University of Catania
Department of Mathematics and Computer Science
PhD in Computer Science

Compact and Flexible Suffix Automata Representations for Online String Matching

Stefano Scafiti

Supervisor
Prof. Simone Faro

XXXVI Cycle

To my family.

Index

Index i

1 Introduction 2
- 1.1 Motivation . . . 2
  - 1.1.1 Main contributions . . . 6
  - 1.1.2 Outline . . . 7

2 Preliminaries 8
  - 2.0.1 Notation . . . 8
  - 2.0.2 Factor based searching . . . 9
  - 2.0.3 Condensed Alphabets . . . 11

3 PBNDM: a sampled bit-parallel suffix automata for large strings 13
- 3.1 The Pruned BNDM Algorithm . . . 13
  - 3.1.1 The Pruned Version of a Pattern . . . 14
  - 3.1.2 The Preprocessing Phase . . . 17
  - 3.1.3 The Searching Phase . . . 18
- 3.2 Experimental comparison . . . 21
- 3.3 Chapter summary . . . 22

4 The Range Automaton: An Efficient Approach to Text-Searching 24
- 4.1 The Range Automaton . . . 24
- 4.2 The Backward Range Automaton Matcher . . . 28
  - 4.2.1 Speeding-up Searching by Condensed Alphabets . . . 30
- 4.3 Extensions to Non-Standard Matching Problems . . . 32
  - 4.3.1 Extension to Swap matching . . . 32
  - 4.3.2 BRAM for Swap Matching . . . 34
- 4.4 Extension to Order Preserving String Matching . . . 37
  - 4.4.1 BRAM for Order Preserving String Matching . . . 39
- 4.5 Extension to Multiple String Matching . . . 41
  - 4.5.1 BRAM for Multiple String Matching . . . 43
- 4.6 Experimental Comparison . . . 44
  - 4.6.1 Exact string matching . . . 45
  - 4.6.2 Experimental Results on Swap Matching . . . 48
  - 4.6.3 Experimental Results on Order Preserving String Matching . . . 49
  - 4.6.4 Experimental Results on Multiple String Matching . . . 51
- 4.7 Chapter summary . . . 54

5 UFM: a Two-Step Simulation of the Suffix Automaton **56**
5.1 The Unique Factor Matcher 56
5.1.1 A Generic Backward-Two-Step-Matcher Algorithm 57
5.1.2 A Practical Implementation: The UFM Algorithm 59
5.1.3 A Relaxed Variant of the UFM Algorithm 64
5.1.4 Improving the space usage 67
5.2 Experimental Results 67
5.2.1 Experimental Setting 68
5.2.2 Evaluation 69
5.2.3 Chapter Summary 72

6 Conclusions **74**

Appendices **75**

A On the Longest Common Cartesian Substring Problem **76**
A.1 Introduction 76
A.2 Notations and Definitions 78
A.3 Building a Cartesian Tree 79
A.4 A Suffix Tree Based Approach 84
A.5 Computing the LCCS by Dynamic Programming 85
A.6 A Constructive Approach for the LCCS Problem 90
A.6.1 A Backward Approach Over the Constructive Solution 97
A.7 Experimental Results 99
A.7.1 Implementation Details 100
A.7.2 Results on Random and Real Data 101
A.7.3 Results on Real Data 106
A.7.4 Conclusions 109

Bibliography **108**

1 Introduction

String matching is a fundamental problem in computer science [1], with a myriad of direct applications into several distinct areas of computing, including information retrieval, data compression, bioinformatics, natual language processing, linguistics and network security. It consists in finding all the occurrences of a pattern x of length m in a text y of length n, both drawn from a common alphabet Σ of size σ. The problem has attracted researchers' interest over the years, inspiring solutions of particular practical and theoretical interest. Among them, those based on automata deserve a special mention, since they have always led to the design of flexible and efficient algorithms.

1.1 Motivation

Despite a multitude of possible solutions already available, in the last few decades, alternative solutions to the problem have been proposed, based on very different techniques [2], proving that the demand for more and more efficient solutions is still high. After the first linear-time solution to the problem has been presented by

Knuth, Morris and Pratt (KMP [3]), efforts have been directed in finding solutions which perform well on the average, rather then on the worst case. It is the case of the Boyer and Moore (BM) [4], which trades the $O(m+n)$ worst case time complexity in turn of a more efficient search times on the average. Later, the Backward-DAWG-Matching (BDM) algorithm [5] reached the optimal $\mathcal{O}(n \log_\sigma m/m)$ time complexity on the average, as proved by Yao [6].

Both the KMP and the BDM algorithms are based on finite automata; in particular, they simulate a deterministic automaton for the language $\Sigma^\star x$ and a deterministic suffix automaton for the language of the suffixes of x, respectively. The subsequent solutions to the problem introduced in the literature [7, 8, 9, 10] have amply demonstrated how the efficiency of such solutions is strictly affected by the encoding used for simulating the underlying automaton.

One of the side effects of the BDM algorithm lies indeed in the use of the deterministic variant of the suffix automaton since the workload required to manage the individual transitions may be not negligible and, although its construction is linear in the size of the string, the proportionality factor hidden in the asymptotic notation is particularly high, making its construction prohibitive in the case of long patterns [2].

An efficient technique which has been extensively used for the simulation of non-deterministic automaton is *bit parallelism* [11]. It has been used, for instance, in the design of the Shift-Or (SO) algorithm [11] and the Backward-Non-deterministic-DAWG-Matching (BNDM) algorithm [12]. The first is based on the non-deterministic simulation of the KMP automaton, while the second is a very fast variant of the BDM algorithm, based on the bit-parallel simulation of the non-deterministic suffix automaton.

Specifically, in the design of automata-based algorithms, bit-parallelism allows to take advantage of the intrinsic parallelism of the bitwise operations inside a computer word, potentially cutting down the number of operations required to simulate the state transitions of the underlying automaton by a factor up to w, i.e. the num-

ber of bits in a computer word. However, one bit per pattern symbol is required for representing the states of the automaton, for a total of $\lceil m/\omega \rceil$ words. This implies that, as long as a pattern fits in a computer word, bit-parallel algorithms are extremely fast, otherwise their performances degrade considerably as $\lceil m/\omega \rceil$ grows.

A common approach to overcome this problem consists in constructing an automaton for a substring of the pattern fitting in a single computer word, to filter out possible candidate occurrences of the pattern. When an occurrence of the selected substring is found, a subsequent naive verification phase allows to establish whether this belongs to an occurrence of the whole pattern. However, besides the costs of the additional verification phase, a drawback of this approach is that, in the case of the BNDM algorithm, the maximum possible shift length cannot exceed ω, which could be much smaller than m.

The Long-BNDM [7] (LBNDM) and the BNDM with eXtended Shift [8] (BXS) algorithms are two efficient solutions specifically designed for simulating the suffix automaton using bit-parallelism in the case of long patterns. Specifically the LBNDM algorithm works by partitioning the pattern in $\lfloor m/k \rfloor$ consecutive substrings, each consisting in $k = \lfloor (m-1)/\omega \rfloor + 1$ characters. Similarly, the BXS algorithm cuts the pattern into $\lceil m/\omega \rceil$ consecutive substrings of length w except for the rightmost piece which may be shorter. In both cases the substrings are superimposed getting a superimposed pattern of length ω. The idea is to search using a filter approach: first the superimposed pattern is searched in the text, then an additional verification phase is run when a candidate occurrence of the pattern has been located.

Cantone *et al.* presented in [9] an alternative technique, still suitable for bit-parallelism, to encode the non-deterministic suffix automaton of a given string in a more compact way. Their encoding is based on factorization of strings in which no character occurs more than once in any factor. It turns out that the non-deterministic automaton can be encoded with k bits, where k is the size of the factorization. As a consequence, the resulting algorithm, called Factorized-BNDM

(FBNDM) tends to be faster in the case of sufficiently long patterns.

Although the several attempts to improve the bit-parallelism approach, such limitation is intrinsic to the approach and the room for improvement is still high.

Due to the challenges in simulating exact structures, research on the topic has recently focused more on *weak recognition approaches* [13, 14, 15, 8]. Weak factor recognition is not a new idea in the string matching field. The Backward Oracle Matching algorithm [16] (BOM) can be considered the pioneer of this approach. It works in the same way as the BDM algorithm, but makes use of the Factor Oracle of the reverse pattern, a structure which can be constructed and handled using fewer resources than the suffix automaton, leading to performance that is in practice superior to that achieved by the BDM algorithm (though the lack of any theoretical proof about the optimal time bound on the average case [16]).

Following the same line, in [8], Durian, Peltola, Salmela and Tarhio presented Q-gram Filtering (QF), a weak factor algorithm which uses hashing of q-grams within the pattern to identify factors. The phase of each q-gram factor is stored into a table as $1 \ll (p \mod q)$, where p is the position of the q-gram within the pattern and q is the length of a q-gram. Successive q-grams read from the text must not only have an entry in table, but also be aligned with the phase of all q-grams which have been read previously. In practice, this is generally very fast.

In [10] Faro and Lecroq presented a very flexible solution based on the intuition that a string where each character is repeated only once (SNR) admits a non-deterministic suffix automaton which can be encoded with a simple integer. The BSDM algorithm is based on the identification of the longest SNR contained within the pattern and on the simulation of this simplified version of the automaton. When this technique is combined with the use of condensed alphabets we are able to efficiently simulate automata for very long patterns.

This research trend reached its peak with the Weak Factor Recognition algorithm [15] (WFR), regarded as one of the most efficient state-of-the-art solutions. Its weak recognition approach is based on indexing all the $O(m^2)$ subsequences of

the pattern x to facilitate their search during the searching phase using a *bloom filter* [17], a probabilistic data structure which implicitly represents a set of elements U by setting a bit in a hash table for each element. In the case of a string matching algorithm, this implies that the algorithm recognizes a superset of all factors $x[i..j]$, $0 \leq i, j < m$, of the input string x. Despite the probabilistic nature of its recognition approach, the WFR algorithm and its variants perform very well in practice by showing a sub-linear performance in practice.

Following the work done in the last decades on weak factor recognition, this thesis aims at addressing the challenge in designing alternative, approximated encoding of the suffix automaton enabling a more efficient simulation in practice. Specifically, it focuses on designing possible encoding of the suffix automaton with the following key properties:

1. P1 - search speed should not degrade as the size of the pattern m grows;
2. P2 - it should be possible to perform state transitions in a straightforward and efficient way;
3. P3 - the amount of space required to store the encoding should not be a function of the pattern size m;
4. P4 - the resulting solutions should be flexible enough to be adapted to other non-standard string matching problems.

The contribution of thesis consists in several efficient online string matching algorithms which have been modelled upon the aforementioned design goals, and which are briefly introduced in the next section.

1.1.1 Main contributions

The research work presented in this thesis can be summarized with the introduction of the following algorithms:

- Pruned BNDM (PBNDM): a variant of the BNDM algorithm, which searches for a sampled version of the input pattern whose bit-parallel automaton can be represented with a reduced number of bits [18, 19]. Although still based on bit-parallelism, the algorithm scales well on long patterns;

- Backward Range Automaton Matcher (BRAM): a weak yet efficient variant of the non-deterministic suffix automaton of a string whose configuration can be encoded in a very simple form and which is particularly suitable to be used for solving a multitude of text-searching problems [20, 14].

- Unique Factor Matcher (UFM): an algorithm based on an approximated, non-standard two-steps simulation of the suffix automaton [21, 19]. Multiple variations of the base algorithm are also presented, which different search and space characteristics.

1.1.2 Outline

The rest of this thesis is organized as follows. Chapter 2 presents some preliminary notions and definitions which are useful to present the rest of the thesis. Later, Chapter 3 starts with the presentation of the Pruned BNDM algorithm and evaluates its behaviour. A similar structure can be found in Chapter 4 and 5, describing, respectively, the Backward Range Automaton algorithm (and its adaptation to several non-standard string matching problems) and the Unique Factor Matcher algorithm, along with with a few variants of it. Finally, Chapter 6 draws final conclusions on this thesis and elucidates possible future work. Additionally, appendix A further extends the work of this thesis by presenting novel results on the *longest common cartestian substring* problem which are unrelated to main topic of the thesis.

2

Preliminaries

This chapter gives the formal definition of the search problems which are discussed along the rest of this thesis. It also establishes the basic notions and definitions which are useful to understand the content of the upcoming chapters.

2.0.1 Notation

Given a finite alphabet Σ, we denote by Σ^m, with $m \geq 0$, the set of all strings of length m over Σ. We represent a string $x \in \Sigma^m$ as an array $x[0\,..\,m-1]$ of characters of Σ and write $|x| = m$ (for $m = 0$ we obtain the empty string ε). Thus, $x[i]$ is the $(i+1)$-st character of x, for $0 \leq i < m$, and $x[i\,..\,j]$ is the substring of x contained between its $(i+1)$-st and the $(j+1)$-st characters, for $0 \leq i \leq j < m$.

For any two strings x and x', we say that x' is a suffix of x if $x' = x[i\,..\,m-1]$, for some $0 \leq i < m$, and write $\mathit{Suff}(x)$ for the set of all suffixes of x. Similarly, x' is a prefix of x if $x' = x[0\,..\,i]$, for some $0 \leq i < m$, and write x_i to indicate the prefix of length i of x, i.e. $x_i = x[0..i-1]$. We write $x \cdot x'$, or more simply xx', for the concatenation of x and x', and x^r for the reverse of the string x.

Given a string $x \in \Sigma^m$, we indicate with $S(x) = (Q, \Sigma, \delta, I, F)$ the non-deterministic

suffix automaton with ϵ-transitions for the language $Suff(x)$, where $Q = \{I, q_0, q_1, \ldots, q_m\}$ is the set of automaton states, I is the initial state, $F = \{q_m\}$ is the set of final states and the transition function $\delta : \mathscr{P}(Q) \times (\Sigma \cup \{\epsilon\}) \longrightarrow \mathscr{P}(Q)$, where $\mathscr{P}(Q)$ is the power set of Q.

Specifically, for any $Q' \subseteq Q$ we have $q_{i+1} \in \delta(Q', c)$ if $q_i \in Q'$ and $c = x[i]$, for $0 \leq i < m$. In addition we have $\delta(\{I\}, \varepsilon) = Q$, while $\delta(Q', c) = \emptyset$ in all other cases. For simplicity, in what follows, we will use the notation $\delta(q, c)$ instead of $\delta(\{q\}, c)$.

The valid configurations $\delta^*(w)$ which are reachable by the automaton $S(x)$ on input $w \in \Sigma^*$ and starting from the initial state I are defined recursively as follows

$$\delta^*(w) := \begin{cases} \{q_0, q_1, ..., q_m\} & \text{if } w = \epsilon, \\ \bigcup_{q' \in \delta^*(I, w')} \delta(q', c) & \text{if } w = w'c, \text{for some } c \in \Sigma, \text{and } w' \in \Sigma^*. \end{cases}$$

Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|", the `left shift` "$\ll$" operator (which shifts to the left its first argument by a number of bits equal to its second argument), and the unary bitwise `not` operator "$\sim$".

2.0.2 Factor based searching

All the algorithms introduced in this thesis, like almost any algorithm representing the baseline for the comparisons conducted, inherit their search strategy from the BDM algorithm. In the BDM algorithm, the text is processed from left to right by sliding a window of size equal to pattern m. At each iteration, the algorithm inspects the characters of the current text window one by one, from right to left, and feeds each of them into the suffix automaton structure which has been constructed on the reverse of input pattern x^r at preprocessing time. The suffix automaton is used to keep track of the longest suffix u of the current window which also appears in the input pattern x. Backward scanning stops when either one of the following conditions holds:

- the window has been fully scanned: in this case an exact match of the input pattern is reported, and the window is shifted by one to the right (Figure 2.1);
- assuming u was the longest factor of the pattern recognized, if the current character c is such that factor cu doesn't appear in x, then the window is shifter in order to align the beginning of the window with the starting position of factor u in the text (Figure 2.2).

