} x'_t)$ retrieved, we also need to determine the value x_t (so that we know which fragment of X we can learn from $\mathsf{CE}(X)[t]$). To avoid knock-on effects, we actually store differences $x_t - x_u$ with respect to the previous index satisfying $\mathsf{CE}(X)[u] \ne \perp$. This completes the intuitive description of the context encoding CE.

Our edit-distance sketch contains the Hamming-distance sketch of $\mathsf{CE}(X)$. For this, we use an existing construction [48], augmented in a black-box manner to support large alphabets (recall that the each forward contexts takes $\tilde{\mathcal{O}}(k)$ bits). Furthermore, to retrieve the starting positions x_t (rather than just the differences $x_t - x_u$), we use a hierarchical Hamming-distance sketch similar to those used in [20, 90]. This way, given sketches of X and Y, we can recover all characters that remain unmasked in $X^{\mathcal{M}_k(X,Y)}$ and $Y^{\mathcal{M}_k(X,Y)}$. The tools developed for greedy encodings are then used to compute $ed(X,Y)$ (or certify $ed(X,Y) > k$) and to retrieve the greedy encoding $\mathsf{GR}_k(X,Y)$. We summarize the properties of the edit-distance sketches below:

Theorem 5.1 *For every constant $\delta \in (0,1)$, there is a sketch sk^E_k (parametrized by integers $n \geq k \geq 1$, an alphabet $\Sigma = [0\,..\,n^{\mathcal{O}(1)})$, and a seed of $\mathcal{O}(\log^2 n)$ random bits) such that:*

1. *The sketch $\mathsf{sk}^E_k(S)$ of a string $\Sigma^{\leq n}$ takes $\mathcal{O}(k^2 \log^3 n)$ bits. Given streaming access to S, it can be constructed in $\tilde{\mathcal{O}}(nk)$ time using $\tilde{\mathcal{O}}(k^2)$ space.*
2. *There exists an $\tilde{\mathcal{O}}(k^2)$-space decoding algorithm that, given $\mathsf{sk}^E_k(X), \mathsf{sk}^E_k(Y)$ for $X, Y \in \Sigma^{\leq n}$, with probability $\geq 1-\delta$ outputs $\mathsf{GR}_k(X,Y)$ and $\min(ed(X,Y), k+1)$. Retrieving $\mathsf{GR}_k(X,Y)$ costs $\tilde{\mathcal{O}}(k^5)$ time, and computing $\min(ed(X,Y), k+1)$ costs $\tilde{\mathcal{O}}(k^3)$ time.*

6 Lower bounds for approximate text indexing

"There are no solutions; there are only trade-offs."
Ursula K. Le Guin, *The Dispossessed*

In this section, we present an overview of the lower bounds for approximate dictionary look-up and text indexing we developed in [51].

6.1 Statements of the problems and existing lower bounds

Let us start with the precise statements of the problems. Many of them have been considered both in the field of algorithms on strings and in the field of computational geometry, and sometimes they are known under different names. We try, where possible, to provide alternative names.

The starting point of our work is approximate text indexing, which can be formally stated as follows (here, we focus on the case of the Hamming distance only):

Problem 6.1 ***Text indexing with k mismatches***
Input: An alphabet Σ, a string $T \in \Sigma^n$ (referred to as text), an integer k.
Output: A data structure (referred to as text index) that maintains the following queries: Given a string $Q \in \Sigma^d$ (referred to as pattern), output a substring / all substrings of T that are within Hamming distance k from Q.

This problem is closely related to the problem of approximate dictionary look-up introduced by Minsky and Papert in 1968 [110]. The problem is also known under the name of *k-neighbour*.

Problem 6.2 ***Dictionary look-up with k mismatches (edits)***
Input: An alphabet Σ, a set of strings in Σ^d of total length n (dictionary).
Output: A data structure that given a string $Q \in \Sigma^d$, outputs all strings in the dictionary that are at Hamming (edit) distance $\leq k$ from Q.

We start by showing conditional lower bounds for approximate dictionary look-up via a reduction from bichromatic closest pair.

Problem 6.3 ***Bichromatic closest pair with mismatches (differences)***
Input: A set of n red strings in $\{0,1\}^m$ and a set of n blue strings in $\{0,1\}^m$.
Output: A pair of a red string S_r and a blue string S_b, such that the Hamming (edit) distance between S_r and S_b is minimized.

Alman and Williams [10] showed that if there is $\delta > 0$ such that for all constant $c > 0$, the bichromatic closest pair with mismatches problem, with $m = c \log n$, can be solved in $\mathcal{O}(n^{2-\delta})$ time, then SETH is false. By a standard reduction, this implies that no algorithm can process a dictionary of n strings of length m in polynomial time, and subsequently answer dictionary look-up queries with $k = \Theta(m)$ mismatches in $\mathcal{O}(n^{1-\delta})$ time.

Another related problem is *dictionary look-up with k don't cares*. In this problem, the query strings may contain up to k don't care symbols, that is, special characters that match any character of the alphabet, and the task is to retrieve all dictionary strings that match the query string. Essentially, it is a parameterized variant of the *partial match* problem (see [121] and references therein). The structure of this problem is very similar to that of approximate dictionary look-up, as each don't care symbol in the query string gives $|\Sigma|$ possibilities for the corresponding symbol in the dictionary strings. On the other hand, it is simpler, as the positions of the don't care symbols are fixed. However, even for this simpler problem all known solutions have an exponential dependency on k, either in the space complexity or in the query time (see [105] and references therein). Afshani and Nielsen [8] showed that in general one cannot avoid this. In more detail, they showed that for $3\sqrt{\log n} \le k = o(\log n)$, any pointer-machine data structure with query time $\mathcal{O}(2^{k/2} + m + occ)$ requires $\Omega(n^{1+\Theta(1/\log k)})$ space, even for the binary alphabet.

6.2 Embedding from Hamming to edit distance: stoppers transform

We start by introducing a deterministic algorithm that we refer to as *stoppers transform*. The stoppers transform receives a set of binary strings of a fixed length m and outputs a set of strings of length $m' = \mathcal{O}(m \log m)$ such that the edit distance between the transformed strings is equal to the Hamming distance between the original strings.

We will apply the transform to binary strings of a fixed length m. We assume that m is power of two, $m = 2^q$ for some integer q. If this is not the case, we append the strings with an appropriate number of zeros, which does not change the Hamming distance between them and increases the length by a factor of at most two.

We fix q characters $c_1, \ldots, c_q$ that do not belong to the binary alphabet $\{0,1\}$. For each $i = 1, 2, \ldots, q$, we define a string $S_i = \underbrace{c_i c_i \ldots c_i}_{6 \cdot 2^i}$. We call S_i a *stopper*.

Let X be a string $\in \{0,1\}^d$, where $d = 2^q$. First, we insert a stopper S_q after the first 2^{q-1} symbols of X. Second, we apply the transform recursively to the strings consisting of the first 2^{q-1} and the last 2^{q-1} symbols of X. We summarize the transform in Algorithm 2.

Algorithm 2 StoppersTransform(X)

```
1: if $q = 0$ then
2:     return $X$
3: else
4:     $X_1 = $ StoppersTransform($X[1, 2^{q-1}]$)
5:     $X_2 = $ StoppersTransform($X[2^{q-1}+1, 2^q]$)
6:     $X = X_1 S_q X_2$
7:     return $X$
```

Intuitively, when the stoppers transform is applied to two equal-length strings X, Y, the stoppers prevent an optimal edit distance alignment to match a character $X[i]$ with a character $Y[j]$, $i \neq j$. This obstacle is created on q levels: For example, when we insert the 0-level stopper S_0 we make it disadvantageous to align $X[i]$ with a character $Y[j]$ such that $i \leq 2^{q-1}$ and $j > 2^{q-1}$, and vice versa. As a consequence, the edit distance between the transformed strings equals the Hamming distance between the original strings.

Thanks to the stoppers transform, it suffices to analyse the lower bounds the Hamming distance, and the lower bounds for the edit distance follow almost automatically. This is a big advantage, as the edit distance is typically much harder to analyse than the Hamming distance. We note that Rubinstein [125] used a similar approach to derive a lower bound for bichromatic closest pair with edits from a lower bound for bichromatic closest pair with mismatches, however his algorithm was randomised and preserved the distances only approximately.

6.3 Conditional lower bounds for dictionary look-up

The first part of our lower bounds is lower bounds for the RAM model conditional on SETH. Namely, we apply the stoppers transform to derive conditional lower bounds for bichromatic closest pair with edits and dictionary look-up with edits from a conditional lower bound for bichromatic closest pair with mismatches [10]. Namely, we show that the bichromatic closest pair with edits cannot be solved in strongly subquadratic time and that any data structure for dictionary look-up with edits that can be constructed in polynomial time cannot have strongly sublinear query time. We note that similar lower bounds can be obtained from the lower bound of Rubinstein for the $(1+\varepsilon)$-approximate bichromatic closest pair, however our proof is simpler as it uses only a series of combinatorial reductions instead of the complex distributed PCP framework. These bounds are tight within polylogarithmic factors since bichromatic closest pair can be solved in $\mathcal{O}(n^2m^2)$ time and dictionary look-up with edits can be solved in $\mathcal{O}(nm^2)$ time. On the other hand, these lower bounds only hold for $k = \Theta(\log n)$.