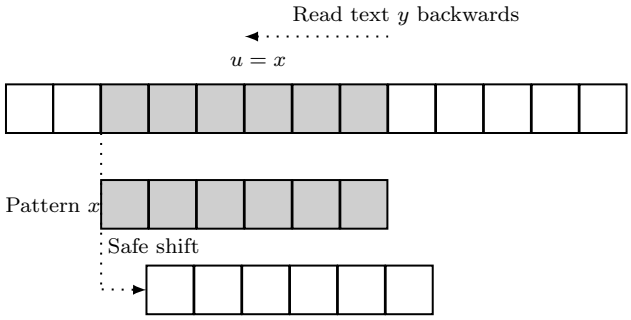

Figure 2.1: A factor u is read backwards until a complete match of the pattern x occurs: the window is advanced by one character.

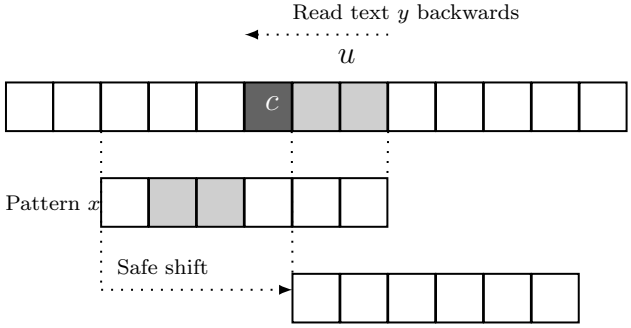

Figure 2.2: A factor u is read backwards until a non-factor at c occurs: it is then safe to shift the pattern past σ.

While the BDM algorithm relies on the suffix automaton of the reverse of the pattern x^r to filter each candidate occurrence, this approach can be easily extended to the case of filtration based algorithm, which make use of a weaker structure to recognize a broader language. This means that each occurrence of the pattern must be additionally verified (using a naive comparison) before reporting its occurrence,

and that the length of the computed shifts can be shorter, in some cases. If the filtration method is accurate enough and its simulation efficient, then the impact of the mentioned drawbacks can be considered not relevant.

2.0.3 Condensed Alphabets

Most of the algorithms presented in this paper are based on condensed alphabets, a technique which have been extensively used in the string matching field to enlarge the size of the input alphabet [13, 10]: in the case of filtration-based algorithms, this usually results in longer shifts and improved search speed. It consists in combining groups of q characters of the original alphabet, for a fixed value q. A hash function $\textsc{Hash} : \Sigma^q \leftarrow \{0, \ldots, 2^\alpha - 1\}$, is used for combining group of adjacent characters, where α is a positive fixed constant. Thus a new condensed pattern x_q of length $m - q + 1$, over the alphabet $\{0, \ldots, 2^\alpha - 1\}$, is obtained from x. Specifically we have:

$$x_q[i\,..\,j] = \textsc{Hash}(x[i \ldots i+q-1]) \cdot \textsc{Hash}(x[i+1 \ldots i+q]) \cdots \textsc{Hash}(x[j \ldots j+q-1]),$$

for $0 \leq i, j \leq m - q$, where $x_q = x_q[0\,..\,m - q]$. Searching is thus implicitly* performed on x_q rather than on the original string x.

The size 2^α of the new condensed alphabet depends on the available memory and on the size of the original alphabet Σ. An efficient method for computing a condensed alphabet was introduced by Wu and Manber [22]. It computes the shift value by using a shift-and-addition procedure and in particular

$$\textsc{Hash}(c_1, c_2, \ldots, c_q) = \left(\sum_{i=1}^{q} (c_i \ll (sh \cdot (q - i))) \right) \mod 2^\alpha$$

where $c_i \in \Sigma$ for $i = 1, \ldots, q$. The value of the shift sh depends on the values of α and q. Since the vast majority of string matching algorithms have an $\mathcal{O}(|\Sigma|)$ space complexity, practical choices of α doesn't exceed 16: greater values would lead to

*The q-gram representation of x and y is computed on the fly, since preprocessing the text would be too expensive.

prohibitive space costs. This also implies a maximum shift value of 16, meaning that q must be chosen to be less than 16 too. Typically, best results in practice are observed when $3 \leq q \leq 7$ and $sh = \{1, 2\}$ (see [13] for instance).

3

PBNDM: a sampled bit-parallel suffix automata for large strings

This section presents a new algorithm for the online exact string matching problem based on a suffix automaton constructed over an approximate version of the pattern x, which we simply call *pruned pattern*, where some specifically selected characters are replaced with don't care symbols. It turns out that the pruned pattern admits a succinct encoding of the bit-parallel suffix automaton which results in improved search speed as the length of the pattern increases. While the algorithm is still based on the bit-parallelism approach, it represents a starting point for the more efficient algorithms introduced in the upcoming chapters.

3.1 The Pruned BNDM Algorithm

As we pointed out in Section 1.1, the efficiency of an algorithm simulating the non-deterministic suffix automaton by bit-parallelism is influenced by the length of the pattern and by the size of the resulting automaton:

- on the one hand the performance of solutions based on bit-parallelism degrade

as m grows despite the use of a condensed alphabet. This is due to the need of representing the whole automaton using a single computer word of size w or to divide the computation on m/w different words;

- on the other hand, automata constructed over longer patterns should lead to larger shifts during the searching phase when a backward scan of the window is performed.

Thus the need for efficient bit-parallel encoding able to keep as low as possible the number of words involved in the encoding and able to preserve, at the same time, the length of the pattern.

In the next section, we present a new algorithm for the online exact string matching problem based on a suffix automaton constructed over an approximate version of the pattern x, which we simply call *pruned pattern*, where some specifically selected characters are replaced with don't care symbols. We will then show how to efficiently simulate the suffix automaton constructed over the pruned version of the pattern using bit-parallelism.

3.1.1 The Pruned Version of a Pattern

Let x be a pattern of size m and let T be a text of size n, both strings over a common alphabet Σ of size σ. In addition let $c \in \Sigma$ be a character of the alphabet occurring in x which we refer as the *pivot* character. A *pruned version* of x over the pivot character c is a string x_c obtained by preserving in x all the occurrences of c, while the remaining positions are allowed to match any character belonging to $\Sigma \setminus \{c\}$. In other words the pruned pattern x_c is obtained from x by replacing any character in $\Sigma \setminus \{c\}$ by a don't care symbol. More formally, for each $c \in \Sigma$, the pruned string x_c is a string of length m defined over the alphabet $\Sigma_c = \{c, \star\}$ where, for $i = 0, 1, .., m-1$, $x_c[i]$ is set to:

$$x_c[i] = \begin{cases} c & \text{if } x[i] = c \\ \star & \text{otherwise} \end{cases}$$

Example 1. *For instance, if we assume that* $x = abbacbbcac$ *is a pattern of length* $m = 10$ *over the alphabet* $\Sigma = \{a, b, c\}$ *and that* a *is the pivot character, then we have that* $x_a = a \star\star a \star\star\star\star a\star$*, where* $\star$ *is the don't care symbol. Similarly we have* $x_b = \star bb \star\star bb \star\star\star$.

The string matching problem allowing for don't care symbols is a well known approximate variant of the exact matching problem [23, 24], also known as string matching on indeterminate strings. It is also well known that the bit parallel simulation of the suffix automaton of an indeterminate pattern can be easily constructed by allowing states corresponding to don't care characters to be activated by any character in the alphabet [11]. However the resulting automaton has a number of states equal to the number of characters in the pattern, inheriting the same problems as any other solution based on this technique.

Here we show how the suffix automaton of a pruned pattern can be simulated using a number of bits proportional to the occurrences of the pivot character, leading to a filtration algorithm which may be particularly efficient for very long patterns.

Specifically, let $\rho(c)$ be the absolute frequency of the pivot character c in x. Then the pruned string x_c of a string x can be encoded as a sequence, $\langle d_0, d_1, ..., d_{\rho(c)}\rangle$, of length $\rho(c)+1$ over the alphabet $\Sigma_x = \{0, 1, 2, \ldots, m-1\}$, where each element of x_c represents the number of consecutive $\star$ symbols between two successive occurrences of the pivot character c, or located at the two extremities of the string.

More formally, let $\langle p_0, p_1, \ldots, p_{\rho(c)-1}\rangle$ be the sequence of all positions in x where the pivot character occurs, with $0 \leq p_1$, $p_{\rho(c)-1} < m$ and $p_{i-1} < p_i$, for $0 < i < \rho(c)$. Then we have, for $0 \leq i \leq \rho(c)$:

$$d_i = \begin{cases} p_0 & \text{if } i = 0 \\ p_{i+1} - p_i - 1 & \text{if } 0 < i < \rho(c) \\ m - p_{\rho(c)-1} - 1 & \text{if } i = \rho(c) \end{cases}$$

We refer to such a representation of the pruned string x_c as its *implicit encoding* and we denote it as $\hat{x}_c$ (see Figure 3.1). It trivially turns out that $d_i < m$ for $0 \leq i \leq k$. More precisely we have

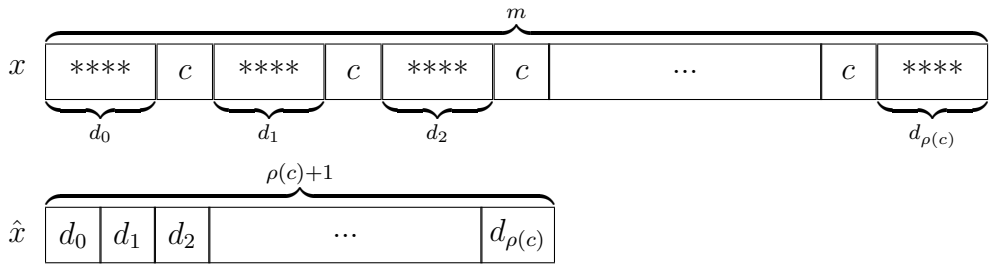

Figure 3.1: The pruned version x_c, for a given pattern x, over a generic character c, and its implicit encoding. Here $\rho(c)$ is the absolute frequency of the pivot character c in x. Then the pruned string x_c of a string x is encoded as a sequence, $\langle d_0, d_1, ..., d_{\rho(c)} \rangle$, of length $\rho(c)+1$ over the alphabet $\Sigma_x = \{0, 1, 2, \ldots, m-1\}$, where each element of x_c represents the number of consecutive $\star$ symbols between two successive occurrences of the pivot character c, or located at the two extremities of the string.

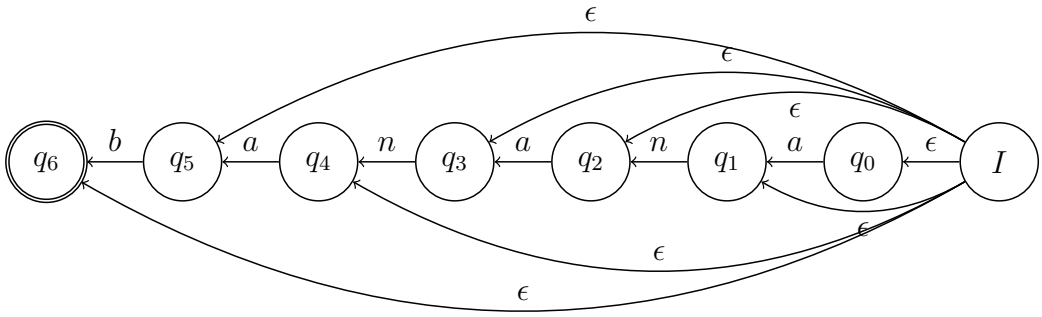

Figure 3.2: Suffix automaton for the reverse of the pattern string $x = banana$.

$$m = \rho(c) + \sum_{i=0}^{\rho(c)} d_i$$

Example 2. *For instance, consider the pattern string* $x =$ `banana`*, whose automaton is depicted in Figure 3.2. Then we have* $x_a =$ `*a*a*a`*,* $x_b =$ `b*****`*,* $x_n =$ `**n*n*`*, while* $\hat{x}_a = \langle 1, 1, 1, 0 \rangle$*,* $\hat{x}_b = \langle 0, 5 \rangle$ *and* $\hat{x}_n = \langle 2, 1, 1 \rangle$*.*

In the next sections we describe the preprocessing and the searching phases of our new algorithm, which we call Pruned BNDM (PBNDM) algorithm, and which solves the exact string matching problem by making use of the suffix automaton of

the pruned pattern.

3.1.2 The Preprocessing Phase

During the preprocessing phase of the PBNDM algorithm, a character c occurring in x is elected to be the pivot character. Since such choice is arbitrary, the pivot character is selected as the character with maximum absolute frequency not exceeding the word size w, if any. If such choice is not possible we truncate the pattern at its longest prefix that contains at least one character with an absolute frequency not exceeding the word size w. It is easy to observe that the selection of the pivot character can be performed in $\mathcal{O}(m)$ time and $\mathcal{O}(|\Sigma|)$ space, by computing the frequencies of all the characters appearing in x. Without loss in generality we can assume that such pivot character can be selected on the pattern x.

Let $\hat{x}_c = \langle d_0, d_1, ..., d_{\rho(c)} \rangle$ be the implicit encoding of x_c. It is a string of length $\rho(c) + 1$ over the alphabet $\hat{\Sigma} = \{0, 1 \ldots, m-1\}$ of size m.

The bit-parallel representation of the suffix automaton of $\hat{x}_c^r$ is computed by means of an array B of m bit-vectors, each of size $\rho(c) + 1$, where the i-th bit of $B[d]$ is set as follows:

$$B[d][i] = \begin{cases} 1 & \text{if } (i = 0 \text{ and } d \geq d_0) \text{ or } (i > 0 \text{ and } d = d_i), \\ 0 & \text{otherwise,} \end{cases}$$

for $0 \leq i \leq \rho(c)$ and $0 \leq d < m$.

A separate discussion should be made for the first transition made on the automaton. Since it is admitted that the first transition can start from any position of the pattern it is necessary to allow that at the first transition each i-th state can be activated by values lower than or equal to d_i. For this purpose an auxiliary set of m bit-vectors is defined, called S, which is used for the simulation of the first transition on the automaton.

More formally, for $0 \leq i \leq \rho(c)$ and $0 \leq d < m$, $S[d][i]$ is defined as:

$$S[d][i] = \begin{cases} 1 & \text{if } (i = 0 \text{ and } d \geq d_0) \text{ or } (i > 0 \text{ and } d \leq d_i), \\ 0 & \text{otherwise} \end{cases}$$

For instance, let us consider the pruned pattern $x_{\mathtt{n}} = \mathtt{**n*n*}$ of the pattern string $x = \mathtt{banana}$ of length $m = 6$. Remembering that $\hat{x}_{\mathtt{n}} = \langle 2, 1, 1 \rangle$, then we have:

$B[0] = 000$	$B[3] = 100$	$S[0] = 011$	$S[3] = 100$
$B[1] = 011$	$B[4] = 100$	$S[1] = 011$	$S[4] = 100$
$B[2] = 100$	$B[5] = 100$	$S[2] = 100$	$S[5] = 100$

The pseudo-code of the preprocessing phase of the algorithm is shown in Figure 3.3. It makes use of two auxiliary procedures: SelectChar and Read. Procedure SelectChar is responsible for selecting the pivot character; procedure Read performs a scan of the string S starting at position $j = i$ and proceeds from right to left until a given position $l \leq j$ is reached or an occurrence of the pivot character c is found. It returns the pair of integers $(i - j, j)$.

The implicit encoding of x_c is computed gradually, during the initialization of table B through procedure Read, which computes the next element of the implicit encoding of x_c. Table S is then computed from table B in a single-pass for loop. The time complexity of the preprocessing phase is $\mathcal{O}(m)$. Since $d_i \leq m$, then the space overhead to store B and S is $\mathcal{O}(m)$ words too. Apart from the tables encoding the transitions of the suffix automaton, B and S, the preprocessing phase returns two additional integers, d_0 and $d_{max} = \max\{d_i \ : \ 0 \leq i \leq \rho(c)\}$, which are used during the searching phase.