6.4 Pointer-machine lower bounds for dictionary look-up

The pointer machine model focuses on the data structures that solely use pointers to access memory locations, i.e. no address calculations are allowed. Similarly to many other lower bound proofs, we use the variant of this model defined in [42]. Consider a reporting problem, such as the text indexing problem for example. Let U be the universe of possible answers (in the case of text indexing, substrings of the text), and let the answer to a query Q be a subset $\mathcal{S}$ of U. We assume that the data structure is a rooted tree of constant degree, where each node represents a memory cell that can store one element of U. Edges of the tree correspond to pointers between the memory cells. The information beyond the elements of U and the pointers is not accounted for, it can be stored and accessed for free by the data structure. Given a query Q, the algorithm must find all the nodes containing the elements of $\mathcal{S}$. It always starts at the root of the tree and at each step can move from a node to its child. The number of explored nodes is a lower bound on the query time, and the size of the tree is a lower bound on the space. We note that all known data structures for approximate dictionary look-up and text indexing are pointer machine data structures, in other words, our bounds show that in order to develop more efficient solutions one would need to come up with a radically new approach.

For many applications k is relatively small, and one might hope to achieve much better bounds for this regime. We show that this is probably not the case by demonstrating a number of pointer-machine lower bounds that give a more precise dependency between the complexity and the number of mismatches (edits). We start by showing a lower bound for the problem of dictionary look-up with k mismatches.

Similar to [8], we use the framework of Afshani [7] that gives a pointer machine lower bound for a problem called *geometric stabbing* defined on the m-dimensional Hamming cube. Namely, in this problem we are given a subset $\mathcal{Q}$ of points, and a set $\mathcal{R}$ of geometric regions in the cube. We must preprocess $\mathcal{R}$ into a data structure, to support the following queries: given a point $P \in \mathcal{Q}$, output all the regions that contain P. [7, Theorem 1] gives the following pointer machine lower bound for this problem:

Theorem 6.1 ([7]) *Consider a set $\mathcal{R}$ of d geometric regions that satisfies the following two conditions for some parameters β, t, and v:*

- *every point in $\mathcal{Q}$ is contained in at least t regions;*
- *the volume of the intersection of every β regions is at most v.*

Then for any pointer-machine data structure, if answering geometric stabbing queries can be done in time $g(d)+\mathcal{O}(\mathtt{output})$, where g is an increasing function and $g(d) \leq t$, then the space used is $S(d) = \Omega(tv^{-1}2^{-\mathcal{O}(\beta)})$.

The first part of our proof is to construct a dictionary and a set of queries. To give some flavour of the proof, we give our definitions of the query set and of the dictionary below. The set of query strings $\mathcal{Q}$ is constructed as follows. A string $P \in \mathcal{Q} \subseteq \{0,1\}^m$ if, when we divide it into k blocks of lengths m/k, $k/2$ blocks contain exactly one set bit, and $k/2$ blocks contain exactly two set bits. Next, we select the set of strings of the dictionary $\mathcal{D}$ and the associated set of regions $\mathcal{R}$, which are simply the Hamming balls of radius k whose centres are the strings in $\mathcal{D}$. The dictionary $\mathcal{D}$ is defined probabilistically in two steps. First, we construct the set of strings in $\{0,1\}^m$ such that if we divide it into blocks of length m/k, each block contains exactly one set bit. (Note that this is similar to the shape of the query strings but is not the same.) Second, we apply a sparsification procedure. It uses two parameters, α and an associated probability p_α, and ensures that the size of $\mathcal{D}$ is relatively small, while the distance between any two strings in it is relatively large (which, intuitively, controls the volume of the intersection of the regions). The procedure first filters the set of strings defined above by selecting each string in this set with a probability p_α, and then if there are two strings at distance at most $\lfloor k(1/4-\alpha)\rfloor$ from each other, it deletes both of them. The remaining strings form the set $\mathcal{D}$.

It is not difficult to see the correspondence between the dictionary look-up problem for $\mathcal{D}$ and $\mathcal{Q}$ and the geometric stabbing problem: Reporting the set of regions in $\mathcal{R}$ a string $P \in \mathcal{Q}$ belongs to is equivalent to reporting the set of dictionary strings that are at distance at most k from this string. The main part of the proof consists in proving that $\mathcal{R}$ and $\mathcal{Q}$ satisfy the hypothesis of Theorem 6.1. Finally, we apply the theorem and translate the conclusion to our original context. Proving that the condition of the theorem is met heavily relies on precisely understanding how the Hamming distance behaves on the special instances we consider. As a consequence, we show that any data structure for dictionary look-up with k mismatches that has query time $\mathcal{O}(m+(\frac{\log n}{k})^k+occ)$ requires $\Omega(c^k n)$ space, for some constant $c>1$, for all even $\frac{8}{\sqrt{3}}\sqrt{\log n} \leq k = o(\log n)$. Applying the stoppers transform, we immediately obtain similar lower bounds for the edit distance.

6.5 Lower bounds for text indexing

Finally, we show a reduction from dictionary look-up to text indexing which gives us lower bounds for text indexing. The main idea of the reduction is quite simple: We define the text as the concatenation of the dictionary strings interleaved with a special gadget string. The gadget string must guarantee that if we align the pattern with a substring that is not in the dictionary, the Hamming distance will be much larger than k. Via this reduction, the lower bounds for approximate dictionary look-up under the Hamming distance imply that for some value of k, there is no data structure for text indexing with k mismatches with sublinear query time unless SETH is false. See a diagram of the reductions in Fig. 2.3. In the pointer machine model, the reduction implies that any data structure for text indexing with k mismatches with query time $\mathcal{O}(m+(\frac{\log n}{2k})^k+occ)$ must use $\Omega(c^k n)$ space, for some constant $c>1$. The bound holds for all even $\frac{8}{\sqrt{3}}\sqrt{\log n} \leq k = o(\log n)$.

7 Streaming algorithms for regular expressions search

"Being able to break a problem into small pieces and recombine them is one of the essential skills of design."

Barbara Liskov, *Keynote Address*

In this section, we give an overview of the main technical ideas we introduced to design a streaming algorithm for recognition of regular languages [59]. It can be seen as a (highly non-trivial) application of streaming pattern matching.

7.1 Statements of the problems

Let us start by giving the precise formulation of the problems.

Definition 11 (Regular expression) *We define regular expressions over an alphabet Σ as well as the languages they match recursively. Let $L(R)$ be the language matched by a regular expression R.*

- *Any $a \in \Sigma \cup \{\varepsilon\}$ is a regular expression and $L(a) = \{a\}$.*

For two regular expressions A and B, we can form a new expression using one of the three symbols $\cdot$ (concatenation), $|$ (union), or * (Kleene star):

- *$A \cdot B$ is a regular expression and $L(A \cdot B) = \{XY : X \in L(A) \text{ and } Y \in L(B)\}$;*
- *$A \mid B$ is a regular expression and $L(A \mid B) = L(A) \cup L(B)$;*
- *A^* is a regular expression and $L(A^*) = \bigcup_{k \geq 0} \{X_1 X_2 \ldots X_k : X_i \in L(A) \text{ for } 1 \leq i \leq k\}$.*

Definition 12 (Thompson automaton [133]) *For a regular expression R we define the Thompson automaton of R, $T(R)$, recursively. This non-deterministic finite automaton (NFA) accepts all strings $s \in L(R)$.*

- *If $R = a \in \Sigma \cup \{\varepsilon\}$, $T(R)$ is constructed as in Figure 7.1a;*

- *If $R = A \cdot B$, $T(R)$ is constructed as in Figure 7.1b. Namely, the initial state of $T(A)$ becomes the initial state of $T(R)$, the final state of $T(A)$ becomes the initial state of $T(B)$, and the final state of $T(B)$ becomes the final state of $T(R)$;*
- *If $R = A|B$, $T(R)$ is constructed as in Figure 7.1c. Namely, the initial state of $T(R)$ goes via ε-transitions both to the initial state of $T(A)$ and to the initial state of $T(B)$, and the final states of $T(A)$ and $T(B)$ go via ε-transitions to the final state of $T(R)$;*
- *If $R = A^*$, $T(R)$ is constructed as in Figure 7.1d. Namely, the initial state of $T(R)$ and the final state of $T(A)$ go via ε-transitions both to the initial state of $T(A)$, and to the final state of $T(R)$.*

Definition 13 (Compact Thompson automaton) *Given a Thompson automaton $T(R)$, we define the compact Thompson automaton $T_C(R)$ as the automaton obtained from $T(R)$ by replacing every maximal path of transitions labelled by $a_1, a_2, \ldots, a_k \in \Sigma$ by a single transition labelled by $a_1 a_2 \ldots a_k$. The non-empty labels of $T_C(R)$ are called atomic strings, and the size of the (multiset) of the atomic strings is defined to be the size of R.*

Figure 7.2 gives an example of the Thompson automata for $R = b(ab|b)^*ab$. We note that in general the size of a regular expression is much smaller than the total number of characters in it and is bounded by twice the number of union and Kleene star symbols plus two. The size of a regular expression measures its “complexity”.

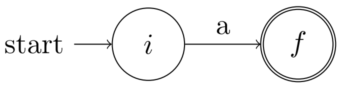
Problem 7.1 ***Regular expression membership and pattern matching***
Input: a string T of length n over an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, where $\sigma = n^{\mathcal{O}(1)}$, and a regular expression R over Σ of size d.
Output (membership): ACCEPT, if $T \in L(R)$.
Output (pattern matching): all positions $1 \le r \le n$, such that there exists a position $1 \le \ell \le r$ such that $T[\ell \mathinner{..} r] \in L(R)$.

7.2 Definitions and tools.

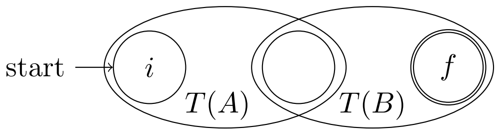
Let $A_1, A_2, \ldots, A_d$ be the atomic strings of the regular expression R. We define $\Pi = \{A_i[1 \mathinner{..} \min\{2^j, |A_i|\}] : 1 \le i \le d, 0 \le j \le \lceil \log |A_i| \rceil\}$. The prefixes of A_i's that belong to Π are called *canonical*.

For a string P and a text T, denote by $\mathsf{occ}(P, T)$ the set of the ending positions of the occurrences of P in T.

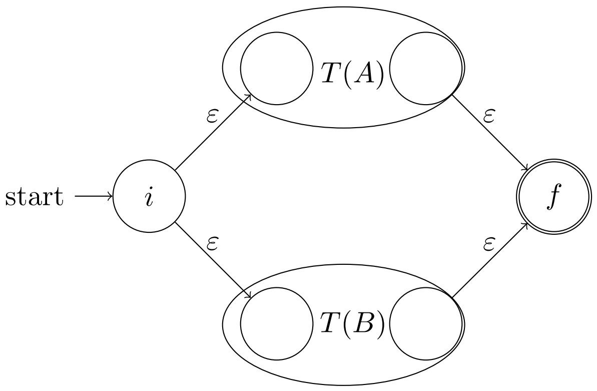
Definition 14 (Partial occurrence of a regular expression) *Consider a string S. We say that a fragment $S[i \mathinner{..} j]$, where $1 \le i \le j \le |S|$, is a partial occurrence of a regular expression R ending with a prefix P of an atomic string A, if there is a walk from the initial state of $T_C(R)$ to the endpoint of the transition corresponding*



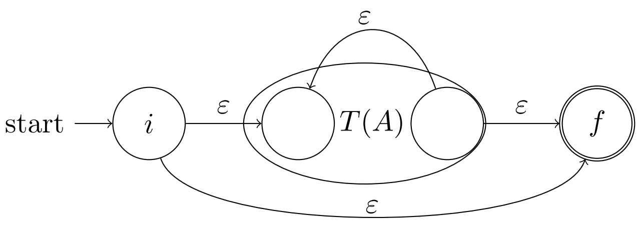
(a) $T(A)$ for $a \in \Sigma \cup \{\varepsilon\}$



(b) $T(A \cdot B)$



(c) $T(A|B)$

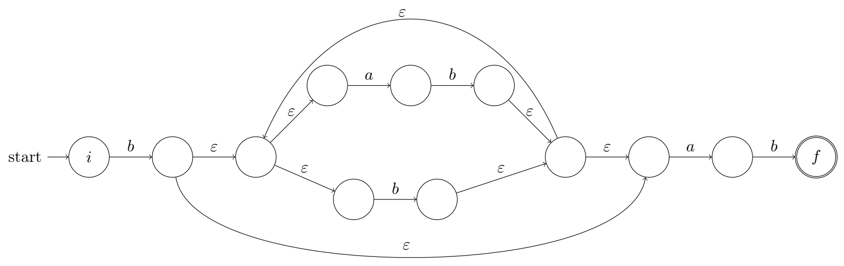


(d) $T(A^*)$

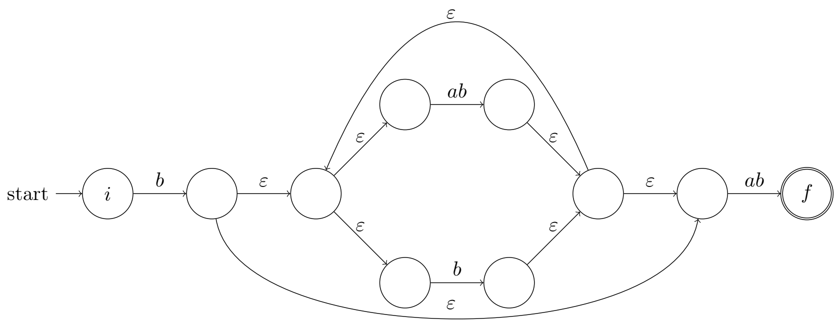
Figure 7.1: Thompson automaton. In each automaton, i and f are the initial and final states, respectively.

to A such that the concatenation of the labels of the transitions in this walk equals $S[i\,..\,j]A[|P|+1\,..]$.

Definition 15 (Witness) *Let P be a canonical prefix of an atomic string, and $r \in \mathsf{occ}(P,T)$. We say that r is a witness if $T[1\,..\,r]$ is a partial occurrence of R ending with P.*



(a) $T(b(ab|b)^*ab)$



(b) $T_C(b(ab|b)^*ab)$

Figure 7.2: The Thompson automatons of the regular expression $b(ab|b)^*ab$.

We also make use of the following well-known fact:

Corollary 7.1 (Of the Fine–Wilf periodicity lemma [64]) *If for strings P, X with $X \leq 2|P|$ we have $|\mathsf{occ}(P,X)| > 2$, then P is periodic and the set $\mathsf{occ}(P,X)$ can be represented as an arithmetic progression with difference $\mathsf{per}(P)$.*[1]

7.3 Overview of the algorithms

We now give an overview of our algorithms. Our solutions for streaming regular expression membership and pattern matching are very similar, so below we focus only on the membership variant of the problem.

We can assume that all atomic strings have length at most n, otherwise they never appear in the text and we can ignore them. Formally, during the preprocessing phase we delete all transitions (u, v) from $T_C(R)$ that are labelled by atomic strings of lengths larger than n. We also assume that $d \leq n$, otherwise we can use the following solution:

[1]Note that when $|\mathsf{occ}(P,X)| \leq 2$, we can represent $\mathsf{occ}(P,X)$ as at most two (degenerate) arithmetic progressions of length 1, we will use this fact to simplify the description of the algorithms.

Lemma 7.1 *Given a streaming text T of length n and a regular expression of size $d \geq n$. Assume that all atomic strings have length at most n each. There is a deterministic algorithm that solves the membership and the pattern matching problems for T and R in $\mathcal{O}(d^2)$ space and $\mathcal{O}(d^3)$ time per character of T.*

Proof. First note that we can afford storing T in full. Second, we build a compact trie on the reverses of the atomic strings of R. The trie occupies $\mathcal{O}(dn) = \mathcal{O}(d^2)$ space. Finally, let $\mathcal{F}$ contain all atomic strings A such that there is an ε-transitions path from the endpoint of the transition labelled by A to the final state of $T_C(R)$.

Define an array D of length $n+1 = \mathcal{O}(d)$ such that $D[0]$ contains a singleton set consisting of the starting state of $T_C(R)$ and $D[r]$, $1 \leq r \leq n$, stores all states u such that u is the end of some transition labelled by an atomic string and $T[1\,..\,r]$ equals the concatenation of the labels of the transitions in some walk from the starting state of $T_C(R)$ to u. Assume that we have constructed $D[1\,..\,r]$. To compute $D[r+1]$, we use the trie to find the atomic strings $A_1, A_2, \ldots, A_q$ equal to $T[1\,..\,r+1], T[2\,..\,r+1], \ldots$ or $T[r+1]$ in $\mathcal{O}(r+q)$ time. Note that $q \leq d$. For each atomic string A_i, $1 \leq i \leq q$, labelling a transition (v, w), we add w to $D[r+1]$ if there is a state u in $D[r+1-|A_i|]$ such that there is an ε-transition path from u to v, which can be checked in $\mathcal{O}(d)$ time and space. In total, the algorithm spends $\mathcal{O}(d^3)$ time to process a character of T ($q = \mathcal{O}(d)$, and for each $1 \leq i \leq q$ the set $D[r+1-|A_i|]$ contains $\mathcal{O}(d)$ states). The algorithm reports that $T \in L(R)$ if $D[n]$ contains a state v, which is an endpoint of a transition labelled by some $A \in \mathcal{F}$. □

From now on, we assume that all atomic strings have length at most n, and that $d \leq n$.