3.1.3 The Searching Phase

The searching phase of the PBNDM algorithm is shown in Figure 3.4. It acts using a filtering method: it first searches for all the occurrences of the pruned pattern x_c in the text and when an occurrence of x_c is found, starting at position j of the text, the algorithm naively checks for the whole occurrence of the pattern, i.e. it checks if $x = y[j..j + m - 1]$.

```
READ(S, i, l, c)
1.  j ← i
2.  while j ≥ l and S[j] ≠ c do
4.      j ← j − 1
6.  return (i − j, j)

PREPROCESS(x, m, c, k)
    Δ Initialize bit-vectors B and S
1.  for i ← 0 to m do
2.      B[i] ← 0
3.      S[i] ← 0
    Δ Compute B, d_0 and d_max
4.  s ← 1
5.  d_0 ← −1
6.  d_max ← −1
7.  i ← m − 1
8.  while i ≥ 0 do
9.      (d, i) ← Read(x, i, 0, c)
10.     B[d] ← B[d] | s
11.     s ← s ≪ 1
12.     i ← i − 1
13.     if d_0 = −1 then d_0 ← d
14.     d_max ← max(d, d_max)
    Δ Compute the bit-vectors S
15. for i ← m − 2 downto 0 do
16.     S[i] ← B[i] | S[i + 1]
    Δ Set first bits for d ≥ d_0
17. for i ← d_0 to d_max do
18.     B[i] ← B[i] | 10^{k−1}
19.     S[i] ← S[i] | 10^{k−1}
20. return (B, S, d_0, d_max)
```

Figure 3.3: The pseudo-codes of the auxiliary procedures used in the PBNDM algorithm. Procedure READ performs a scan of the string S starting at position i and proceeding from right to left until a given position $l \leq i$ is reached or an occurrence of the pivot character c is found. Procedure PREPROCESS encodes the bit-parallel representation of the suffix automaton of $\hat{x}_c^r$.

As in the original BNDM algorithm, a window w of length m is shifted over the text, starting from the left end of the text and sliding from left to right. At each iteration of the algorithm a position of the window w is attempted performing a scanning of its characters proceeding from right to left and performing the transitions over the automaton accordingly.

During the backward scanning, $\hat{w}_c = \langle w_0, w_1, ..., w_l \rangle$ is computed on the fly by

```
PBNDM(x, m, y, n)
 1.  (c, k) ← SelectChar(x, m)
 2.  (B, S, d_0, d_max) ← Preprocess(x, m, c, k)
 3.  j ← 0
 4.  while j ≤ n − m do
 5.      prfx ← 0
 6.      D ← 1^k
 7.      i ← m − 1
 8.      l ← Max(i − d_max, 0)
 9.      (w, i) ← Read(x, i, l, c)
10.      if w > d_max then
11.          j ← j + m − d_0
12.          continue
13.      D ← D & S[w]
14.      if D & 1^{k−1} do
15.          prfx ← w + 1
16.          if prfx = m then
17.              if x = y[j..j + m − 1] then
18.                  output j
19.              prfx ← m − 1
20.      D ← D ≪ 1
21.      s ← w + 1
22.      while i ≥ 0 and D ≠ 0^k do
23.          i ← i − 1
24.          l ← Max(i − d_max, 0)
25.          (w, i) ← Read(x, i, l, c)
26.          D ← D & B[w]
27.          if D & 1^{k−1} do
28.              prfx ← d_0 + s
29.              if prfx = m then
30.                  if x = y[j..j + m − 1] then
31.                      output j
32.                  prfx ← m − 1
33.          s ← s + w + 1
34.          D ← D ≪ 1
35.      j ← j + m − prfx
```

Figure 3.4: The pseudo-code of the Pruned-BNDM (PBNDM) algorithm.

procedure READ, and the automaton configurations s_i, represented as a bit-vector D of $\rho(c) + 1$ bits, are updated accordingly.

If $w_l > d_{max}$, then the prefix of x_c of size d_0 has been recognized, so the window is shifted without performing any transition. Otherwise, the first transition is performed by setting $D \leftarrow D \,\&\, S[w_l]$, so that all states s_i such that $d_i \geq w_l$ are kept active. Transition on any subsequent w_i, for $0 < i \leq l$, is implemented as

$D \leftarrow D \,\&\, B[w_i]$. Moreover, by the definition of B and S, s_0 is kept active during the i-th transition if $w_i \geq d_0$. When, after performing the i-th transition, state s_0 is active, then a prefix of x_c of size $d_0 + \sum_{j=i+1}^{k} w_j$ has been recognized.

Apart from the case where x_c is recognized, each attempt ends when either D becomes zero or it is established that $w_i > d_{max}$ while proceeding in the backward scan.

As the original BNDM algorithm, the PBNDM algorithm has a $\mathcal{O}(nm)$ worst case time complexity and a $\mathcal{O}(\sigma + m)$ space complexity.

3.2 Experimental comparison

In this section, we compare the new algorithm against other suffix automaton based solutions, focusing on those which make use of bit-parallelism for solving the problem with long strings. In particular we included the following algorithms:

- BNDM: the Backward-Nondeterministic-DAWG-Matching algorithm [12];
- SBNDM: the Simplified BNDM algorithm [7];
- LBNDM: the Long BNDM algorithm [7];
- BSX: the BNDM algorithm [12] with Extended Shift [8];
- FBNDM: the Factorized variant [25] of the BNDM algorithm [12];
- PBNDM: our PBNDM algorithm, presented in Section 3.1;

All algorithms have been implemented in the C programming language and have been tested using the Smart tool [26]. Experiments have been executed locally on a computer running Linux Ubuntu 20.04.1 with an Intel Core i5 3.40 GHz processor and 8GB RAM. Our tests have been run on a genome sequence, a protein sequence, and an English text, each of size 10MB. Such sequences are provided by the Smart research tool and are available online for download (additional details on the sequences can be found in Faro et al. [26]). In our implementations the value of the

Average shifts on a genome sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	29	29	29	29	29	29	29	29	29	29	29
SBNDM	30	30	30	30	30	30	30	30	30	30	30
LBNDM	61	112	106	27	32	64	128	**256**	**512**	**1024**	**2048**
BXS	59	117	**219**	1	1	1	1	1	1	1	1
FBNDM	**62**	66	67	67	66	67	67	67	67	67	67
PBNDM	60	**123**	142	**139**	**137**	**130**	**130**	125	125	122	122
Gain	-3%	+5%	-35%	+107%	+107%	+94%	+1%	-51%	-75%	-88%	-94%

Average shifts on a protein sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	31	31	31	31	31	31	31	31	31	31	31
SBNDM	31	31	31	31	31	31	31	31	31	31	31
LBNDM	**63**	**126**	248	470	713	353	206	286	526	1027	2048
BXS	62	125	**252**	**502**	**984**	1710	1635	1	1	1	1
FBNDM	**63**	**126**	146	143	141	145	144	143	144	145	144
PBNDM	56	118	244	492	979	**1928**	**3022**	**3015**	**2942**	**2910**	**2871**
Gain	-11%	-6%	-3%	-2%	-1%	+13%	+85%	+954%	+459%	+183%	+41%

Average shifts on a natural language sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	30	30	30	30	30	30	30	30	30	30	30
SBNDM	31	31	31	31	31	31	31	31	31	31	31
LBNDM	**63**	126	247	471	824	1035	797	702	910	1467	2609
BXS	62	125	**252**	**505**	**1010**	**2008**	3910	7076	10410	11015	8533
FBNDM	**63**	**127**	156	157	156	156	156	155	156	156	157
PBNDM	58	122	245	493	982	1970	**3940**	**7784**	**15438**	**30706**	**60803**
Gain	-7%	-4%	-2%	-2%	-3%	-2%	+1%	+10%	+48%	+178%	+613%

Table 3.1: Average shifts achieved by bit-parallel algorithms on a genome sequence (on top), a protein sequence (in the middle) and natural language text (on bottom). The gain rows report the percentage of deviation with respect to the first or second best time.

word size* has been fixed to $w = 32$ and patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 6 \leq i \leq 16\}$. For each length, the mean over the running times of 500 runs (expressed in Gigabytes per second) and the average shift advancements has been computed.

3.3 Chapter summary

In this chapter, we introduced PBNDM, a new algorithm based on a novel encoding of the suffix automaton of a string, suitable for patterns exceeding the word size w. The algorithm is based on a pruned version of the pattern whose automaton can be encoded in an implicit form using few bits. Experimental results suggest that

*The value of the word size has been chosen in order to better emphasize scaling problems of the several bit-parallel algorithms.

Experimental results on a genome sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	1.78	1.79	1.78	1.81	1.78	1.80	1.76	1.76	1.76	1.74	1.73
SBNDM	1.86	1.82	1.88	1.88	1.84	1.86	1.86	1.85	1.84	1.84	1.82
LBNDM	1.74	1.91	0.90	0.20	0.20	0.22	0.23	0.24	0.25	0.25	0.25
BXS	1.44	1.67	1.31	-	-	-	-	-	-	-	-
FBNDM	**2.21**	**2.29**	**2.28**	**2.26**	**2.29**	**2.27**	**2.28**	**2.25**	**2.26**	**2.22**	**2.21**
PBNDM	1.26	1.83	1.92	1.90	1.89	1.86	1.88	1.83	1.78	1.82	1.82

Experimental results on a protein sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	2.24	2.23	2.21	2.23	2.17	2.21	2.19	2.19	2.21	2.21	2.16
SBNDM	2.33	2.31	2.36	2.31	2.30	2.34	2.33	2.29	2.34	2.30	2.26
LBNDM	2.34	2.56	**2.70**	**2.81**	2.63	1.32	0.60	0.47	0.45	0.46	0.47
BXS	2.18	2.36	2.61	2.52	2.49	2.16	-	-	-	-	-
FBNDM	**2.48**	**2.71**	2.67	2.71	**2.68**	2.68	2.67	2.67	2.68	2.65	2.38
PBNDM	1.27	1.69	2.18	2.47	2.56	**2.70**	**2.68**	**2.68**	**2.69**	**2.66**	**2.63**

Experimental results on a natural language text

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	1.87	1.88	1.89	1.89	1.86	1.88	1.86	1.90	1.87	1.88	1.82
SBNDM	1.94	1.95	1.93	1.95	1.92	1.94	1.95	1.92	1.91	1.94	1.90
LBNDM	2.15	2.44	**2.65**	**2.81**	**2.77**	2.52	1.71	1.05	0.77	0.65	0.58
BXS	1.91	2.25	2.54	2.54	2.74	**2.84**	2.70	2.50	0.95	0.29	-
FBNDM	**2.28**	**2.60**	2.57	2.56	2.60	2.61	2.58	2.60	2.53	2.57	2.54
PBNDM	1.07	1.59	2.14	2.38	2.45	2.76	**2.84**	**3.64**	**4.93**	**6.18**	**7.40**

Table 3.2: Experimental results on a genome sequence (on top), a protein sequence (in the middle) and a natural language text (on bottom). Searching speed is reported in GB/s. Best results have been bold faced.

the new solution is competitive when compared for searching long strings against existing bit-parallel algorithms. It is also worth noticing that, despite the algorithm still suffers in the case of small alphabets due to the reduced length of the shifts, the new encoding is flexible enough to be applied in all those solutions that make use of such data structure, even in the case of non-standard and approximate pattern matching.

4

The Range Automaton: An Efficient Approach to Text-Searching

In this chapter we present the *Range Automaton*, a weak yet efficient variant of the non-deterministic suffix automaton of a string whose configuration can be encoded in a very simple form and which is particularly suitable to be used for solving a multitude of text-searching problems. We will firstly model the approach in the case of exact string matching and present an efficient algorithm, named Backward Range Automaton Matcher, which turns out to be very fast in many practical cases. Later, we will show how the Range Automaton can be adapted in an effective way also to non-standard string matching problems such as *swap matching* and *multiple string matching* and *order preserving pattern matching*. Experimental results suggest that the new approach is flexible and effective for all three search problems addressed.

4.1 The Range Automaton

Let x be a string of length m over the alphabet Σ. The *Range Automaton* of a pattern x is a *weaker* version of the non-deterministic Suffix Automaton of x in the

sense that, while using an encoding that can allow to keep track of the set of all active states of the automaton, it adopts a *weak* transition approach, meaning that also transitions not tagged with the current character may be activated.

Despite this weak transition approach, the Range Automaton has the interesting feature of operating as an Oracle: the recognized language contains all the factors of x and (possibly) other strings as well. This is the price to pay for an automaton that can allow a simpler encoding and a more efficient simulation.

Before entering into the details of the description of the Range Automaton it is advisable that some useful notions are introduced.

We define the *position function*, $\rho : \Sigma \longrightarrow \mathscr{P}(\{0, 1, \ldots, m-1\})$, as the function which maps each character $c \in \Sigma$ to the set of positions where c occurs in x. If c doesn't occur in x we agree to set $\rho(c) = \emptyset$. More formally, $\rho(c) := \{i \mid P[i] = c, 0 \leq i < m\}$, for each $c \in \Sigma$. Particularly important for our discussion is the following definition of a range-set.

Definition 1 (Range-Set)**.** *Given a string x of length m and a termination symbol $\$ \notin \Sigma$, a range-set of x is a set of contiguous positions in the string $x\$$. We use the notation $[i : j]$ to denote the range-set of positions in x from i to j, extremes included. Formally $[i : j] = \{i, i+1, \ldots, j\}$, where $0 \leq i \leq j \leq m$.*

The symbol \$ is concatenated at the end of x in order to extend its length of one character and allow the value m to be included in any range-set.

We denote by $\mathcal{R}^m$ the set of all possible range-sets associated to a given string on length m. Formally

$$\mathcal{R}^m = \{[i : j] \mid 0 \leq i \leq j \leq m\}.$$

We also define the *range function*, denoted by $r : \Sigma \longrightarrow \mathcal{R}^m$, as the function which maps each character c to the tightest set-range where the character c occurs in the pattern. More formally, for $c \in \Sigma$, $r(c)$ is defined as follows.

$$r(c) = \begin{cases} [\min \rho(c) : \max \rho(c)] & \text{if } \rho(c) \neq \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

Example 3. *Given the pattern* $x = \mathsf{banana}$*, we have that* $r(\mathsf{a}) = [1:5] = \{1,2,3,4,5\}$*,* $r(\mathsf{b}) = [0:0] = \{0\}$ *and* $r(\mathsf{n}) = [2:4] = \{2,3,4\}$*, while* $r(c) = \emptyset$ *for any other character* c *not appearing in* x*.*

Given a range-set R, we denote by $R{\ll}k$ the *left shift operation* on R by k positions. The result of such shift operation is a new range-set obtained by decreasing each element of R by k. More formally, if $R = [i:j]$, we have:

$$R{\ll}k := \begin{cases} \emptyset & \text{if } R = \emptyset \text{ or } j < k, \\ \{0, 1, \ldots, j-k\} & \text{if } i < k \text{ and } j \geq k. \\ \{i-k, \ldots, j-k\} & \text{if } i \geq k. \end{cases}$$

Example 4. *Given a range-set* $R = [2:5] = \{2,3,4,5\}$ *of size* 4*, we have that* $R \ll 1 = [1:4] = \{1,2,3,4\}$ *and* $R \ll 2 = [0:3] = \{0,1,2,3\}$*. In addition we have also* $R \ll 4 = [0:1] = \{0,1\}$ *and* $R \ll 6 = \emptyset$*.*

We notice that a one-to-one correspondence can be defined between the states of the suffix automaton $S(x)$ and the positions within the string $x\$$. Consequently it is possible to map any range-set in $\mathcal{R}$ to a set of states in the suffix automaton. Formally we can map any position i to the state q_i, for $0 \leq i \leq m$, and any range-set $[i:j]$ to the set of states $\{q_i, q_{i+1}, \ldots, q_j\}$, for $0 \leq i \leq j \leq m$.

We are now ready to define the Range Automaton used in our approach. Using the correspondence between any range-set of the pattern x and the set of states in the suffix automaton of x, in the following definition we will deal with the sets of states as *range-sets*. In this context a configuration of the Range Automaton of x is maintained as a single range-set, which identifies the set of all active states of the automaton. In other words if $[i:j]$ is the range-set which represents the configuration of the Range Automaton, each state q_k, with $k \in [i:j]$, is an active state.

Definition 2 (The Range Automaton)**.** *Given a string $x \in \Sigma^m$, we denote with $A(x) = (Q, \Sigma, \gamma, I_r, F)$ the non-deterministic range suffix automaton of x. It is defined as follows:*

- $Q = [0 : m] = \{0, 1, \ldots, m\}$ *is the set of states of the automaton;*
- $I_r = [0 : m] = Q$ *is the set of initial states;*
- $\gamma : \mathcal{R}^m \times \Sigma \longrightarrow \mathcal{R}^m$ *is the transition function, where $\gamma(R, c)$ is defined as $\gamma(R, c) = (R \ll 1) \cap r(c)$, for any $R \in \mathcal{R}^m$ and $c \in \Sigma$;*
- $F = [0 : 0] = \{0\}$ *is the set of final states.*

The valid configurations $\gamma^*(I_r, W)$ which are reachable by the Range Automaton $A(x)$ on input $W \in \Sigma^*$, with $|W| = n$, are defined recursively as follows

$$\gamma^*(I_r, W) = \begin{cases} [0 : m] & \text{if } n = 0 \\ \gamma(\gamma^*(I_r, W[0\,..\,n-2]), W[n-1]) & \text{if } n > 0 \end{cases}$$

The following technical lemma allows to characterize the Range Automaton* as an oracle, proving that it recognizes (at least) all the factors of the pattern.