Intuition: non-periodic case. To give intuition behind our solution, consider a very simple case when every canonical prefix is not periodic. By Corollary 7.1, if none of the strings in Π is periodic, we can use the following approach. For each $P \in \Pi$ and T, we run the streaming pattern matching algorithm [120] and at any moment store the two most recent witnesses for P discovered by the algorithm. When the algorithm discovers a new position $r \in \mathsf{occ}(P, T)$, we must decide whether it is a witness. Let $P = A[1\,..\min\{2^k, |A|\}]$, where A is an atomic string.

If $k = 0$, we consider the starting node u of the transition in the compact Thompson automaton $T_C(R)$ labelled by A. Suppose that there is an ε-transitions path from the endpoints of the transitions labelled by atomic strings $A_{i_1}, A_{i_2}, \ldots, A_{i_j}$ to u. We then check if $(r-1)$ is a witness for at least one of $A_{i_1}, A_{i_2}, \ldots, A_{i_j}$. If it is, then r is a witness. Importantly, if $r-1$ is a witness for $A_{i_{j'}}$, $1 \leq j' \leq j$, it is the most recent one and is stored in the memory of the instance of the pattern matching algorithm for $A_{i_{j'}}$ and T. Suppose now that $k \geq 1$. We then must check whether $(r - 2^{k-1})$ is a witness for $A[1\,..\,2^{k-1}]$. If it is, then r is a witness for P. Note that by Corollary 7.1, if $(r - 2^{k-1})$ is a witness for $A[1\,..\,2^{k-1}]$, it is one of the two most recent ones and will be stored by the pattern matching algorithm for $A[1\,..\,2^{k-1}]$.

Let $\mathcal{F}$ contain all atomic strings A such that there is an ε-transitions path from the endpoint of the transition labelled by A to the final state of $T_C(R)$. In the regular expression pattern matching problem, we report all positions r such that r is a witness in $\mathsf{occ}(A, T)$ for some $A \in \mathcal{F}$. In the regular expression membership problem, $T \in L(R)$ if n is a witness for $\mathsf{occ}(A, T)$, for some $A \in \mathcal{F}$.

We do not provide the formal analysis of the algorithm, as we only give it for intuition, but it is easy to see that it uses $\mathcal{O}(d^2 \log^2 n)$ space and $\mathcal{O}(d \log^2 n)$ time per character of the text (recall that we do not account for the time spent during the preprocessing phase).

General case: main technical contributions. In general, unfortunately, some of the canonical prefixes are periodic. However, by Corollary 7.1, we obtain that every $r \in \mathsf{occ}(P, T)$, where P is a canonical prefix of some atomic string periodic with period ρ, belongs to a fragment of form $(\Delta(P))^k$, where $\Delta(P) = P[|P|-\rho+1\,..]$ and k is an integer. Instead of storing the last two witnesses for each canonical prefix, we would like to store the witnesses in the last two fragments of form $(\Delta(P))^k$. However, the number of such witnesses can be large. Our main technical novelty is a compressed representation of such witnesses. We give a high-level overview of the approach we use for the membership problem, our solution to the regular expression pattern matching problem is similar. We show that for each fragment of form $(\Delta(P))^k$ it suffices to store a small, carefully selected subset of witnesses that belong to this fragment. The remaining ones can be restored in small space at request.

Consider a witness $r \in \mathsf{occ}(P, T)$, where $P \in \Pi$. By definition, there is a partition $T[1\,..\,r] = T[\ell_1\,..\,r_1]T[\ell_2\,..\,r_2]\ldots T[\ell_m\,..\,r_m]$ such that each fragment in the partition, except for the last one, is an atomic string, and the last one equals P. Furthermore, by Corollary 7.1, r must belong to some fragment $F = T[i\,..\,i + k\rho - 1] = (\Delta(P))^k$. Let m' be the index of the first fragment such that $r_{m'} \geq i$. Consider the fragment $W = T[\ell_{m''}\,..\,r_{m''}]$, $m' \leq m'' \leq m$, containing a position $i + 2\rho - 1$ (we call this position an "anchor"). Note that W is a canonical prefix of some atomic string and $r_{m''} \in \mathsf{occ}(W, T)$ is a witness. If there are a few witnesses $t \in \mathsf{occ}(W, T)$ such that $T[t - |W| + 1, t]$ contains the anchor $i + 2\rho - 1$, we can store them explicitly. Otherwise, there is a periodic fragment containing $i + 2\rho - 1$, and we can recurse for it by choosing a new anchor close to its starting point. We choose the definition of anchors so that the recursion stops in a logarithmic number of steps and for some of the anchors there is a witness that we store explicitly for this anchor.

To summarize, the idea of the compact representation of witnesses that belong to a fragment of form $(\Delta(P))^k$ is to choose a logarithmic set of anchors close to the starting point of the fragment, and for each of these anchors to store a constant number of witnesses for each canonical prefix in Π. Suppose now that $r \in \mathsf{occ}(P, T)$, where P is a canonical prefix of an atomic string A, r belongs to a fragment $F = T[i\,..\,i + k\rho - 1] = (\Delta(P))^k$. To decide whether it is a witness we use the following approach. From above we know that r is a witness iff there is a witness $r' \in \mathsf{occ}(A', T)$, where A' is an atomic string, that we store in the compact representation of witness in F, and there is a path in $T_C(R)$ from the ending node of the transition labelled by A' and to the starting node of the transition labelled by A such that the concatenation of the strings on the edges of the path equals $T[r'+1\,..\,r-|A|]$ (which is a substring of $(\Delta(P))^k$). Unfortunately, it is not clear how to verify this condition in a straightforward way as we do not have random access neither to $\Delta(P)$, nor to the strings on the edges of $T_C(R)$. Instead, using anchors again, we show that verifying this condition can be reduced to the following question, where G is a graph of size $\mathrm{poly}(d, \log n)$:

Problem 7.2 ***Walks in a weighted graph***
Input: a directed multigraph G with non-negative integer weights on edges, two nodes, and a number x.
Output: ACCEPT, if there is a walk from the first node to the second one of total weight x.

We show the following theorem:

Theorem 7.1 *There exists an algorithm which, given a directed multigraph G with non-negative integer weights on edges, its two nodes v_1, v_2, and a number x, decides if there is a walk from v_1 to v_2 of total weight x in $\mathcal{O}((|E(G)|+|V(G)|^3)\cdot x \cdot \mathrm{polylog}\, x)$ time and $\mathcal{O}((|E(G)|+|V(G)|^3)\cdot \mathrm{polylog}\, x)$ space and succeeds with probability $\geq 1/2$.*

Let $N = |V(G)|$. For the simpler case when the graph is unweighted, we could use a folklore approach and compute the x-th power of the adjacency matrix in $\mathcal{O}(N^3 \log x)$ time and $\mathcal{O}(N^2)$ space. In order to handle arbitrary weights of edges, we compute the arrays C_k of bit-vectors of length $x+1$, where $C_k[u,v][d]$ stores a bit indicator of whether there exists a walk from u to v in G of at most 2^k edges of total weight exactly d. The following formula holds:

$$C_k[u,v][d] = \bigvee_{\substack{w \in V(G) \\ i \in \{0,\ldots,d\}}} C_{k-1}[u,w][i] \wedge C_{k-1}[w,v][d-i]$$

Using the fast Fourier transform to compute the convolutions, we obtain an algorithm with time $\mathcal{O}(N^3 x \log^2 x)$ and space $\mathcal{O}(N^2 x)$.

In our application, x can be equal to n, and the approach above uses $\Omega(n)$ space, which is prohibitive. In order to improve the space complexity, we represent the above computations as a circuit with binary Or and Convolution$_x$ gates operating on bit-vectors of length $x+1$. Every element $C_k[u,v]$ requires a separate gate and while computing its value we need to perform N convolutions, for every possible intermediate node w, so in total there are $\mathcal{O}(N^3 \log x)$ gates. The Convolution$_x$ gates store only the first $x+1$ bits of the results, as we never need paths of total weight larger than x. We are interested only in a single bit of output of the circuit, namely $C_{\lceil \log x \rceil}[v_1, v_2][x]$. If there were only Or gates in the circuit, we could store only the x-th element at each gate. In order to handle also $\textsc{Convolution}_x$ gates, we use the discrete Fourier transform over a suitably chosen ring.