Lemma 1. *Let x be a string of length m and let $S(x)$ be the non-deterministic suffix automaton with ϵ-transitions for the language Suff(x). In addition let $A(x)$ be the Range Automaton of x. We have that if $q_i \in \delta^*(I, W)$, for a string $W \in \Sigma^*$, then $i \in \gamma^*(I_r, W)$.*

Proof. Proof Let W be a string of length n. We proceed by induction on n.

For the base case, we have $n = 0$, i.e. $W = \epsilon$. In this case $\delta^*(I, W) = \{q_0, q_1, \ldots, q_m\}$ and $\gamma^*(I_r, W) = [0 : m]$, so the lemma trivially holds.

*Note that the definition of the Range Automaton turns out to be *reversed* with respect to the definition of the suffix automaton, i.e. state i in the suffix automaton corresponds to state $m - i$ in the Range Automaton. This choice was made to make the definition of the Range Automaton consistent with the description of the algorithm presented in Section 4.2, which scans each text window starting from the last character.

Let now $n > 0$ and let us suppose that the lemma holds for every string of length $l \leq n-1$. Since $|W| = n > 0$ we can write $W = W'c$, with $W' \in \Sigma^{n-1}$. By inductive hypothesis, if $q_i \in \delta^*(I, W')$, then $i \in \gamma^*(I_r, W')$. Since $\gamma^*(I_r, W')$ is a range, then $[i' : j'] \subseteq \gamma^*(I_r, W')$, where i' and j' are, respectively, the minimum and the maximum of the set $\{i \mid q_i \in \delta^*(I, W')\}$. Remembering that $\delta^*(I, W) = \bigcup_{q' \in \delta^*(I,W')} \delta(q', c)$, we have that if $q_k \in \delta^*(I, W)$ then, by the definition of δ the following inequalities hold:

- $\max(0, i' - 1) \leq k \leq \max(0, j' - 1)$,
- $f \leq k \leq l$, where $[f : l] = r(c)$.

By the first inequality it follows that $k \in ([i' : j'] \ll 1)$. Since for the second inequality $k \in r(c)$, then $k \in ([i' : j'] \ll 1) \cap r(c)$. Moreover, we observe that, since $[i' : j'] \subseteq \gamma^*(I_r, W')$, then $([i' : j'] \ll 1) \cap r(c) \subseteq \gamma^*(I_r, W)$ holds. Thus, we can conclude that $k \in \gamma^*(I_r, W)$ too. □

The following Corollary allows to characterize the range automaton as a useful tool to search for a pattern in a text. It follows from Lemma 1.

Corollary 1. *Let x be a pattern of length m and let y be a text of length n. In addition let $A(x)$ be the Range Automaton of x. If the prefix $x[0\,..\,i]$ occurs in y at position j, i.e. $x[0\,..\,i] = y[j\,..\,j+i]$, then $0 \in \gamma^*(I, (y[j\,..\,j+i])^r)$.*

4.2 The Backward Range Automaton Matcher

In this section we describe the Backward Range Automaton Matcher (BRAM) designed for the exact online string matching problem and discuss its time and space complexity. In our presentation we will refer to the pseudo-code of the BRAM algorithm depicted in Figure 4.1.

As before, let x be a pattern of length m and let y be a text of length n, both strings defined over an alphabet Σ of size σ.

The preprocessing phase of the BRAM algorithm consists in the computations of the function $r(c)$, for each $c \in \Sigma$ (lines 1-6) by means of two simple for loops, taking time $\mathcal{O}(\sigma)$ and $\mathcal{O}(m)$, respectively. Thus, the preprocessing phase achieves an overall $\mathcal{O}(m + \sigma)$-time and $\mathcal{O}(\sigma)$-space complexity.

The searching phase of the algorithm proceeds along the same line of the BNDM algorithm, where the configuration of the Range Automaton is maintained as a range set R.

The algorithm works by sliding a window W of length m along the text starting form the left end of the text and proceeding from left to right. At the end of each attempt the window is shifted to the right by a given amount $s > 0$. This process continues until the right end of the text is reached.

Suppose we are in any of the attempts of the search phase, assuming that $W = y[j \,..\, j + m - 1]$. At the beginning of the attempt the configuration of the automaton is initialized to the set of initial states I_r. This is done by setting $R = [0 : m] = \{0, 1, \ldots, m\}$ (line 6). Thus all automaton states are active.

While proceeding in the backward scan of the window the configuration of the automaton is updated accordingly. Specifically, after reading a character c, the configuration of the automaton is updated by the following operation:

$$R \leftarrow (R \ll 1) \cap r(c),$$

which is always performed at the beginning of each iteration of the while cycle at line 8.

The algorithm keeps track of the length of the prefixes recognized by the Range Automaton during the backward scan by maintaining a variable x which is initialized to 0 at the beginning of each attempt. By Lemma 1, a prefix of x is recognized whenever $0 \in R$. When this condition occurs, the algorithm updates the length of the prefix just identified (line 12). This information will later be used to carry out the correct advancement of the window along the text (line 17).

The backward scan proceeds until R becomes empty, a condition which occurs

when the substring $y[j+m-i\,..\,j+m-1]$ is not recognized by the automaton and no state in the automaton is active. In this case the window is advanced in order to align the first character of x with the starting position of the last recognized prefix.

However, observe that $R = \emptyset$ can occur also when exactly m characters have been scanned. If such condition occurs (line 10) then a candidate occurrence of the pattern has been located and a naive check is performed to verify the occurrence of the whole pattern starting from position j of the text.

Regarding the space and time complexity of the resulting algorithm it is straightforward to observe that the searching phase of the BRAM algorithm runs in $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space.

Example 5. *Let* $x =$ `banana` *be a pattern of length* 6 *and assume* $W =$ `anaban` *is the current window of the text. Plainly we have* $r(\texttt{a}) = [1:5]$, $r(\texttt{b}) = [0:0]$ *and* $r(\texttt{n}) = [2:4]$. *The following table shows the configurations of the Range Automaton obtained during the backward scan of the string* W.

Iteration	*Operation*	*Range-Set Computation*	*Configuration*
iteration 0.	*initial state*	$R_0 = [0:5]$	`[b a n a n a]`
iteration 1.	*read* `n`	$R_1 = [0:4] \cap [2:4] = [2:4]$	`b a[n a n]a`
iteration 2.	*read* `a`	$R_2 = [1:3] \cap [1:5] = [1:3]$	`b[a n a]n a`
iteration 3.	*read* `b`	$R_3 = [0:2] \cap [0:0] = [0:0]$	`[b]a n a n a`
iteration 4.		$R_4 = \emptyset$	

4.2.1 Speeding-up Searching by Condensed Alphabets

For the sake of simplicity, algorithm 4.2 has been described assuming that characters are processed one by one. However, practical implementations of the algorithm strongly rely on the condensed alphabet expansion presented in Section 2.0.3.

Fig. 4.8, reported in Section 4.6, shows experimental evaluations to compare the performances of the BRAM algorithm under various conditions and for different values of the parameter q (we postpone the reader to Section 4.6 for a detailed description of the experimental settings).

```
PREPROCESSING(x, m)
 1.  for each c ∈ Σ do
 2.      r(c) ← ∅
 3.  for c ∈ x do
 4.      i ← min ρ(c)
 5.      j ← max ρ(c)
 6.      r(c) ← [i : j]
 7.  return r

VERIFY(x, m, y, j)
 1.  for i = 0 to m − 1 do
 2.    if x[i] ≠ y[j + i] then
 3.      return False
 4.  return True

BRAM(x, m, y, n)
 1.  r ← Preprocessing(x, m)
 2.  j ← 0
 3.  while j ≤ n − m do
 4.      p ← 0
 5.      i ← m
 6.      R ← [0 : m]
 7.      do
 8.          R ← (R ≪ 1) ∩ r(y[j + i − 1])
 9.          i ← i − 1
10.          if (0 ∈ R) then
11.              if (i > 0) then
12.                  p ← m − i
13.              else
14.                  if Verify(x, m, y, j) then
15.                      Output(j)
16.      while (R ≠ ∅)
17.      j ← j + m − p
```

Figure 4.1: The pseudocode of the BRAM algorithm and its auxiliary procedures.

From experimental evaluations shown in Fig. 4.8, it turns out that the performances of the algorithm strongly depend on the values of m, q and σ. When the size of the alphabet is small then larger values of the parameter q are more efficient.

Such difference is less sensible when the size of the alphabet gets larger. However it turns out that the smaller is the length of the pattern the lower is the performance of the algorithm. This behavior is more evident for larger values of the parameter q. Thus, the choice of the parameter q should be directed to larger values when the size of alphabet decreases or when the length of the pattern increases.

4.3 Extensions to Non-Standard Matching Problems

In this section we will show that our basic algorithm is flexible enough to be adapted to some other non-conventional pattern matching problems with only minor modifications. Indeed, to derive each new solution, we will only modify the implementation of the PREPROCESSING and VERIFY procedures, leaving unchanged the main structure of the BRAM algorithm.

Obviously, this generalization is not feasible for any text searching problem, but the two examples that we report in this paper show that this is true for at least a certain class of search problems. Adaptability to other problems not belonging to this class should be assessed on a case-by-case basis. However, the classification of such problems is not among the aims of this paper.

4.3.1 Extension to Swap matching

The *string matching problem with swaps* (*swap matching* problem, for short) is a well-studied variant of the classic string matching problem, and was introduced for the first time in 1995 as one of the open problems in non-standard string matching [27]. Many solutions, mostly focused on the practical aspects of the problem, have been presented in the last few years (see for instance [28, 29, 30, 31]).

The swap matching problem consists in finding all occurrences, up to character swaps, of a pattern x of length m in a text y of length n, with x and y sequences of characters drawn from a same finite alphabet Σ of size σ. More precisely, the pattern is said to *swap-match the text at a given location* j if adjacent pattern characters

can be swapped, if necessary, so as to make it identical to the substring of the text starting (or, equivalently, ending) at location j. All swaps are constrained to be disjoint, i.e., each character can be involved in at most one swap. Moreover, we make the agreement that identical adjacent characters are not allowed to be swapped. Definitions 3 and 4 formally define the problem.

Definition 3. *A* swap permutation *for a string x of length m is a permutation $\pi : \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$ such that:*

(a) if $\pi(i) = j$ then $\pi(j) = i$
(characters at positions i and j are swapped);

(b) for all i, $\pi(i) \in \{i-1, i, i+1\}$
(only adjacent characters can be swapped);

(c) if $\pi(i) \neq i$ then $x[\pi(i)] \neq x[i]$
(identical characters can not be swapped).

For a given string x and a swap permutation π for x, we write $\pi(x)$ to denote its *swapped version*, namely $\pi(x) = x[\pi(0)] \cdot x[\pi(1)] \cdots x[\pi(m-1)]$.

Definition 4. *Given a text y of length n and a pattern y of length m, x is said to* swap-match *(or to have a* swapped occurrence*)* at location $j \geq m-1$ of y *if there exists a swap permutation π of x such that $\pi(x)$ matches y at location j, i.e., $\pi(x) = y[j-m+1 \,..\, j]$.*

Definition 4 immediately leads to the following implementation of the VERIFY procedure shown in Figure 4.2, which is able to check if a pattern x performs a swap match beginning at position j of the text y.

It simply checks, for each pattern position j, whether character $x[j]$ matches the corresponding text position $y[i+j]$. In the case of a mismatch, it additionally checks if text characters $y[i+j]$ and $y[i+j+1]$ are swapped with respect to the corresponding pattern characters $x[j]$ and $x[j+1]$ before returning a mismatch for

```
VERIFY(x, m, y, j)
 1.  i ← 0
 2.  while (i < m) do
 3.    if (x[i] = y[j + i]) then i ← i + 1
 4.    else
 5.      if (i < m − 1 and
 6.            x[i] = y[j + i + 1] and
 7.            x[i + 1] = y[j + i])
 8.        then j ← j + 2
 9.        else return False
10.  return True
```

Figure 4.2: The pseudo-code of VERIFY procedure adapted to swap matching.

the current alignment. It is straightforward to observe that such verification can be done in $O(m)$ time.

4.3.2 BRAM for Swap Matching

In order to adapt the approach based on the Range Automaton to the approximate case of swap matching, we generate a superimposed pattern where each position corresponds to the set of all the characters contained in any swap permutation of the pattern at the same position. Specifically, given an input pattern x of length m, we generate a superimposed pattern $\hat{x}$ of length m where the element $\hat{x}[i]$, for $0 \leq i < m$, is a set of characters. A character $c \in \Sigma$ is contained in the set $\hat{x}[i]$ if there is a swap permutation π of x such that $\pi(x)[i] = c$. Since a character can only be involved in a single swap and characters at the end of a string can only swap with the internal characters, then the set at position i of the superimposed pattern $\hat{x}$ can be formally defined as follows:

$$\hat{x}[i] = \begin{cases} \{x[i]\} & \text{if } i = 0 \text{ and } i = m-1 \\ \{x[i], x[i+1]\} & \text{if } i = 0 \text{ and } i < m-1 \\ \{x[i-1], x[i]\} & \text{if } i > 0 \text{ and } i = m-1 \\ \{x[i-1], x[i], x[i+1]\} & \text{if } i > 0 \text{ and } i < m-1 \end{cases}$$

```
PREPROCESSING(x, m)
 1. for each c ∈ Σ do r(c) ← ∅
 2. for i ← 0 to m − 1 do ρ(x[i]) ← ρ(x[i]) ∪ {i}
 3. ρ(x[0]) ← ρ(x[0]) ∪ {1}
 4. for i ← 1 to m − 2 do
 5.       ρ(x[i]) ← ρ(x[i]) ∪ {i − 1, i + 1}
 6. ρ(x[m − 1]) ← ρ(x[m − 1]) ∪ {m − 2}
 7. for each c ∈ Σ do
 8.       i ← min ρ(c)
 9.       j ← max ρ(c)
10.       r(c) ← [i : j]
11. return r
```

Figure 4.3: The PREPROCESSING procedure of the BRAM algorithm adapted to the swap matching problem.

We recall that equal adjacent characters cannot be involved in a swap, so if $\hat{x}[i] = \{x[i-1], x[i], x[i+1]\}$ and $x[i] = x[i+1]$, its value reduces to $\hat{x}[i] = \{x[i-1], x[i]\}$.

Example 6. *Let $x =$ "banana" be a string of length 6. The superimposed pattern is the sequence of set of characters $\hat{x}$ given by:*

$$\hat{x} = \langle\{a, b\}, \{a, b, n\}, \{a, n\}, \{a, n\}, \{a, n\}, \{a, n\}\rangle$$

Then we extend the definition of the position function ρ to the case of a pattern whose elements are sets of characters. Formally:

$$\rho(c) = \{i : 0 \leq i < m \text{ and } c \in \hat{x}[i]\}. \tag{4.1}$$

Such definition allows us to easily adapt the preprocessing phase of the BRAM algorithm to the case of swap mathing.

The pseudocode of the PREPROCESSING procedure is shown in Figure 4.3. It is straightforward to observe that the preprocessing phase can be executed in $O(m)$ time.

The search phase is developed in the same way to the original BRAM algorithm. Suppose we are in any of the attempts of the search phase, assuming that

$W = y[j \,..\, j + m - 1]$. At the beginning of the attempt the configuration of the automaton is initialized to the set of initial states I_r. This is done, as usual, by setting $R = [0 : m] = \{0, 1, \ldots, m\}$. Thus at the beginning of each attempt all automaton states are active. Suppose R is the configuration of the range automaton after having read the suffix $y[j+q+1 \,..\, j+m-1]$, while proceeding in the backward scan of the window. The configuration of the automaton is then updated on the character $y[j + q]$, by the following operation:

$$R \leftarrow (R \ll 1) \cap r(y[j + q])$$

If $0 \in R$ after the whole scan of the current window of the text we verify the occurrence of any given pattern in § at position j of the text by running procedure VERIFY(x, m, T, j). Since the verification procedure takes $O(m)$ time the whole searching phase achieves an $O(mn)$ worst case time complexity.

Example 7. *Let $x =$* `banana` *be a pattern of length* 6 *and assume again $W =$* `anaban` *is the current window of the text. Plainly we have $r($*`a`*$) = [0 : 5]$, $r($*`b`*$) = [0 : 1]$ and $r($*`n`*$) = [1 : 5]$. The following table shows the configurations of the Range Automaton obtained during the backward scan of the string W.*

Iteration	*Operation*	*Range-Set Computation*	*Configuration*
iteration 0.	*initial state*	$R_0 = [0 : 5]$	`[0 1 2 3 4 5]`
iteration 1.	*read* `n`	$R_1 = [0 : 4] \cap [1 : 5] = [1 : 4]$	`0[1 2 3 4]5`
iteration 2.	*read* `a`	$R_2 = [0 : 3] \cap [0 : 5] = [0 : 3]$	`[0 1 2 3]4 5`
iteration 3.	*read* `b`	$R_3 = [0 : 2] \cap [0 : 1] = [0 : 1]$	`[0 1]2 3 4 5`
iteration 4.	*read* `a`	$R_4 = [0 : 0] \cap [0 : 5] = [0 : 0]$	`[0]1 2 3 4 5`
iteration 5.		$R_5 = \emptyset$	

The algorithm described above can also be easily extended to the case of condensed alphabets (see Section 2.0.3), by applying the strategy already adopted by Faro and Pavone [31] for the generalization of the Skip-Search algorithm, implemented with q-grams, to the case of swap matching. We do not go into the implementation details of this extension but we refer the reader to the original paper. We underline that the resulting algorithm still achieves an $O(m)$-space complexity and an $O(mn)$ worst case time complexity.

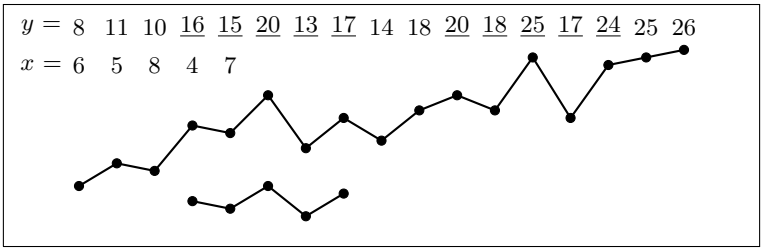

Figure 4.4: Example of a pattern x of length 5 over an integer alphabet with two order preserving occurrences in a text y of length 17, at positions 3 and 10.

4.4 Extension to Order Preserving String Matching

The *order-preserving pattern matching problem* [32, 33, 34, 35, 36, 37] (OPPM in short) is an approximate variant of the exact pattern matching problem which has gained more and more attention in recent years. In this variant the characters of x and y are drawn from an ordered alphabet Σ with a total order relation defined on it. The task of the problem is to find all substrings of the text with the same *relative order* as the pattern.

For instance the relative order of the sequence $x = \langle 6, 5, 8, 4, 7\rangle$ is the sequence $\langle 3, 1, 0, 4, 2\rangle$ since the element 6 has rank 3 in x, the element 5 as rank 1 in x, and so on. Thus x occurs in the string $y = \langle 8,\ 11,\ 10,\ 16,\ 15,\ 20,\ 13,\ 17,\ 14,\ 18,\ 20,\ 18,\ 25,\ 17,\ 20,\ 25,\ 26\rangle$ at position 3, since subsequence $\langle 16, 15, 20, 13, 17\rangle$ shares with x the same relative order. Another occurrence of x in y is at position 10 (see Fig. 4.4).

The OPPM problem finds applications, for example, to time series analysis like share prices on stock markets, weather data or to musical melody matching of two musical scores, where finding patterns affected by relative orders is more significant than considering their absolute values.

Assuming that a total order relation "$\leq$" is defined on the elements of the input alphabet Σ, we say that two (non-null) sequences x, y over Σ are order-isomorphic if the relative order of their elements is the same. More formally:

Definition 5 (Order-isomorphism)**.** *Two non-null sequences x, y of the same length,*

over a totally ordered alphabet $(\Sigma, \leq)$*, are said to be* order-isomorphic*, and we write* $x \approx y$*, if the following condition holds*

$$x[i] \leq x[j] \iff y[i] \leq y[j], \quad \textit{for } 0 \leq i, j < |x| \,.$$

From a computational point of view, it is convenient to characterize the order of a sequence by means of two functions: the *rank* and the *equality* functions.

Definition 6 (Rank function)**.** *Let* x *be a non-null sequence over a totally ordered alphabet* $(\Sigma, \leq)$*. The* rank function of x *is the bijection from* $\{0, 1, \ldots, |x| - 1\}$ *onto itself defined, for* $0 \leq i < |x|$*, by*

$$rk_x(i) = \left|\{k : x[k] < x[i] \textit{ or } (x[k] = x[i] \textit{ and } k < i)\}\right| .$$

Definition 7 (Equality function)**.** *Let* x *be a sequence of length* $m \geq 2$ *over a totally ordered alphabet* $(\Sigma, \leq)$*. The* equality function *of* x *is the binary map* $eq_x \colon \{0, 1, \ldots, m-2\} \to \{0, 1\}$ *where, for* $0 \leq i \leq m - 2$*,*

$$eq_x(i) = \begin{cases} 1 & \textit{if } x[rk_x^{-1}(i)] = x[rk_x^{-1}(i+1)] \\ 0 & \textit{otherwise.} \end{cases}$$

Lemma 2. [37] *Let* x *and* y *be two sequences of the same length* $m \geq 2$*, over a totally ordered alphabet. Then* $x \approx y$ *if and only if the following conditions hold:*

(i) $y[rk_x^{-1}(i)] \leq y[rk_x^{-1}(i+1)]$*, for* $0 \leq i < m - 1$

(ii) $y[rk_x^{-1}(i)] = y[rk_x^{-1}(i+1)]$ *if and only if* $eq_x(i) = 1$*, for* $0 \leq i < m - 1$. ■

Thus in order to establish whether two given sequences of the same length m are order-isomorphic, it is enough to compute their rank and equality functions. The cost of the test is dominated by the cost $\mathcal{O}(m \log m)$ of sorting the sequences.

Based on Lemma 2, the procedure VERIFY verifies correctly whether a sequence y is order-isomorphic to a sequence x of the same length as y. It receives as input the functions rk_x and eq_x and the sequence y, and returns true if $x \approx y$, false otherwise. A mismatch occurs when one of the three conditions of lines 2, 3, or 4 holds. Notice that the time complexity of the procedure VERIFY is linear in the size of its input sequence y.

```
VERIFY(y, j, m, inv-rk, eq)
1.   for i ← 0 to m − 2 do
2.     if (y[j + inv-rk(i)] > y[j + inv-rk(i + 1)]) then return false
3.     if (y[j + inv-rk(i)] < y[j + inv-rk(i + 1)] and eq(i) = 1) then return false
4.     if (y[j + inv-rk(i)] = y[j + inv-rk(i + 1)] and eq(i) = 0) then return false
5.   return true
```

Figure 4.5: The auxiliary procedure which verifies whether the text window of length m and starting at position j is order-isomorphic to a pattern having inverse rank function *inv-rk* and equality function *eq*.

4.4.1 BRAM for Order Preserving String Matching

The new algorithm is based on the efficient filtration technique presented in [36] by Faro *et al.*. Their method, called *Neighborhood Ranking* filtering approach, makes use of information extracted from groups of q integers of the input string [†], and relies on the following definition of q-neighborhood:

Definition 8 (q-neighborhood)**.** *Given a string x of length m, we define the q-neighborhood of the element $x[i]$, with $0 \leq i < m - q$, as the sequence of $q + 1$ elements from position i to $i + q$ in x, i.e. the sequence $\langle x[i], x[i + 1], \ldots, x[i + q]\rangle$.*

Given a string x of length m, we can compute the relative position of the element $x[i]$ compared with the element $x[j]$ by querying the inequality $x[i] \geq x[j]$. For brevity we will write in symbol $\beta_x(i, j)$ to indicate the boolean value resulting from the above inequality. It is easy to observe that if $\beta_x(i, j) = 1$ we have that $rk_x^{-1}(i) \geq rk_x^{-1}(j)$ ($x[j]$ precedes $x[i]$ in the ordering of the elements of x), otherwise $rk_x^{-1}(i) < rk_x^{-1}(j)$.

The neighborhood ranking (NR) approach associates each position i of the string x (where $0 \leq i < m - q$) with the sequence of the relative positions between $x[i]$ and $x[i + j]$, for $j = 1, \ldots, q$. In other words, it computes the binary sequence

[†]In this context, the value of q represents a trade-off between the computational time required for computing the q-grams for each window of the text and the computational time needed for checking false positive candidate occurrences.

$\langle \beta_x(i, i+1), \beta_x(i, i+2), \ldots, \beta_x(i, i+q) \rangle$ of length q indicating the relative positions of the element $x[i]$ compared with other values in its q-neighborhood. Of course, the relative position of $\beta(i, i)$ is not included in the sequence, since it doesn't give any additional information.

Since there are 2^q possible configurations of a binary sequence of length q the string x is converted in a sequence χ_x^q of length $m - q$, where each element $\chi_x^q[i]$, for $0 \leq i < m - q$, is a value such that $0 \leq \chi_x^q[i] < 2^q$. More formally we have the following definition:

Definition 9 (q-NR sequence)**.** *Given a string x of length m and an integer $q < m$, the q-NR sequence associated with x is a numeric sequence χ_x^q of length $m - q$ over the alphabet $\{0, \ldots, 2^q\}$ where*

$$\chi_x^q[i] = \sum_{j=1}^{q} \left(\beta_x(i, i+j) \times 2^{q-j} \right), \text{ for all } 0 \leq i < m - q$$

Example 8. *Let $x = \langle 5, 6, 3, 8, 10, 7, 1, 9, 10, 8 \rangle$ be a sequence of length 10. The 4-neighborhood of the element $x[2]$ is the subsequence $\langle 3, 8, 10, 7, 1 \rangle$. Observe that $x[2]$ is greater than $x[6]$ and less than all other values in its 4-neighborhood. Thus the ranking sequence associated with the element of position 2 is $\langle 0, 0, 0, 1 \rangle$ which translates in a NR value equal to 1. In a similar way we can observe that the NR sequence associated with the element of position 3 is $\langle 0, 1, 1, 0 \rangle$ which translates in a NR value equal to 6. The whole 4-NR sequence of length 6 associated to x is $\chi_x^4 = \langle 4, 8, 1, 6, 15, 8 \rangle$.*

The preprocessing phase of the BRAM algorithm consists in the computations of the inverse-rank function *inv-rk*$_x$ (Definition 6), the equality function eq_x (Definition 7), and the range function r (Section 4.1). The rank function rk_x, and its inverse *inv-rk*$_x$, can be computed (line 1) by using a stable sort algorithm in $\mathcal{O}(m \log m)$ time while the equality function eq_x can be computed (line 2) in $\mathcal{O}(\sigma)$ time. The range function r is computed (lines 3-10) by means of two simple for loops. The first for loop of line 3 initializes $r(c)$ to the empty set, for each $0 \leq c < 2^q$. The second

```
PREPROCESSING(x, m, y, n)
 1.  inv-rk_x ←Compute-Inverse-Rank-Function(x, m, q)
 2.  eq_x ←Compute-Equality-Function(x, m, q)
 3.  for each c ∈ {0..2^q − 1} do
 4.        r(c) ← ∅
 5.  for i ← 0 to m − q do
 6.        c ← χ^q_x[i]
 7.        if (r(c) = ∅) then r(c) ← [i : i]
 8.        else
 9.              [a : b] ← r(c)
10.              r(c) ← [a : i]
```

Figure 4.6: The preprocessing of the BRAM algorithm for the OPPM problem.

for loop of line 5 iterates over the pattern x, computes on the fly the element $\chi^q_x[i]$, for $0 \leq i \leq m-q$, and updates the range set $r(\chi^q_x[i])$ accordingly. The two for loops take $\mathcal{O}(\sigma)$-time and $\mathcal{O}(m)$-time, respectively. Thus the preprocessing phase of the BRAM algorithm, whose pseudocode is reported in Figure 4.6 achieves an overall $\mathcal{O}(m \log m + \sigma)$-time and $\mathcal{O}(\sigma)$-space complexity.

The searching phase of the algorithm exactly works as in the case of exact string matching, with the only difference that transitions are applied on $\chi^q_y[i]$. Thus, it still takes $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space.

4.5 Extension to Multiple String Matching

The *multiple string matching problem* is the natural generalization of the exact string matching problem to a set X of k different patterns[‡]. A trivial solution to such problem consists in applying an exact string matching algorithm for locating each

[‡]We assume, with a simplification, that the patterns in X have all the same length m. However, this simplification is not excessively limiting since we could easily transform the set X into a set that has exactly this feature, considering only the prefixes of length m' of all patterns, where m' is the length of the shortest pattern in X. Whenever the occurrence of a pattern is identified, it is always possible to check the presence of the cut suffix in the text. A more practical approach to handle patterns of varying length can be found in [38].

pattern $x \in X$. If we use the well–known Knuth-Morris-Pratt algorithm [3], whose time complexity is linear in the size of the text, the resulting algorithm achieves an $O(k(m+n))$ worst case time complexity. However, it is well known that the problem has a complexity in the worst case that is still linear on the size of the text. The renowned algorithm by Aho and Corasick [39] achieves this result by using a generalized automaton built on the set of k patterns. Its time complexity is $O(mk+n)$. An alternative construction of the Aho and Corasick automaton [40] achieves $\mathcal{O}(km+n\log|\Sigma|+occ)$, where occ is the total number of occurrences of the set of patterns in the text.

The optimal average complexity of the problem is $O(n\log_\sigma(km)/m)$ [41], a bound which is achieved by the Set-Backward-DAWG-Matching (SBDM) algorithm [42, 5]. The SBDM algorithm builds an exact indexing structure for the reverse strings in X, such as a factor automaton or a generalized suffix tree.

Hashing also provides a simple and efficient method for indexing suffixes of the strings of x. It has been used first by Wu and Manber [43] to design an efficient algorithm for multiple string matching with a sub-linear average complexity which uses an index table for blocks of q characters.

Also the Shift-Or [11] and BNDM [12] algorithms, which are based on bit-parallelism, can be easily extended to the multiple patterns case by deriving the corresponding automata from the maximal trie of the set of patterns [43, 44]. The resulting algorithms have a $O(\sigma\lceil m/w\rceil)$-space complexity and work in $O(n\lceil m/w\rceil)$ and $O(n\lceil m/w\rceil m)$ worst-case searching time complexity, respectively. Another efficient solution is the MBNDM algorithm [45], which computes a superimposed pattern from the patterns of the input set when using a condensed alphabet of q characters, and performs filtering with the standard BNDM.

The use of SIMD instructions in the multiple string matching problem has been explored only recently with the MPSSEF algorithm [46], which represents one of the most efficient solutions currently available for the problem. It runs in $O(nm)$ time complexity and uses $O(km+2^\alpha)$ additional space, where α indicates the size of the

packed word in bits.

4.5.1 BRAM for Multiple String Matching

Our solution for the multiple string matching problem makes again use of the superimposition technique to filter the candidates occurrences, along the same lines of what described in Section 4.3.1. Specifically, assume that $X = \{x_0, x_1, \ldots, x_{k-1}\}$ is the set of k patterns, each of length m. We define the superimposed pattern of the k strings in X as the pattern $\hat{x}$ obtained by merging the characters of each of the k patterns at corresponding positions. More formally, for $0 \leq i < m$,

$$\hat{x}[i] = \bigcup_{j=0}^{k-1} \{x_j[i]\}.$$

In other words, the i-th character of $\hat{x}$ is the set of the i-th characters of all $x \in X$. For simplicity, we will also write $\hat{x} = \bigcup_{j=0}^{k-1} x_j$.