We use the technique introduced by Lokshtanov and Nederlof [107] and then modified by Bringmann [33] to work with numbers modulo p instead of complex numbers. Informally, they show that if we operate on $\mathbb{Z}_p^t$ (vectors of length t with elements in $\mathbb{Z}_p$ for suitably chosen p and t) instead of the bit-vectors, we can compute $out(C)[x]$, the x-th element of the output of the circuit C in $\mathcal{O}(|C| \cdot t \cdot \text{polylog } p)$ time and $\mathcal{O}(|C| \log p)$ space. However, there are technical difficulties that we need to overcome to apply their technique to our solution. The approach of Bringmann [33] requires that $t > x$ and $\mathbb{Z}_p$ contains a t-th root of the unity. The main difficulty is to choose these numbers as small as possible as they directly affect the complexity of the algorithm. This question was also faced by Bringmann [33], who showed two variants of the framework, one using the Extended Riemann Hypothesis and the other unconditional but with polynomially higher time and space, which is not good enough for our streaming application. By using Bombieri–Vinogradov theorem and facts about counting primes in arithmetic progressions, we obtain an unconditional time bound comparable to that of Bringmann that assumes the Extended Riemann Hypothesis.

8 Language Distance Problem

"You can never understand one language until you understand at least two."

Geoffrey Willans

In this section, we give an overview of the small-space algorithms we developed in [18] for the online language distance problem for two formal languages, palindromes and squares. To develop them, we used, in particular, techniques from the solutions for streaming k-mismatch and k-edit pattern matching problems.

8.1 Statements of the problems and our results

Let us start with the formal statements of the problems and our results. We study the complexity of the *online* and *low-distance* version of the language distance problem for two classical languages: the language PAL of all palindromes, where a palindrome is a string that is equal to its reversed copy, and the language SQ of all squares, where a square is the concatenation of two copies of a string.

Problem 8.1 *k-LHD-PAL (resp. k-LHD-SQ)*
Input: *A string T of length n and a positive integer k.*
Output: *For each $1 \le i \le n$, report $\min\{k+1, hd_i\}$, where hd_i is the minimum Hamming distance between $T[1\,..\,i]$ and a string in PAL (resp. in SQ).*

Problem 8.2 *k-LED-PAL (resp. k-LED-SQ)*
Input: *A string T of length n and a positive integer k.*
Output: *For each $1 \le i \le n$, report $\min\{k+1, ed_i\}$, where ed_i is the minimum edit distance between $T[1\,..\,i]$ and a string in PAL (resp. in SQ).*

We give streaming algorithms which use $\mathrm{poly}(k, \log n)$ time per character and $\mathrm{poly}(k, \log n)$ space for all four problems. While streaming algorithms are extremely efficient, they are randomized by nature, which means that there is a small probability that they may produce incorrect results. Motivated by this, we also study the problems in the read-only model. In this model, we show *deterministic* algorithms for the four problems that use $\mathrm{poly}(k, \log n)$ time per character and $\mathrm{poly}(k, \log n)$ *extra* space (not accounting for the input); see Table 8.1 for a summary. As a side result of independent interest, we develop the first *deterministic* read-only algorithms for computing k-mismatch and k-edit occurrences of a pattern in a text using $\mathrm{poly}(k, \log n)$ space.

Problem	Model	Time per character	Space
k-LHD-PAL	Streaming	$O(k \log^3 n)$	$O(k \log n)$
k-LHD-SQ	Streaming	$\tilde{O}(k)$	$O(k \log^2 n)$
k-LHD-PAL/SQ	Read-only	$O(k \log n)$	$O(k \log n)$
k-LED-PAL/SQ	Streaming	$\tilde{O}(k^2)$	$\tilde{O}(k^2)$
k-LED-PAL/SQ	Read-only	$\tilde{O}(k^4)$ (amortised)	$\tilde{O}(k^4)$

Table 8.1: Summary of the complexities of the algorithms given in [18].

Below, we explain the main ideas behind our streaming algorithms.

8.2 Hamming distance, palindromes, and squares

Due to the self-similarity of palindromes and squares, the Hamming distance from a string U to PAL and SQ can be measured in terms of the self-similarity of U.

Property 8.1 *Each string $U \in \Sigma^m$ satisfies*

$$\mathsf{hd}(U, PAL) = \mathsf{hd}(U[\,.\,.\lfloor m/2 \rfloor], U(\lceil m/2 \rceil\,.\,.]^R) = \frac{1}{2}\mathsf{hd}(U, U^R).$$

Property 8.2 *Each string $U \in \Sigma^m$ satisfies $\mathsf{hd}(U, SQ) = \mathsf{hd}(U[\,.\,.m/2], U(m/2\,.\,.])$ if m is even and $\mathsf{hd}(U, SQ) = \infty$ if m is odd.*

For the rest of this section, let $\mathsf{sk}_k^{\mathsf{hd}}()$ denote the k-mismatch sketch [48] and $\mathsf{hd}_d(X, Y) = \min\{d + 1, \mathsf{hd}(X, Y)\}$. Using Property 8.1, we can reduce the k-LHD-PAL problem to that of computing the threshold Hamming distance between the current prefix of the input string and its reverse. The algorithm maintains the sketches $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[\,.\,.i])$ and $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[\,.\,.i]^R)$. Upon arrival of $T[i]$, it constructs $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[i])$, updates $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[\,.\,.i])$ and $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[\,.\,.i]^R)$, and computes $d = \mathsf{hd}_{\le 2k}(T[\,.\,.i], T[\,.\,.i]^R)$. Property 8.1 implies $\mathsf{hd}_{\le k}(T[\,.\,.i], \mathrm{PAL}) = \lceil d/2 \rceil$.

The algorithm uses $\mathcal{O}(k \log n)$ bits, which is nearly optimal: Indeed, by Property 8.1, if $U = VW$, with $|V| = |W|$, then $\mathsf{hd}(U, U^R) = 2 \cdot \mathsf{hd}(V, W^R)$. Therefore, using a standard reduction from streaming algorithms to one-way communication complexity protocols, we obtain a lower bound of $\Omega(k)$ bits for the space complexity of streaming algorithms for the k-LHD-PAL problem from the $\Omega(k)$ bits lower bound for the communication complexity of the Hamming distance [86].

Our solution for k-LHD-SQ is more involved. Property 8.2 allows us to derive $\mathsf{hd}_{\le k}(T[\,.\,.2i], \mathrm{SQ})$ from the sketches $\mathsf{sk}_k^{\mathsf{hd}}(T[\,.\,.i])$ and $\mathsf{sk}_k^{\mathsf{hd}}(T[\,.\,.2i])$: we can combine them to obtain $\mathsf{sk}_k^{\mathsf{hd}}(T(i\,.\,.2i])$, and a distance computation on $\mathsf{sk}_k^{\mathsf{hd}}(T[\,.\,.i])$ and $\mathsf{sk}_k^{\mathsf{hd}}(T(i\,.\,.2i])$ returns $\mathsf{hd}_{\le k}(T[\,.\,.i], T(i\,.\,.2i]) = \mathsf{hd}_{\le k}(T[\,.\,.2i], \mathrm{SQ})$.

Naively applying this procedure requires storing the sketch $\mathsf{sk}_k^{\mathsf{hd}}(T[..i])$ until the algorithm has read $T[..2i]$, that is, storing $\Theta(n)$ sketches at the same time. To reduce the number of sketches stored, we use a filtering procedure based on the following observation:

Observation 8.1 *If* $\mathsf{hd}(T[..2i], SQ) \leq k$ *and* $\ell \in [1..i]$*, then* $i+\ell$ *is a* k*-mismatch occurrence of* $T[..\ell]$*, that is,* $\mathsf{hd}(T[..\ell], T(i..i+\ell]) \leq k$*.*

Example 8.1 *For* $k = 1, \ell = 2$*, and* $i = 3$*, the string* $T[..6] = \texttt{abc}\underline{\texttt{a}}\underline{\texttt{c}}\texttt{c}$ *is a* 1*-mismatch square (by Property 8.2) and the fragment* $T(3..5] = \texttt{ac}$ *is a 1-mismatch occurrence of the prefix* $T[..2] = \texttt{ab}$*.*

Observation 8.1 motivates our filtering procedure: if we choose some prefix $P = T[..\ell]$ of the string, we only need to store every $i \geq \ell$ such that $i+\ell$ is a k-mismatch occurrence of P. Clifford, Kociumaka and Porat [48] showed a data structure $\mathcal{S}$ that exploits the structure of such occurrences and stores them using $O(k \log^2 n)$ bits of space while allowing reporting the occurrence at position $i + \ell$ when $T[i + \ell + \Delta]$ is pushed into $\mathcal{S}$ — we say that $\mathcal{S}$ reports the k-mismatch occurrences of P in T with *a fixed delay* Δ [48]. Our algorithm needs to receive the occurrence at position $i + \ell$ when $T[2i]$ is pushed into the stream, i.e., we require $\mathcal{S}$ to report occurrences with *non-decreasing* delays. We present a modification of the data structure [48] to allow non-decreasing delays, and use it to implement a space-efficient streaming algorithm for k-LHD-SQ.

8.3 Edit distance, palindromes, and squares

Similar to the Hamming distance, we show that the edit distance from a string U to PAL or SQ can be expressed in terms of "self-similarity" of U. This allows us to use similar approaches as for the Hamming distance problems, where tools for the Hamming distance are replaced by appropriate tools for the edit distance. By replacing the Hamming distance sketch with the edit distance sketch of [22], we obtain streaming algorithms for k-LED-PAL and $(1+\varepsilon)$-k-ED-PAL. Furthermore, the results of [22] show a reduction from the edit distance to the Hamming distance via locally consistent string decompositions, which allows us to solve k-LED-SQ in streaming by reducing to k-LHD-SQ.

Bibliography

[1] European nucleotide archive. https://www.ebi.ac.uk/ena/browser/about/statistics. Accessed: 2025-07-24. (Cited on page 4.)

[2] Sequence read archive. https://www.ncbi.nlm.nih.gov/sra/docs/sragrowth/. Accessed: 2025-07-24. (Cited on page 4.)

[3] Software Heritage Project. https://www.softwareheritage.org/. Accessed: 2025-07-24. (Cited on page 4.)

[4] A. Abboud and K. Bringmann. Tighter connections between formula-SAT and shaving logs. In *Proceedings of ICALP 2018*, volume 107 of *LIPIcs*, pages 8:1–8:18, 2018. (Cited on page 18.)

[5] A. Abboud, T. D. Hansen, V. V. Williams, and R. Williams. Simulating branching programs with edit distance and friends: Or: A polylog shaved is a lower bound made. In *Proceedings of STOC 2016*, pages 375—-388, 2016. (Cited on page 7.)

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

[7] P. Afshani. Improved pointer machine and I/O lower bounds for simplex range reporting and related problems. *Int. J. Comput. Geometry Appl.*, 23(4-5):233–252, 2013. (Cited on pages 14 and 41.)

[8] P. Afshani and J. S. Nielsen. Data structure lower bounds for document indexing problems. In *Proceedings of ICALP 2016*, volume 55 of *LIPIcs*, pages 93:1–93:15, 2016. (Cited on pages 14, 39 and 41.)

[9] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. (Cited on page 17.)

[10] J. Alman and R. Williams. Probabilistic polynomials and Hamming nearest neighbors. In *Proceedings of FOCS 2015*, pages 136–150, 2015. (Cited on pages 14, 15, 39 and 40.)

[11] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Molecular Biology*, 215(3):403–410, 1990. (Cited on page 7.)

[12] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004. (Cited on pages 6 and 11.)

[13] A. Amir and B. Porat. Approximate on-line palindrome recognition, and applications. In *Proceedings of CPM 2014*, volume 8486 of *LNCS*, pages 21–29, 2014. (Cited on page 19.)

[14] A. Babu, N. Limaye, J. Radhakrishnan, and G. Varma. Streaming algorithms for language recognition problems. *Theor. Comput. Sci.*, 494:13–23, 2013. (Cited on page 15.)

[15] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of STOC 2015*, pages 51—-58, 2015. (Cited on page 7.)

[16] A. Backurs and P. Indyk. Which regular expression patterns are hard to match? In *Proceedings of FOCS 2016*, pages 457–466. IEEE Computer Society, 2016. (Cited on page 17.)

[17] O. Barkol and Y. Rabani. Tighter lower bounds for nearest neighbor search and related problems in the cell probe model. *J. Comput. Syst. Sci.*, 64(4):873–896, June 2002. (Cited on page 14.)

[18] G. Bathie, T. Kociumaka, and T. Starikovskaya. Small-space algorithms for the online language distance problem for palindromes and squares. In *Proceedings of ISAAC 2023*, volume 283 of *LIPIcs*, pages 10:1–10:17, 2023. (Cited on pages 19, 20, 51 and 52.)

[19] G. Bathie and T. Starikovskaya. Property testing of regular languages with applications to streaming property testing of visibly pushdown languages. In *Proceedings of ICALP 2021*, volume 198 of *LIPIcs*, pages 119:1–119:17, 2021. (Cited on page 15.)

[20] D. Belazzougui and Q. Zhang. Edit distance: Sketching, streaming, and document exchange. In *Proceedings of FOCS 2016*, pages 51–60, 2016. (Cited on pages 12, 30, 34 and 36.)

[21] P. Berenbrink, F. Ergün, F. Mallmann-Trenn, and E. S. Azer. Palindrome recognition in the streaming model. In *Proceedings of STACS 2014*, volume 25, pages 149–161, 2014. (Cited on page 19.)

[22] S. Bhattacharya and M. Koucký. Locally consistent decomposition of strings with applications to edit distance sketching. In *Proceedings of STOC 2023*, pages 219–232, 2023. (Cited on pages 13 and 53.)

[23] S. Bhattacharya and M. Koucký. Streaming k-edit approximate pattern matching via string decomposition. In *ICALP 2023*, volume 261 of *LIPIcs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. (Cited on page 12.)

[24] P. Bille. New algorithms for regular expression matching. In *2006 International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 4051 of *Lecture Notes in Computer Science*, pages 643–654. Springer, 2006. (Cited on page 17.)

[25] P. Bille and M. Farach-Colton. Fast and compact regular expression matching. *Theoretical Computer Science*, 409(3):486–496, 2008. (Cited on page 17.)

[26] P. Bille and M. Thorup. Faster regular expression matching. In *Proceedings of ICALP 2009*, volume 5555 of *LNTCS*, pages 171–182. Springer, 2009. (Cited on page 17.)

[27] P. Bille and M. Thorup. Regular expression matching with multi-strings and intervals. In *Proceedings of SODA 2010*, pages 1297–1308, 2010. (Cited on pages 17 and 18.)

[28] A. Borodin, R. Ostrovsky, and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proceedings of STOC 1999*, pages 312–321, 1999. (Cited on page 14.)

[29] D. Breslauer. Saving comparisons in the Crochemore–Perrin string-matching algorithm. *Theoretical Computer Science*, 158(1):177–192, 1996. (Cited on page 5.)

[30] D. Breslauer and Z. Galil. Real-time streaming string-matching. *ACM Transaction on Algorithms*, 10(4):22:1–22:12, 2014. (Cited on page 5.)

[31] D. Breslauer and Z. Galil. Real-time streaming string-matching. *ACM Trans. Algorithms*, 10(4):22:1–22:12, 2014. (Cited on pages 18 and 29.)

[32] D. Breslauer, R. Grossi, and F. Mignosi. Simple real-time constant-space string matching. *Theoretical Computer Science*, 483:2–9, 2013. Special Issue Combinatorial Pattern Matching 2011. (Cited on page 5.)

[33] K. Bringmann. A near-linear pseudopolynomial time algorithm for subset sum. In *Proceedings of SODA 2017*, pages 1073–1084, 2017. (Cited on pages 19 and 50.)

[34] K. Bringmann, A. Grønlund, and K. G. Larsen. A dichotomy for regular expression membership testing. In *Proceedings of FOCS 2017*, pages 307–318, 2017. (Cited on pages 17 and 18.)

[35] K. Bringmann and M. Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of FOCS 2015*, pages 79–97, 2015. (Cited on page 7.)

[36] D. Chakraborty, E. Goldenberg, and M. Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of STOC 2016*, pages 712–725, 2016. (Cited on page 34.)

[37] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. Compressed indexes for approximate string matching. *Algorithmica*, 58(2):263–281, October 2010. (Cited on page 8.)

[38] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. A linear size index for approximate pattern matching. *J. of Discrete Algorithms*, 9(4):358 – 364, 2011. (Cited on page 8.)

[39] T. M. Chan, S. Golan, T. Kociumaka, T. Kopelowitz, and E. Porat. Approximating text-to-pattern Hamming distances. In *Proceedings of STOC 2020*, pages 643–656, 2020. (Cited on pages 6 and 11.)

[40] P. Charalampopoulos, T. Kociumaka, and P. Wellnitz. Faster approximate pattern matching: A unified approach. In *Proceedings of FOCS 2020*, pages 978–989, 2020. (Cited on pages 7, 13 and 30.)

[41] P. Charalampopoulos, T. Kociumaka, and P. Wellnitz. Faster pattern matching under edit distance : A reduction to dynamic puzzle matching and the seaweed monoid of permutation matrices. In *Proceedings of FOCS 2022*, pages 698–707. IEEE, 2022. (Cited on page 7.)

[42] B. Chazelle. Lower bounds for orthogonal range searching: I. The reporting case. *J. ACM*, 37(2):200–212, Apr. 1990. (Cited on page 41.)

[43] N. Chomsky and M. Schützenberger. The algebraic theory of context-free languages. In *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics*, pages 118–161. Elsevier, 1963. (Cited on page 16.)

[44] P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007. (Cited on page 17.)

[45] R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. Starikovskaya. Dictionary matching in a stream. In *Proceedings of ESA 2015*, volume 9294 of *LNCS*, pages 361–372, 2015. (Cited on pages 18 and 27.)