Example 9. *Let* $X = \{\texttt{banana}, \texttt{ananas}, \texttt{brands}\}$ *be a set of* 3 *patterns of length* 6. *The superimposed pattern is the sequence of sets of characters* $\hat{x}$ *given by:*

$$\hat{x} = \langle \{a, b\}, \{a, n, r\}, \{a, n\}, \{a, n\}, \{a, d, n\}, \{a, s\} \rangle$$

We make use of the definition of the extended position function ρ to a set of characters as given in (4.1). Such definitions allow us to easily adapt the preprocessing phase of the BRAM algorithm to multiple patterns. The pseudocode of the PREPROCESSING and VERIFY procedures is shown in Figure 4.7. It is straightforward to observe that the preprocessing phase can be executed in $O(km)$ time.

The search phase remains once again unchanged with respect to that proposed in the original BRAM algorithm. Thus R is the configuration of the Range Automaton and $0 \in R$ after the whole scan of the current window of the text we verify the occurrence of any given pattern in X at position j of the text by running procedure VERIFY(x, m, y, j). Since the verification procedure takes $O(km)$ time the overall time complexity of the algorithm is $O(mnk)$.

```
PREPROCESSING(x, m, k)
 2.  for each c ∈ Σ do
 3.        ρ(c) ← ∅
 2.  for j ← 0 to k − 1 do
 2.        for i ← 0 to m − 1 do
 3.              ρ(x_j[i]) ← ρ(x_j[i]) ∪ {i}
 4.  for c ∈ Σ do
 5.        i ← min ρ(c)
 6.        j ← max ρ(c)
 7.        r(c) ← [i : j]
 8.  return r

VERIFY(x_i, m, k, y, j)
 1.  for r = 0 to k − 1 do
 2.    if x_r = y[j . . j + m − 1] then
 3.        return True
 4.  return False
```

Figure 4.7: The PREPROCESSING and VERIFY procedures of the BRAM algorithm adapted to the multiple string matching problem.

Example 10. *Let* $X = \{$`banana`, `ananas`, `brands`$\}$ *be a set of* 3 *pattern of length* 6 *and assume again* $W =$ `anaban` *is the current window of the text. Plainly we have* $r($`a`$) = [0 : 5]$, $r($`b`$) = [0 : 0]$, $r($`d`$) = [4 : 4]$, $r($`n`$) = [1 : 4]$, $r($`r`$) = [1 : 1]$ *and* $r($`s`$) = [5 : 5]$. *The following table shows the configurations of the Range Automaton obtained during the backward scan of the string* W.

Iteration	*Operation*	*Range-Set Computation*	*Configuration*
iteration 0.	*initial state*	$R_0 = [0 : 5]$	`[0 1 2 3 4 5]`
iteration 1.	*read* `n`	$R_1 = [0 : 4] \cap [1 : 4] = [1 : 4]$	`0[1 2 3 4]5`
iteration 2.	*read* `a`	$R_2 = [0 : 3] \cap [0 : 5] = [0 : 3]$	`[0 1 2 3]4 5`
iteration 3.	*read* `b`	$R_3 = [0 : 2] \cap [0 : 0] = [0 : 0]$	`[0]1 2 3 4 5`
iteration 4.		$R_4 = \emptyset$	

4.6 Experimental Comparison

We report in this section the results of an extensive experimentation of the BRAM algorithm for each of the considered searching problems. All algorithms have been

implemented in the C programming language[§] and have been tested using the SMART tool [26]. All experiments have been executed locally on a computer running Linux Ubuntu 20.04.1 with an Intel Core i5 3.40 GHz processor and 8GB RAM. Our tests have been run on a genome sequence, a protein sequence, and an English text (each of size 10MB). Such sequences are provided by the SMART research tool and are available online for download.[¶].

The following subsections analyze in detail, case by case, the results obtained by comparing the solutions based on the new range automaton in the four scenarios presented in this chapter: exact string matching, swap matching, order preserving string matching and multiple string matching, respectively.

4.6.1 Exact string matching

For the online exact string matching problem, we mostly focused on those algorithms which make use of the suffix automaton. Specifically, the following 10 algorithms (implemented in 33 variants, depending on the values of their parameters) have been compared: the BNDM_q algorithm [12] implemented with q-grams, for $2 \leq q \leq 6$; the LBNDM algorithm [7]; the BSX_q algorithm [8] implemented using q-grams, with $2 \leq q \leq 4$; the FBNDM algorithm [9] of the BNDM algorithm [12]; the BSDM_q algorithm [10] using q-grams characters, with $3 \leq q \leq 7$; the BRAM_q algorithm, implemented using q-grams characters, with $3 \leq q \leq 7$.

For completeness, we also evaluated some among the most efficient algorithms in practice and specifically: the Maximal Average Shift algorithm and its variants [47] (MAS, MAS_4 and TMAS), specifically designed for genome sequences and short patterns;[‖] the Weak Factors Recognition (WFR) algorithm [48], implemented using q-grams, with $3 \leq q \leq 7$ and its variant (TWFR); the Exact Packed String Matching

[§]The source code of the new BRAM algorithm is available at the following link: `https://github.com/ostafen/range-automaton`

[¶]Additional details on the sequences can be found in Faro et al. [26]

[‖]Search speed of MAS and its variants, MAS_4 and TMAS, has been omitted starting from $m = 256$, since the preprocessing time of such solutions become prohibitive as the length of the pattern increases.

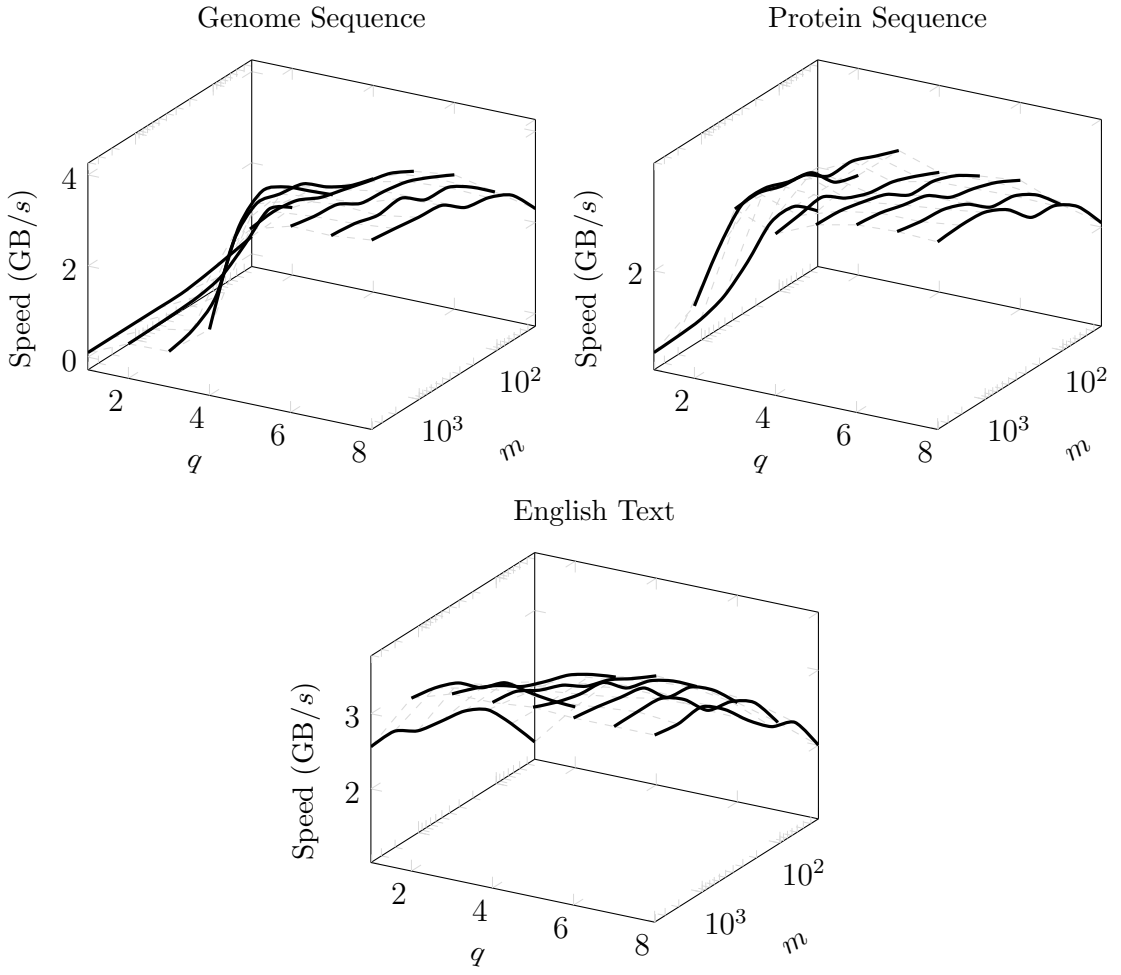

Figure 4.8: Running times of the BRAM algorithm extended with condensed alphabets using groups of q characters. We report searching speed of the algorithms for different values of q. Experimental tests have been conducted on a genome sequence and a protein sequence. Speed is reported in GB/s.

(EPSM) algorithm [49] based on SIMD instructions.**

In the experimental evaluation, patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 5 \leq i \leq 16\}$. In all cases, the mean over the search speed (expressed in Gigabytes per seconds) of 1000 runs has been reported. Table 4.1 summarises our evaluations. Each table is divided

**We notice that the EPSM algorithm is designed for simply counting the number of matching occurrences without reporting the corresponding positions.

Genome Sequence

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
$BNDM_q$	**3.05**	3.07	3.01	2.96	3.05	3.01	3.03	2.96	2.94	3.00	2.94	2.94
LBNDM	1.56	1.78	2.00	0.94	0.19	0.20	0.23	0.25	0.25	0.25	0.25	0.26
BXS_q	2.96	2.86	2.98	3.07	2.96	3.03	2.96	3.09	2.98	2.98	2.76	2.89
FBNDM	1.76	2.25	2.42	2.26	2.26	2.47	2.34	2.48	2.41	2.37	2.29	2.38
$BSDM_q$	2.48	2.53	2.58	2.65	2.65	2.79	2.74	2.82	2.74	2.76	2.73	2.73
$BRAM_q$	2.81	**3.11**	**3.32**	**3.41**	**3.76**	**3.97**	**3.91**	**4.21**	**6.98**	**11.3**	**13.5**	**11.1**
MAS	0.96	1.18	1.40	1.64	-	-	-	-	-	-	-	-
MAS_4	2.19	2.82	3.26	3.37	-	-	-	-	-	-	-	-
TMAS	1.18	1.54	1.74	1.74	-	-	-	-	-	-	-	-
EPSM	**3.37**	**3.41**	**3.62**	**3.59**	3.76	3.94	3.67	3.97	4.00	4.98	4.65	4.88
WFR_q	3.05	3.26	3.39	**3.59**	**3.81**	3.84	3.94	4.10	6.78	9.57	6.60	2.38
$TWFR_q$	2.49	2.63	3.21	3.30	3.76	3.81	4.00	4.07	6.78	9.39	6.69	2.48

Protein Sequence

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
$BNDM_q$	2.42	2.42	2.41	2.43	2.44	2.38	2.41	2.39	2.41	2.39	2.56	2.56
LBNDM	1.76	2.01	2.14	2.31	2.41	2.20	1.11	0.49	0.39	0.39	0.40	0.39
BXS_q	**2.67**	**2.70**	**2.67**	**2.70**	2.67	2.70	2.65	2.68	2.63	2.64	2.63	2.61
FBNDM	1.97	2.13	2.29	2.29	2.29	2.28	2.26	2.26	2.25	2.27	2.45	2.25
$BSDM_q$	2.33	2.44	2.48	2.52	2.54	2.54	2.52	2.56	2.54	2.50	2.79	2.57
$BRAM_q$	2.21	2.38	2.52	2.61	**2.82**	**2.77**	**2.96**	**2.98**	**5.37**	**8.88**	**12.2**	**10.8**
EPSM	2.48	2.56	2.65	**2.81**	**2.86**	2.87	2.87	2.91	3.01	3.49	4.07	3.79
WFR_q	2.34	2.48	2.58	2.71	2.81	**2.96**	**2.98**	**3.00**	**5.43**	9.04	10.61	5.49
$TWFR_q$	2.37	2.49	2.56	2.70	**2.86**	2.94	**2.98**	**3.00**	5.37	**8.88**	10.39	5.55

English Text

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
$BNDM_q$	2.50	2.57	2.58	2.56	2.61	2.57	2.41	2.35	2.36	2.34	2.35	2.3
LBNDM	1.68	2.06	2.35	2.53	2.58	2.61	2.23	1.63	1.05	0.74	0.61	0.52
BXS_q	2.57	2.65	2.6	2.58	2.58	2.57	2.6	2.58	2.58	2.53	2.50	2.44
FBNDM	1.93	2.16	2.42	2.44	2.43	2.43	2.20	2.18	2.21	2.15	2.20	2.17
$BSDM_q$	**2.60**	2.67	2.74	2.73	2.77	2.77	2.54	2.60	2.58	2.60	2.65	2.65
$BRAM_q$	2.45	**2.70**	**2.81**	**2.94**	**3.05**	**3.15**	**3.01**	**3.05**	**5.49**	**9.21**	**12.5**	**11.3**
EPSM	**2.65**	**2.71**	**2.84**	**3.00**	**3.07**	3.07	2.94	2.98	3.26	3.62	3.81	4.14
WFR_q	2.50	2.65	2.74	2.82	3.00	2.89	3.00	3.03	5.25	8.00	7.40	3.81
$TWFR_q$	2.64	**2.71**	2.76	2.91	3.03	3.11	2.94	3.00	5.25	8.00	7.40	3.81

Table 4.1: Experimental results obtained for exact string matching on a genome sequence (at the top), a protein sequence (in the center) and an English text (in the bottom). Searching speed is reported in GB/s. Best results have been bold faced.

into two blocks. The first block presents results of the most efficient automata based algorithms while the second block concerns the search speed obtained by other algorithms. Best results have been boldfaced both among automata-based

algorithms and among the entire set of algorithms.

Among the automata-based algorithms the new algorithm turns out to be the best in many cases, obtaining increasingly higher performances as the length of the pattern increases, showing considerable speed ups, especially in the case of long patterns. In particular, other algorithms are superior only for $m = 32$, and, in the case of protein sequences, up to $m = 256$. However, as m grows beyond 1024, the BRAM algorithm becomes by far faster than the previous solutions, reaching a search speed up to 4.6 times higher than the second best solution.

Extending the comparison also to non-automata-based solutions, it is interesting to note how the BRAM algorithm scales better as the size of the pattern increases, outperforming all the remaining algorithms starting from $m = 1024$, both in the case of genome sequences and for texts in natural language. In the case of protein sequences, both WFR_q and TWFR_q turn out to be competitive up to $m = 8192$, but fail to scale-up with respect to the new approach for larger values of m. Moreover, we also notice how the BRAM algorithm is still very competitive also for patterns of medium size, since the search speed never deviates too much from the best results.

4.6.2 Experimental Results on Swap Matching

For evaluating the swap matching variant of our BRAM algorithm, we compared a set of 6 algorithms. According to [30] and [31] the first five algorithms in the list are the most efficient solutions for this problem:

- BPCS: the Bit Parallel Cross Sampling [28];
- BPBCS: the Bit Parallel Backward Cross Sampling [29];
- BPSRA: the Bit Parallel Swap Reactive Automata [30];
- BPSRO: the Bit Parallel Swap Reactive Oracle [30];
- SKIP_q: the Skip Search algorithm for swap matching [31], using q-grams and implemented with $1 \leq q \leq 5$;

- BRAM$_q$: the Backward Range Automaton Matcher presented in Section 4.3.1, implemented using $1 \leq q \leq 5$.

We considered patterns of length m ranging in the set $\{2^i \mid 0 \leq i \leq 12\}$. Figure 4.9 summarises the running times of our evaluations, including the speed-up (in percentage) obtained by the BRAM algorithm against the best running time among the previous solutions. Any positive values denote a performance improvement. Running times representing best results have been bold-faced.

From experimental results, it turns out that the Skip Search algorithm and the BRAM both emerge as winners, for small and large strings respectively, with BRAM being specifically advantaged for genome sequences (where it achieves the best results in 7/11 cases). Noticeably, the BRAM algorithm is always close to the Skip Search algorithms also in those cases in which the latter is superior in terms of absolute times, while it tends to be by far faster in all other cases, reaching important speed-ups as m increases, up to 70% and 103% depending on the text buffer. This makes our BRAM algorithm one of the most efficient solutions for the swap matching problem.