[46] R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. Starikovskaya. The k-mismatch problem revisited. In *Proceedings of SODA 2016*, pages 2039–2052, 2016. (Cited on pages 6, 9, 10, 11, 12, 20, 24 and 29.)

[47] R. Clifford, M. Jalsenius, and B. Sach. Cell-probe bounds for online edit distance and other pattern matching problems. In *Proceedings of SODA 2015*, pages 552–561, 2015. (Cited on page 7.)

[48] R. Clifford, T. Kociumaka, and E. Porat. The streaming k-mismatch problem. In *Proceedings of SODA 2019*, pages 1106–1125, 2019. (Cited on pages 9, 11, 12, 13, 18, 29, 31, 36, 52 and 53.)

[49] R. Clifford and B. Sach. Pseudo-realtime pattern matching: Closing the gap. In *Proceedings of CPM 2010*, volume 6129 of *LNTCS*, pages 101–111, 2010. (Cited on page 27.)

[50] A. L. Cobbs. Fast approximate matching using suffix trees. In *Proceedings of CPM 1995*, LNCS, pages 41–54, 1995. (Cited on page 8.)

[51] V. Cohen-Addad, L. Feuilloley, and T. Starikovskaya. Lower bounds for text indexing with mismatches and differences. In *Proceedings of SODA 2019*, pages 1146–1164, 2019. (Cited on pages 13, 14, 15, 16, 20 and 38.)

[52] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of STOC 2004*, pages 91–100, 2004. (Cited on pages 7, 8 and 12.)

[53] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM J. Comput.*, 31(6):1761–1782, 2002. (Cited on page 7.)

[54] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings of STOC 2002*, page 592–601, 2002. (Cited on page 17.)

[55] M. Crochemore. String-matching on ordered alphabets. *Theoretical Computer Science*, 92(1):33–47, 1992. (Cited on page 5.)

[56] M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38(3):650–674, July 1991. (Cited on page 5.)

[57] M. Crochemore and W. Rytter. Periodic prefixes in texts. In *Sequences II*, pages 153–165, 1993. (Cited on page 5.)

[58] M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. (Cited on page 5.)

[59] B. Dudek, P. Gawrychowski, G. Gourdel, and T. Starikovskaya. Streaming regular expression membership and pattern matching. In *Proceedings of SODA 2022*, pages 670–694, 2022. (Cited on pages 18, 20 and 43.)

[60] F. Ergun, H. Jowhari, and M. Sağlam. Periodicity in streams. In *Proceedings of APPROX-RANDOM 2010*, pages 545–559, 2010. (Cited on page 5.)

[61] S. Faro and T. Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2), Mar. 2013. (Cited on page 3.)

[62] S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio. The string matching algorithms research tool. `https://smart-tool.github.io/smart/`. Accessed: 2025-07-22. (Cited on page 3.)

[63] S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio. The string matching algorithms research tool. In *Proceedings of the PSC 2016*, pages 99–111, 2016. (Cited on page 3.)

[64] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965. (Cited on page 46.)

[65] M. J. Fischer and M. S. Paterson. String-matching and other products. Technical report, Massachusetts Institute of Technology, USA, 1974. (Cited on page 17.)

[66] N. François, F. Magniez, M. de Rougemont, and O. Serre. Streaming property testing of visibly pushdown languages. In *ESA 2016*, volume 57 of *LIPIcs*, pages 43:1–43:17, 2016. (Cited on page 15.)

[67] Z. Galil. Open problems in stringology. In *Combinatorial Algorithms on Words*, pages 1–8, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. (Cited on page 17.)

[68] M. Ganardi, D. Hucke, D. König, M. Lohrey, and K. Mamouras. Automata theory on sliding windows. In *Proceedings of STACS 2018*, volume 96 of *LIPIcs*, pages 31:1–31:14, 2018. (Cited on page 15.)

[69] M. Ganardi, D. Hucke, and M. Lohrey. Querying regular languages over sliding windows. In *Proceedings of FSTTCS 2016*, volume 65 of *LIPIcs*, pages 18:1–18:14, 2016. (Cited on page 15.)

[70] M. Ganardi, D. Hucke, and M. Lohrey. Randomized sliding window algorithms for regular languages. In *Proceedings of ICALP 2018*, volume 107 of *LIPIcs*, pages 127:1–127:13, 2018. (Cited on page 15.)

[71] M. Ganardi, D. Hucke, and M. Lohrey. Sliding window algorithms for regular languages. In *Proceedings of LATA 2018*, volume 10792, pages 26–35, 2018. (Cited on page 15.)

[72] M. Ganardi, D. Hucke, M. Lohrey, K. Mamouras, and T. Starikovskaya. Regular languages in the sliding window model. *TheoretiCS*, 4, 2025. (Cited on page 15.)

[73] M. Ganardi, D. Hucke, M. Lohrey, and T. Starikovskaya. Sliding window property testing for regular languages. In *ISAAC 2019*, volume 149 of *LIPIcs*, pages 6:1–6:13, 2019. (Cited on page 15.)

[74] M. Ganardi, A. Jeż, and M. Lohrey. Sliding windows over context-free languages. In *Proceedings of MFCS 2018*, volume 117 of *LIPIcs*, pages 15:1–15:15, 2018. (Cited on page 15.)

[75] M. N. Garofalakis, R. Rastogi, and K. Shim. Mining sequential patterns with regular expression constraints. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(3):530–552, 2002. (Cited on page 16.)

[76] P. Gawrychowski, O. Merkurev, A. M. Shur, and P. Uznanski. Tight tradeoffs for real-time approximation of longest palindromes in streams. *Algorithmica*, 81(9):3630–3654, 2019. (Cited on page 19.)

[77] P. Gawrychowski and T. Starikovskaya. Streaming dictionary matching with mismatches. *Algorithmica*, 84(4):896–916, 2022. (Cited on pages 12 and 13.)

[78] P. Gawrychowski and P. Uznański. Towards unified approximate pattern matching for Hamming and L_1 distance. In *Proceedings of ICALP 2018*, volume 107 of *LIPIcs*, pages 62:1–62:13, 2018. (Cited on page 6.)

[79] L. Gąsieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: sequential sampling. In *Proceedings of CPM 1995*, pages 78–89, 1995. (Cited on page 5.)

[80] L. Gąsieniec, W. Plandowski, and W. Rytter. The zooming method: a recursive approach to time-space efficient string-matching. *Theoretical Computer Science*, 147(1):19–30, 1995. (Cited on page 5.)

[81] S. Golan, T. Kociumaka, T. Kopelowitz, and E. Porat. The streaming k-mismatch problem: Tradeoffs between space and total time. In *Proceedings of CPM 2020*, volume 161 of *LIPIcs*, pages 15:1–15:15, 2020. (Cited on pages 9 and 11.)

[82] S. Golan, T. Kopelowitz, and E. Porat. Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In *Proceedings of ICALP 2018*, volume 107 of *LIPIcs*, pages 65:1–65:16, 2018. (Cited on pages 9, 11 and 18.)

[83] S. Golan, T. Kopelowitz, and E. Porat. Streaming pattern matching with d wildcards. *Algorithmica*, 81(5):1988–2015, 2019. (Cited on page 18.)

[84] S. Golan and E. Porat. Real-time streaming multi-pattern search for constant alphabet. In *Proceedings of ESA 2017*, volume 107 of *LIPIcs*, pages 41:1–41:15, 2017. (Cited on page 18.)

[85] E. Grigorescu, E. Sadeqi Azer, and S. Zhou. Streaming for aibohphobes: Longest palindrome with mismatches. In *Proceedings of FSTTCS 2018*, volume 93, pages 31:1–31:13, 2018. (Cited on page 19.)

[86] W. Huang, Y. Shi, S. Zhang, and Y. Zhu. The communication complexity of the Hamming distance problem. *Information Processing Letters*, 99(4):149–153, 2006. (Cited on pages 9 and 52.)

[87] T. N. D. Huynh, W.-K. Hon, T.-W. Lam, and W.-K. Sung. Approximate string matching using compressed suffix arrays. *J. Theor. Comput. Sci.*, 352(1):240–249, Mar. 2006. (Cited on page 8.)

[88] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *J. of Computer and System Sciences*, 62(2):367–375, 2001. (Cited on pages 7 and 14.)

[89] P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. In *Proceedings of FOCS 1998*, page 166, 1998. (Cited on page 17.)

[90] C. Jin, J. Nelson, and K. Wu. An improved sketching bound for edit distance. In *Proceedings of STACS 2021*, volume 187 of *LIPIcs*, pages 45:1–45:16, 2021. (Cited on pages 30, 34 and 36.)

[91] T. Johnson, S. M. Muthukrishnan, and I. Rozenbaum. Monitoring regular expressions on out-of-order streams. In *Proceedings of ICDE 2007*, pages 1315–1319, 2007. (Cited on page 16.)

[92] A. Kalai. Efficient pattern-matching with don't cares. In *Proceedings of SODA 2002*, page 655–656, 2002. (Cited on page 17.)

[93] H. Karloff. Fast algorithms for approximately counting mismatches. *J. of Information Processing Letters*, 48(2):53–60, 1993. (Cited on page 26.)

[94] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. of R&D*, 31(2):249–260, 1987. (Cited on pages 5 and 11.)

[95] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton: multitouch gestures as regular expressions. In *Proceedings of CHI 2012*, page 2885–2894, 2012. (Cited on page 16.)

[96] S. C. Kleene. *Representation of events in nerve nets and finite automata.* RAND Corporation, Santa Monica, CA, 1951. (Cited on page 16.)

[97] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. (Cited on page 3.)

[98] T. Kociumaka, E. Porat, and T. Starikovskaya. Small-space and streaming pattern matching with k edits. In *Proceedings of FOCS 2021*, pages 885–896. IEEE, 2021. (Cited on pages 12, 13, 20 and 29.)

[99] S. Kosaraju. Efficient string matching. 1987. (Cited on page 6.)

[100] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. S. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of SIGCOMM 2006*, page 339–350, 2006. (Cited on page 16.)

[101] T.-W. Lam, W.-K. Sung, and S.-S. Wong. Improved approximate string matching using compressed suffix data structures. *J. Algorithmica*, 51(3):298–314, Jul 2008. (Cited on page 8.)

[102] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986. (Cited on pages 6 and 26.)

[103] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989. (Cited on page 7.)

[104] LeetCode. Problem 139. Word break. `https://leetcode.com/problems/word-break/`. (Cited on page 17.)

[105] M. Lewenstein, J. I. Munro, V. Raman, and S. V. Thankachan. Less space: Indexing for queries with wildcards. *Theoretical Computer Science*, 557:120 – 127, 2014. (Cited on page 39.)

[106] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of VLDB 2001*, page 361–370, 2001. (Cited on page 16.)

[107] D. Lokshtanov and J. Nederlof. Saving space by algebraization. In *Proceedings of STOC 2010*, pages 321–330, 2010. (Cited on pages 19 and 50.)

[108] F. Magniez, C. Mathieu, and A. Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM J. Comput.*, 43(6):1880–1905, 2014. (Cited on page 15.)

[109] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980. (Cited on page 7.)

[110] M. Minsky and S. A. Papert. *Perceptrons: An introduction to computational geometry.* MIT press, 1969. (Cited on page 38.)

[111] P. Muir, S. Li, S. Lou, D. Wang, D. Spakowicz, L. Salichos, J. Zhang, G. Weinstock, F. Isaacs, J. Rozowsky, and M. Gerstein. The real cost of sequencing: Scaling computation to keep pace with data generation. *Genome Biology*, 17(1), Mar. 2016. Publisher Copyright: © 2016 Muir et al. (Cited on page 5.)

[112] M. Murata. Extended path expressions of XML. In *Proceedings of PODS 2001*, page 126–137, 2001. (Cited on page 16.)

[113] E. W. Myers. A four Russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(2):432–448, Apr. 1992. (Cited on page 17.)

[114] E. W. Myers. *What's behind Blast*, pages 3–15. Springer London, 2013. (Cited on page 7.)

[115] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar. 2001. (Cited on page 6.)

[116] G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *Proceedings of RECOMB 2001*, page 231–240, 2001. (Cited on page 16.)

[117] N. Nisan. Pseudorandom generators for space-bounded computation. *Comb.*, 12(4):449–461, 1992. (Cited on page 34.)

[118] C. Noam. Systems of syntactic analysis. *Journal of Symbolic Logic*, 18:242–256, 1958. (Cited on page 16.)

[119] R. Nussinov and A. B. Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences of the United States of America*, 77 11:6309–13, 1980. (Cited on page 1.)

[120] B. Porat and E. Porat. Exact and approximate pattern matching in the streaming model. In *Proceedings of FOCS 2009*, pages 315–323, 2009. (Cited on pages 5, 9, 10, 18, 27, 29 and 47.)

[121] M. Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, June 2011. (Cited on pages 15 and 39.)

[122] J. Radoszewski and T. Starikovskaya. Streaming k-mismatch with error correcting and applications. In *Proceedings of DCC 2017*, pages 290–299, 2017. (Cited on pages 9, 10 and 11.)

[123] J. Radoszewski and T. Starikovskaya. Streaming k-mismatch with error correcting and applications. *Journal of Information and Computation*, 271:104513, 2020. (Cited on pages 9 and 10.)

[124] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. (Cited on page 11.)

[125] A. Rubinstein. Hardness of approximate nearest neighbor search. In *Proceedings of STOC 2018*, pages 1260–1268, 2018. (Cited on pages 14, 15 and 40.)

[126] W. Rytter. On maximal suffixes and constant-space linear-time versions of KMP algorithm. *Theoretical Computer Science*, 299(1):763 – 774, 2003. (Cited on page 5.)

[127] B. Saha. Fast & space-efficient approximations of language edit distance and RNA folding: An amnesic dynamic programming approach. In *Proceedings of FOCS 2017*, pages 295–306. IEEE Computer Society, 2017. (Cited on page 1.)

[128] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *Proceedings of FOCS 1996*, pages 320–328, 1996. (Cited on page 7.)

[129] A. Sandelin, W. Alkema, P. Engström, W. W. Wasserman, and B. Lenhard. Jaspar: an open-access database for eukaryotic transcription factor binding profiles. *Nucleic acids research*, 32(suppl_1):D91–D94, 2004. (Cited on page 10.)

[130] P. Schepper. Fine-grained complexity of regular expression pattern matching and membership. In *Proceedings of ESA 2020*, volume 173 of *LIPIcs*, pages 80:1–80:20, 2020. (Cited on page 18.)

[131] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359 – 373, 1980. (Cited on page 7.)

[132] T. Starikovskaya. Communication and streaming complexity of approximate pattern matching. In *CPM 2017*, volume 78 of *LIPIcs*, pages 13:1–13:11, 2017. (Cited on pages 12 and 13.)

[133] K. Thompson. Programming techniques: regular expression search algorithm. *Communication of ACM*, 11(6):419–422, 1968. (Cited on pages 16 and 43.)

[134] D. Tsur. Fast index for approximate string matching. *J. Discrete Algorithms*, 8:339–345, 2010. (Cited on page 8.)

[135] D. Tunkelang. Retiring a great interview problem. `https://thenoisychannel.com/2011/08/08/retiring-a-great-interview-problem/`, 2011. (Cited on page 17.)

[136] E. Ukkonen. Approximate string-matching over suffix trees. In *Proceedings of CPM 1993*, volume 684 of *LNCS*, pages 228–242, 1993. (Cited on page 8.)

[137] P. Weiner. Linear pattern matching algorithms. In *Proceedings of SWAT 1973*, pages 1–11, 1973. (Cited on page 3.)

[138] X. Xia. Position weight matrix, gibbs sampler, and the associated significance tests in motif characterization and prediction. *Scientifica*, 2012(1):917540, 2012. (Cited on page 10.)

[139] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of ANCS 2006*, pages 93–102, 2006. (Cited on page 16.)

[140] S. Zacchiroli. Why Software Heritage is creating a global software archive. `https://www.polytechnique-insights.com/en/columns/economy/source-code-building-a-universal-software-archive/`. Accessed: 2025-07-24. (Cited on page 4.)

RÉSUMÉ

Cette habilitation à diriger des recherches porte sur les algorithmes et les structures de données pour le problème fondamental de la recherche de motifs et ses diverses applications. Traditionnellement, la recherche de motifs a occupé une place centrale dans des domaines où les données peuvent être représentées sous forme de texte, tels que la bioinformatique, la recherche d’information et la sécurité numérique. Cependant, les données modernes dans ces domaines posent de nouveaux défis : elles sont de plus en plus massives, fragmentées, bruitées ou en évolution constante. Pour relever ces défis, cette thèse développe les fondements des algorithmes et des structures de données pour la recherche de motifs et ses applications pour ces formes complexes de données. Ses objectifs sont doubles : approfondir notre compréhension des bornes inférieures sur la complexité de ces problèmes et concevoir des méthodes efficaces pouvant avoir un impact en bioinformatique, en recherche d’information et en sécurité numérique.

ABSTRACT

This habilitation thesis focuses on algorithms and data structures for the fundamental problem of pattern matching and its diverse applications. Traditionally, pattern matching has been central in domains where data can be represented as text, such as bioinformatics, information retrieval, and digital security. Yet, modern data in these areas poses new challenges: inputs are increasingly massive, fragmented, noisy, or continuously evolving. To address these challenges, this thesis develops the foundations of algorithms and data structures for pattern matching and its applications for such complex forms of data. Its goals are twofold: to deepen our understanding of the computational limits of these problems, and to design efficient methods that can have impact in bioinformatics, information retrieval, and digital security.