4.6.3 Experimental Results on Order Preserving String Matching

In this section we present experimental results in order to evaluate the performance of the algorithm presented in this paper. In [36] Faro and Kulekci applied the q-neighborhood ranking approach to the SBNDM2 algorithm. However in our experimental evaluation, we found it appropriate to also include the SKIP algorithm [37], because it scales better as m grows (and thus, it is more competitive with our BRAM algorithm). All of the algorithms have been implemented using the q-neighborhood filtering approach, for increasing values of $q = 4, 8, 12$.

We evaluated our filter based solutions in terms of efficiency, reporting the average running times, in gigabytes per second (GB/s), and, when possible, also the speed-up with respect to the second best solution. We tested our solutions

Genome Sequence

m	BPCS	BPBCS	BPSRA	BPSRO	SKIP$_q$	BRAM$_q$	SPEED-UP
4	0.54	0.31	0.31	**0.7**	$0.51^{(4)}$	$0.44^{(4)}$	$\sim$
8	0.62	0.54	0.54	0.81	$\mathbf{0.86}^{(4)}$	$0.84^{(4)}$	$\sim$
16	0.62	0.85	0.85	0.83	$1.01^{(4)}$	$\mathbf{1.02}^{(4)}$	1%
32	0.62	1.34	1.34	0.84	$1.14^{(4)}$	$\mathbf{1.27}^{(4)}$	11%
64	0.61	1.31	1.31	0.82	$1.31^{(4)}$	$\mathbf{1.7}^{(4)}$	30%
128	0.62	1.32	1.32	0.83	$1.46^{(4)}$	$\mathbf{2.03}^{(4)}$	39%
256	0.62	1.32	1.32	0.83	$1.52^{(4)}$	$\mathbf{2.17}^{(5)}$	43%
512	0.62	1.3	1.3	0.83	$1.44^{(4)}$	$\mathbf{1.95}^{(5)}$	35%
1024	0.63	1.31	1.31	0.84	$1.18^{(4)}$	$\mathbf{1.35}^{(5)}$	14%
2048	0.62	1.31	1.31	0.83	$\mathbf{1.05}^{(4)}$	$0.77^{(5)}$	$\sim$
4096	0.62	1.31	1.31	0.83	$\mathbf{0.91}^{(4)}$	$0.51^{(5)}$	$\sim$

Protein Sequence

m	BPCS	BPBCS	BPSRA	BPSRO	SKIP$_q$	BRAM$_q$	SPEED-UP
4	0.55	0.5	0.5	0.74	$\mathbf{0.89}^{(2)}$	$0.82^{(2)}$	$\sim$
8	0.55	0.72	0.72	0.73	$\mathbf{1.43}^{(4)}$	$1.26^{(4)}$	$\sim$
16	0.55	1.21	1.21	0.73	$\mathbf{1.95}^{(4)}$	$1.84^{(4)}$	$\sim$
32	0.55	1.74	1.74	0.73	$\mathbf{2.26}^{(4)}$	$2.11^{(4)}$	$\sim$
64	0.55	1.73	1.73	0.73	$\mathbf{2.31}^{(4)}$	$2.21^{(4)}$	$\sim$
128	0.55	1.72	1.72	0.74	$2.36^{(4)}$	$\mathbf{2.48}^{(4)}$	5%
256	0.55	1.75	1.75	0.73	$2.36^{(4)}$	$\mathbf{2.68}^{(4)}$	14%
512	0.55	1.71	1.71	0.73	$2.19^{(4)}$	$\mathbf{2.74}^{(4)}$	25%
1024	0.55	1.76	1.76	0.74	$2.07^{(4)}$	$\mathbf{2.91}^{(4)}$	40%
2048	0.55	1.72	1.72	0.73	$1.8^{(4)}$	$\mathbf{2.68}^{(4)}$	48%
4096	0.54	1.74	1.74	0.74	$1.43^{(4)}$	$\mathbf{2.43}^{(4)}$	70%

Natural Language Text

m	BPCS	BPBCS	BPSRA	BPSRO	SKIP$_q$	BRAM$_q$	SPEED-UP
4	0.55	0.44	0.44	0.72	$\mathbf{0.95}^{(2)}$	$0.89^{(2)}$	$\sim$
8	0.55	0.7	0.7	0.72	$\mathbf{1.43}^{(3)}$	$1.26^{(3)}$	$\sim$
16	0.55	1.1	1.1	0.74	$\mathbf{1.96}^{(4)}$	$1.77^{(4)}$	$\sim$
32	0.55	1.57	1.57	0.72	$\mathbf{2.31}^{(4)}$	$2.13^{(4)}$	$\sim$
64	0.55	1.55	1.55	0.73	$\mathbf{2.41}^{(4)}$	$2.26^{(4)}$	$\sim$
128	0.55	1.57	1.57	0.73	$2.45^{(4)}$	$\mathbf{2.49}^{(4)}$	2%
256	0.55	1.57	1.57	0.71	$2.35^{(4)}$	$\mathbf{2.61}^{(4)}$	11%
512	0.55	1.59	1.59	0.72	$2.29^{(4)}$	$\mathbf{2.77}^{(4)}$	21%
1024	0.55	1.57	1.57	0.73	$2.13^{(4)}$	$\mathbf{3.01}^{(4)}$	41%
2048	0.56	1.56	1.56	0.73	$1.86^{(4)}$	$\mathbf{3.07}^{(4)}$	65%
4096	0.55	1.6	1.6	0.73	$1.49^{(3)}$	$\mathbf{3.03}^{(4)}$	103%

Figure 4.9: Experimental results obtained by running 6 swap matching algorithms on three text buffers. Experimental results have been conducted on three text buffers: (on the top) a genome sequence, (in the middle) a protein sequence and (on the bottom) a natural language text. Results are expressed in GB/s.

on sequences of short integer values only, where each element is an integer in the range $[0 \dots 256]$ (according to [36], considering long integers and floating point values doesn't affect too much the results). All texts have 1 million of elements. In particular we tested our algorithm on a RAND-δ sequence of random integer values varying around a fixed mean equal to 100 with a variability of δ.

For each text in the set we randomly select 100 patterns extracted from the text and compute the average running time over the 100 runs. All the algorithms have been implemented using the C programming language and have been compiled a computer running Linux Ubuntu 20.04.1 with an Intel Core i5 3.40 GHz processor and 8GB RAM. During the compilation we used the `-O3` optimization option. In the following table running times are expressed in GB/s. Best results have been underlined to be easily located.

Experimental results on RAND-δ numeric sequences have been conducted with values of $\delta = 5$, 20, and 40. Table 4.2 summarises our evaluations.

From experimental results, it turns out that the SBNMD2 algorithm is the most appropriate for sequences of length $m \leq 16$. However, it fails to scale for longer sequences because of the intrinsic limitation due the bit-parallelism approach. As the input size m increases, the SKIP and the BRAM algorithms both emerge as winners, for medium and large strings respectively. Indeed, as the value of m exceeds 64, the BRAM algorithm tends to be faster and faster, reaching important speed-ups (up to 62% faster with respect to the second best time) depending on the text buffer. Noticeably, our algorithm is the fastest for most of the time with respect to the alternative solutions. This makes our BRAM algorithm one of the most effective solutions for the order preserving matching problem.

4.6.4 Experimental Results on Multiple String Matching

In the case of multiple patterns, we compared the performances of the BRAM algorithm against the following best algorithms known in literature for the multiple string matching problem:

$\delta = 5$

m	8	16	32	64	128	256	512	1024	2048	4096
SBNDM2$_4$	3.91	3.41	3.49	3.18	3.21	3.19	3.20	3.17	3.17	3.20
SBNDM2$_8$	3.91	3.64	3.34	3.25	3.42	3.23	3.22	3.21	3.23	3.23
SBNDM2$_{12}$	**3.94**	**4.28**	4.36	4.16	4.15	4.15	4.13	4.15	4.19	4.16
SKIP$_4$	-	2.63	**4.98**	**6.18**	6.88	7.51	7.75	8.72	9.77	10.17
SKIP$_8$	0.99	2.89	3.84	4.52	5.09	5.49	5.68	5.81	6.34	6.88
SKIP$_{12}$	-	2.60	4.83	**6.18**	6.98	7.63	7.88	8.57	9.39	10.39
BRAM$_4$	1.03	1.76	2.45	2.21	1.65	1.48	1.30	1.13	0.93	0.67
BRAM$_8$	0.53	2.16	3.79	5.61	7.51	9.39	10.85	11.10	7.75	2.36
BRAM$_{12}$	-	1.46	3.64	6.03	**8.42**	**10.17**	**11.36**	**13.20**	**14.80**	**16.84**
Speed-Up -	-	-	-	-	**1.20**	**1.33**	**1.44**	**1.54**	**1.51**	**1.62**

$\delta = 20$

m	8	16	32	64	128	256	512	1024	2048	4096
SBNDM2$_4$	3.87	3.38	3.52	3.21	3.24	3.22	3.23	3.20	3.14	3.23
SBNDM2$_8$	3.87	3.60	3.37	3.28	3.45	3.26	3.19	3.24	3.20	3.26
SBNDM2$_{12}$	**3.90**	**4.24**	4.32	4.12	4.11	4.19	4.09	4.19	4.15	4.20
SKIP$_4$	-	2.76	**5.43**	6.98	8.14	9.04	9.21	10.61	11.63	11.63
SKIP$_8$	1.02	3.11	4.10	4.88	5.74	6.42	6.42	6.88	7.75	8.00
SKIP$_{12}$	-	2.77	5.37	**7.08**	8.14	8.88	9.39	10.39	11.63	11.63
BRAM$_4$	1.04	1.79	2.48	2.14	1.63	1.44	1.31	1.16	0.95	0.68
BRAM$_8$	0.54	2.21	3.79	5.49	7.51	9.39	10.85	11.36	7.18	2.28
BRAM$_{12}$	-	1.52	3.67	6.26	**8.28**	**9.96**	**11.63**	**13.20**	**15.26**	**16.84**
Speed-Up -	-	-	-	-	**1.2**	**1.12**	**1.23**	**1.27**	**1.31**	**1.45**

$\delta = 40$

m	8	16	32	64	128	256	512	1024	2048	4096
SBNDM2$_4$	3.85	3.46	3.44	3.13	3.16	3.24	3.25	3.22	3.12	3.15
SBNDM2$_8$	**3.97**	3.69	3.29	3.20	3.37	3.28	3.27	3.26	3.28	3.28
SBNDM2$_{12}$	3.88	**4.22**	4.43	4.10	4.21	4.09	4.19	4.09	4.25	4.10
SKIP$_4$	-	2.81	5.25	6.98	8.28	8.88	9.21	10.39	11.36	12.52
SKIP$_{12}$	-	2.77	**5.55**	**7.51**	8.14	9.04	9.39	10.39	11.63	12.52
SKIP$_8$	1.03	3.15	4.17	4.83	5.74	6.34	6.78	7.08	7.29	8.00
SKIP$_{12}$	-	2.77	**5.55**	**7.51**	8.14	9.04	9.39	10.39	11.63	12.52
BRAM$_4$	1.10	1.82	2.52	2.20	1.70	1.52	1.33	1.18	0.98	0.72
BRAM$_8$	0.55	2.18	3.73	5.68	7.75	9.77	10.85	11.36	7.29	2.10
BRAM$_{12}$	-	1.47	3.79	6.26	**8.57**	**9.96**	**11.36**	**13.20**	**14.80**	**16.84**
Speed-Up	-	-	-	-	**1.5**	**1.10**	**1.21**	**1.27**	**1.27**	**1.35**

Table 4.2: Experimental results on a Rand$-\delta$ short integer sequence. Search speed is reported in GB/s. Best results have been bold faced.

- MBNDM: the Multiple Backward DAWG Matching algorithm [45, 50];
- WM: the Wu-Manber algorithm, implemented with q-grams for $1 \leq q \leq 8$ [20];
- BRAM$_q$: the presented BRAM algorithm for multiple string matching, implemented with q-grams for $1 \leq q \leq 8$.

For completeness, we also included the MPSSEF$_\alpha$ algorithm [46], where α referes

Genome Sequence

$k = 10^2$

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}
MBNDM$_q$	1.14	1.14	1.14	1.13	1.12	1.11
WM$_q$	1.10	1.36	1.46	1.55	1.64	**1.71**
BRAM$_q$	**1.54**	**1.52**	**1.95**	**2.07**	**2.00**	1.69
MPSSEF$_\alpha$	1.64	2.04	2.15	2.27	2.29	2.12

$k = 10^3$

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}
MBNDM$_q$	0.56	0.56	0.56	0.56	1.12	0.55
WM$_q$	0.79	1.00	1.11	1.17	1.17	1.16
BRAM$_q$	**0.83**	**1.14**	**1.33**	**1.55**	**1.47**	**1.44**
MPSSEF$_\alpha$	1.51	1.76	1.77	1.77	1.75	1.76

$k = 10^4$

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}
MBNDM$_q$	0.01	0.01	0.01	0.01	0.01	0.01
WM$_q$	0.01	0.01	0.01	0.01	0.01	0.01
BRAM$_q$	**0.06**	**0.08**	**0.08**	**0.09**	**0.08**	**0.09**
MPSSEF$_\alpha$	0.17	0.17	0.14	0.13	0.11	0.10

Table 4.3: Running times, reported in GB/s, for $k \in \{10^2, 10^3, 10^4\}$ on a genome sequence. Best results over the first set of algorithms have been boldfaced.

to the number of packed characters and has been set to 32, 64 and 128. However, since the MPSSEF algorithm is based on SIMD instructions, the comparison only takes into account the first set of algorithms.

For each text buffer, experiments have been repeated by generating sets of 100, 1000 and 10000 patterns of fixed length m. In all cases the patterns were randomly extracted from the text and the value of m was made ranging over the values 32, 64, 128, 256, 512, and 1024. Times are reported in Table 4.3, 4.4 and 4.5. The BRAM algorithm performs very well both as the text buffer changes and as the number of patterns increase, with slight fluctuations for small patterns which allow WM to predominate in minor cases. As the pattern size increases, performance improves less noticeably than the base algorithm, since the filter is affected by the number of patterns, but in any case in a way superior to the remaining algorithms.

Protein Sequence

	Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}
$k = 10^2$	MBNDM$_q$	**1.51**	1.44	1.52	1.53	1.50	1.48
	WM$_q$	1.28	1.50	1.56	1.65	1.70	1.76
	BRAM$_q$	1.23	**1.52**	**1.74**	**1.77**	**2.00**	**2.13**
	MPSSEF$_\alpha$	1.63	2.04	2.19	2.28	2.36	2.11
$k = 10^3$	MBNDM$_q$	0.64	0.63	0.62	0.60	0.58	0.52
	WM$_q$	**0.80**	**0.95**	**0.95**	0.85	0.67	0.49
	BRAM$_q$	0.79	0.92	0.93	**0.91**	**0.87**	**0.67**
	MPSSEF$_\alpha$	0.88	0.92	0.91	0.88	0.86	0.80
$k = 10^4$	MBNDM$_q$	**0.24**	**0.24**	0.22	**0.19**	0.16	0.11
	WM$_q$	0.23	0.21	0.18	0.13	0.09	0.05
	BRAM$_q$	0.21	0.19	**0.24**	**0.19**	**0.18**	**0.14**
	MPSSEF$_\alpha$	0.17	0.16	0.15	0.13	0.11	0.09

Table 4.4: Running times, reported in GB/s, for $k \in \{10^2, 10^3, 10^4\}$ on a protein sequence. Best results over the first set of algorithms have been boldfaced.

English Text

	Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}
$k = 10^2$	MBNDM$_q$	1.15	1.12	1.07	1.12	1.12	1.12
	WM$_q$	1.19	1.36	1.53	**1.58**	1.73	1.76
	BRAM$_q$	**1.21**	**1.43**	**1.68**	1.55	**1.92**	**2.01**
	MPSSEF$_\alpha$	1.68	2.04	2.14	2.26	2.27	2.05
$k = 10^3$	MBNDM$_q$	0.29	0.29	0.28	0.28	0.27	0.27
	WM$_q$	**0.37**	**0.46**	0.51	0.52	0.46	0.36
	BRAM$_q$	0.32	0.43	**0.55**	**0.54**	**0.58**	**0.49**
	MPSSEF$_\alpha$	0.58	0.59	0.60	0.60	0.57	0.53
$k = 10^4$	MBNDM$_q$	0.04	0.04	0.04	0.04	0.04	0.03
	WM$_q$	**0.04**	0.04	0.05	0.04	0.04	0.03
	BRAM$_q$	**0.04**	**0.05**	**0.07**	**0.05**	**0.07**	**0.07**
	MPSSEF$_\alpha$	0.09	0.10	0.09	0.08	0.07	0.06

Table 4.5: Running times, reported in GB/s, for $k \in \{10^2, 10^3, 10^4\}$ on a protein sequence. Best results over the first set of algorithms have been boldfaced.

4.7 Chapter summary

In this chapter, we introduced the Range Automaton, a weak version of the non-deterministic suffix automaton of a string whose configuration can be encoded as a

simple pair of integers. Such encoding turns out to be effective in order to overcome the intrinsic space limitation of bit-parallel simulations of the suffix automaton. We then introduced a new efficient string matching algorithm, named Backward Range Automaton Matcher (BRAM), based on the Range Automaton of the pattern and derived some extensions of it also to non-conventional string matching problems, considering, in particular, the swap matching and the multiple pattern matching problem. We conducted an extensive experimental evaluation from which it turns out that our newly presented algorithm is very competitive when compared with the most efficient algorithms known in literature, both in its exact and extended form.

The good performances obtained by the BRAM algorithm suggest that its encoding is simple and flexible and allows us to imagine that it can be easily adapted also to other relevant text-processing problems we have not taken into consideration in this work.

5

UFM: a Two-Step Simulation of the Suffix Automaton

In this chapter, we present a new general approach to the exact string matching problem based on a non-standard two-step simulation of the suffix automaton of the pattern. We then introduce UFM, an exact string matching algorithm which efficiently implements the new approach in practice and present several variations and optimizations of it. Experimental results suggest that the new solutions are competitive with the most effective algorithms available for the exact string matching problem in practical cases, scaling much better when the length of the pattern increases.

5.1 The Unique Factor Matcher

As we have already noticed the efficiency of a suffix automaton based algorithm relies on the right trade-off between the encoding used to represent the underlying automaton and the size of the automaton itself. Regarding the first point it turns out that automata admitting simpler encoding are more efficient in practice. This is the

case, for instance, of bit-parallel based solutions which limit the size of the automaton to the machine word size in turn of an efficient representation, like LBNDM and BXS. However, on the other hand, longer shifts are achieved when the size of the underlying automaton is close to the length of the pattern. This is the case of the FBNDM algorithm which trades a more complex representation in exchange for a higher size of the automaton.

In this second part of the work we present a new algorithm, called Unique Factor Matcher (UFM), which allows a compact representation of a suffix automaton while managing to represent significantly longer patterns than what is currently done by other solutions in the literature, greatly increasing the efficiency of the search in the case of very long patterns.

Before going into the detail of the description of the UFM algorithm we present in the following section a generic algorithm, called Backward-Two-Step-Matcher (BTSM), for the online exact string matching problem based on a simplified and efficient simulation of the suffix automaton of the reverse of the pattern which, however, doesn't require its whole construction. As we will see, the UFM algorithm represents a specific implementation of the BTSM algorithm.

5.1.1 A Generic Backward-Two-Step-Matcher Algorithm

Let x be a pattern of length m over an alphabet Σ of size σ. Given the suffix automaton $S(x) = \langle Q, \Sigma, \delta, I, F \rangle$ of x, we define the *minimum transitions function* $\gamma : \Sigma^+ \longrightarrow \{1, 2, ..., m\}$ which associates any string $w \in \Sigma^*$ with the length of its shortest prefix which must be read in order to reach a configuration containing at most one state. More formally, for each string $w \in \Sigma^+$, we have

$$\gamma(w) = \min\{1 \leq \ell \leq m \ : \ |\delta^*(w_\ell)| \leq 1\}.$$

Note that such a prefix of w always exists, since $l = m$, in the worst case. Moreover, by the definition of δ, it trivially follows that if $\ell = \gamma(w)$ then $|\delta^*(w_\mu)| \leq 1$ for any

$\ell \leq \mu \leq m$.

In addition, we define the *position function* $\text{POS} : \Sigma^* \longrightarrow \{-1, 0, 1, ..., m-1\}$ as the function which maps any $w \in \Sigma^*$ to its unique starting position inside the pattern x, if such position exists, or to -1 otherwise. Formally, we have

$$\text{POS}(w) = \begin{cases} m - i & \text{if} \quad \delta^*(w) = \{q_i\},\ 0 < i \leq m, \\ -1 & \text{otherwise.} \end{cases}$$

Assume, for instance to match the pattern $x = \mathsf{banana}$ against the text window $w = \mathsf{anaban}$. Then, we have $\gamma(w) = 3$ and $\text{POS}(\mathsf{ban}) = 0$.

We are now ready to present the generic BTSM algorithm. The main underlying idea is that the recognition process of a string w through the automaton $S(x^r)$ can be simplified by dividing it in two separate steps: a first non-deterministic step possibly followed by a deterministic step.

Specifically, as before, let x be a pattern of length m and let y be a text of length n, both strings over a common alphabet Σ of size σ and let $S(x^r) = \langle Q, \Sigma, \delta, I, F \rangle$ the suffix automaton for the the reverse of the pattern.

As in the case of the standard BDM algorithm, the searching phase of the BTSM algorithm works by sliding a window w of length m along the text, starting from the left end of the text and proceeding from left to right. At each iteration of the algorithm a new window position is attempted. For each attempt the recognition process of a string w through the automaton $S(x^r)$ is divided in the following two steps:

- *non-deterministic step:* during the first step an integer value μ is computed, depending on w, such that $\gamma(w^r) \leq \mu \leq m$ and $\mu+1$ transitions are followed all at once by computing the position $p = \text{POS}(w^r_\mu)$ corresponding to the unique active state q (if any) belonging to $\delta^*(w^r_\mu)$. If no active state q exists, i.e. if $p = -1$, the window is advanced to the right by one position, if $\mu = m$, by $m - \mu$ positions otherwise.

- *deterministic step:* If $p \geq 0$, then the computation proceeds with the subsequent transitions, which are simulated by comparing each character of the pattern, starting from position p, with its counterpart in the text, until a mismatch occurs or until p transitions have been performed. If a mismatch occurs then the window is simply advanced by $m - \mu$ positions to the right. Otherwise if p characters are read then a prefix of size $k + \mu$ of the pattern has been recognized. If $k + \mu = m$ then the pattern itself has been recognized and a match is reported, otherwise the window is shifted in order to align the first character of x with the starting position of the recognized prefix.

Denoting by $f(m)$ the computational effort related to the computation of μ and p, the worst case time complexity of the BTSM algorithm is $\mathcal{O}((m + f(m)) \cdot n)$.

The approach described above represents a generic way to avoid managing multiple states while simulating a suffix automaton for a given string at the cost of reducing the length of the shifts. Indeed, the only way to compute the exact shift value s consists in recognizing each suffix of x^r (i.e. each prefix of x), through the use of a full suffix automaton. Performing $\mu + 1$ transitions at once implies that only suffixes of length $\mu' \geq \mu$ can be recognized. However, provided that we can determine a value of μ which is close to $\gamma(w^r)$ and that $\delta^*(w^r_\mu)$ can be computed efficiently for any given text window w, overestimating the shifts values impacts less the efficiency than simulating the full automaton.

5.1.2 A Practical Implementation: The UFM Algorithm

In this section we show how to turn the generic BTSM algorithm into a concrete efficient string matching algorithm. Our approach for estimating a good approximation for $\gamma(w)$ relies on the definition of *unique characters*, i.e. characters which occur only once in the pattern. Although it can be rare for a given character $c \in \Sigma$ to occur only once, especially when m grows, we will show that, by convenient alphabet transformations, it could become very likely to happen. The resulting algorithm is

called Unique-Factor-Matcher (UFM).

As before, let x be a pattern of length m over an alphabet Σ. For each character $c \in \Sigma$, we denote by $f_x(c)$ the number of occurrences of the character c inside x and we say that c is a *unique character* of x if $f_x(c) = 1$.

In addition, for each position of the pattern, we define the *unique distance function* $d : \{0, 1, ..., m-1\} \longrightarrow \{0, 1, ..., m-1\}$ as the function which maps each position i of the pattern to the distance with respect to the rightmost position $j \leq i$ such that $x[j]$ is a unique character in x, or to i itself if such position doesn't exist. If such a unique character does not occur in x we set by default $d(i) = i$. More formally, for $0 \leq i < m$, we have

$$d(i) = min(\{i - j \mid 0 < j \leq i \wedge f_x(x[j]) = 1\} \cup \{i\}).$$

Starting from the previous definition, we define $\bar{d}(c) := max\{d(i) \mid 0 \leq i < m \wedge \ x[i] = c\}$ for each character c appearing in x, while we set $\bar{d}(c) = -1$ if c does not occur in x. For instance, assume $x = \mathsf{pepsi}$ is a string over the alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{i}, \mathsf{e}, \mathsf{p}, \mathsf{s}\}$. Then, we have $\bar{d}(p) = 1$, $\bar{d}(c) = 0$, for any $c \in \{\mathsf{e}, \mathsf{s}, \mathsf{i}\}$ while $\bar{d}(c) = -1$ for $c \in \{\mathsf{a}, \mathsf{b}\}$.

The following two technical lemmas define how unique characters of the pattern can be used to compute, for a given string w, a candidate value μ, such that $\gamma(w) \leq \mu \leq m$. Roughly speaking we prove in Lemma 3 that $|\delta^*(w)| \leq 1$ for any string w ending with a unique character. In addition we prove in Lemma 4 that if the second transition of the suffix automaton of x^r is performed on a character $c \in \Sigma$, then performing $\bar{d}(c) + 1$ transitions is enough to get (at most) a unique active state on the automaton.

Lemma 3. *Let x be a string of length m over Σ and let $S(x) = \langle Q, \Sigma, \delta, I, F\rangle$ be the suffix automaton for x. Moreover, let $c \in \Sigma$ such that $f_x(c) = 1$. Then for each $Q' \in \mathscr{P}(Q)$, $|\delta(Q', c)| \leq 1$ holds.*

```
HashInsert(Ht, x, s, μ)
 1.  c ← x[s + μ − 1]
 2.  n ← NewNode()
 3.  n.len ← μ
 4.  n.start ← s
 5.  n.next ← Ht[c]
 6.  Ht[c] ← n

HashGet(Ht, x, f, μ)
 1.  c ← f[μ − 1]
 2.  n ← Ht[c]
 3.  while n ≠ Nil do
 4.    s ← n.start
 5.    if n.len = μ and x[s..s + μ − 1] = f
 6.      then return n.start
 7.    n ← n.next
 8.  return -1

Preprocessing(x, m)
 1.  for c ∈ Σ do
 2.    F(c) ← 0
 3.    D(c) ← −1
 4.    Ht(c) ← Nil
 5.  for i ← 0 to m − 1 do
 6.    c ← x[i]
 7.    F(c) ← F(c) + 1
 8.  last ← −1
 9.  for i ← 0 to m − 1 do
10.    c ← P[i]
11.    if F(c) = 1 then
12.      D(c) = 0
13.      last = i
14.    else if last ≥ 0
15.      D(c) ← Max(D(c), i − last)
16.    else D(c) ← i
17.  for i ← 0 to m − 1 do
18.    c ← x[i]
19.    μ ← D(c) + 1
20.    if i + 1 ≥ μ then
21.      HashInsert(Ht, x, i + 1 − μ, μ)
22.  return (D, Ht)
```

Figure 5.1: The pseudocode of auxiliary procedures used in the UFM algorithm.

Proof. Since c occurs only once in x, then $c = x[i]$ for some $0 \leq i < m$. By the definition of δ it follows that $\delta(Q', c)$ is nonempty if and only if $q_i \in Q'$. Specifically, when $q_i \in Q'$ we have $\delta(Q', c) = \{q_{i+1}\}$ and $\delta(Q', c) = \emptyset$ otherwise. In both cases

```
UFM(x, m, y, n)
 1.  (D, Ht) ← Preprocessing(P, m)
 2.  j ← m − 1
 3.  while j < n do
 5.    d ← D(x[j])
 6.    if d < 0 then
 7.      j ← j + m
 8.      continue
 9.    μ ← d + 1
10.    p ← HashGet(Ht, y[j − μ..j], μ)
11.    j ← j − μ
12.    if μ = m then j ← j − 1
13.    if p ≥ 0 then
14.      k ← 0
15.      while k < p and y[j − k] = x[p − k − 1] do
16.        k ← k + 1
17.      if k = p do
18.        if k + μ = m then
19.          output j
20.        else
21.          j ← j − k
22.    j ← j + m
```

Figure 5.2: The pseudocode of the UFM algorithm.

$|\delta(Q', c)| \leq 1$ holds. □

Lemma 4. *Let x, w be strings of length m, both over a common alphabet Σ, and let $S(x^r) = \langle Q, \Sigma, \delta, I, F \rangle$ the suffix automaton for x^r. Then, $\bar{d}(w[0]) + 1 \geq \gamma(w)$.*

Proof. Let $\mu = \bar{d}(w[0]) + 1$ and let $Q = |\delta^*(w_\mu)|$. Without loss of generality, we can suppose that $Q \neq \emptyset$. Then, there must exists at least one factor of the pattern $f = x[i..i + \mu - 1]$, for some $0 \leq i \leq m - \mu$, such that $f = w_\mu$. Moreover, by the definition of $\bar{d}$, it follows that such a factor must unique inside x, and consequently $|\delta^*(f)| = |\delta^*(w_\mu)| \leq 1$ also holds, thus implying $\mu \geq \gamma(w)$. □

The preprocessing phase of the UFM algorithm is shown in Figure 5.1. It starts by computing the frequency of each character of the alphabet, in order to find the unique characters of the string x. Then, function $\bar{d}$ is computed in the form of a table D. We recall that, given a text window w of length m, $\mu = D(w[m-1]) + 1$ characters must be read in order to be sure that the suffix automaton for x^r contains

at most one state. When this happens, we need to efficienty recover the starting position of the string $w[m-\mu..m-1]$ inside x. In other words, we need an efficient method to implement the position function Pos. To this purpose, a hash table Ht can be used, storing the starting position of several unique factors of x. In particular, for each position i of the pattern x, factor $x[i-\mu..i]$ of length $\mu = D(x[i]) + 1$ is inserted into the table, whenever $i + 1 \geq \mu$. Note that, character $x[i]$ itself is used as the hash code of factor $x[i-\mu..i]$. In this way, each bucket Ht[c] of the table contains exactly $f_x(c)$ elements, for each $c \in \Sigma$. The Preprocessing takes $O(m)$ space $O(m)$ time to be performed.

The searching phase of the UFM algorithm is shown in Figure 5.2. It follows the structure of the BTSM algorithm, specifically adapted to handle funtions D and hash table Ht. In particular, a window w of length m is slided along the text y. For each window, value $d = D(w[m-1])$ is retrieved. If d is nonnegative, then the suffix of w of length $\mu = d + 1$ is searched in the hash table, in order to get its starting position in the pattern, otherwise the window is instantly shifted by m characters to the right. The searching then proceeds as in the BTSM algorithm.

Regarding the complexity of the algorithm, we observe that a single entry of table Ht could contain up to $\mathcal{O}(m)$ factors in the worst case, each of size equal to $k = \mathcal{O}(m)$. This means that a single call to procedure HashGet at line 10 could require up to $\mathcal{O}(km) = \mathcal{O}(m^2)$ to be performed, leading to an overall complexity of $\mathcal{O}(m^2 n)$ in the worst case. Such a worst case occurs, for example, when searching the pattern $x = a^m$ within the text $y = a^n$, for any $m < n$.

Example 11. *Assume again to match the pattern $x =$ banana against the text window $w =$ anaban, where $\gamma(w^r) = 3$ and $\text{Pos}(\text{ban}^r) = \text{Pos}(\text{nab}) = 0$. The first step of the UFM algorithm begins by computing $\mu = \bar{d}(n) + 1 = 3$ (note that in this particular case the estimation is exact, since $\gamma(w^r) = \mu$ holds). At this point, the computation of the position function is performed by looking up the suffix ban of w in the hash table Ht. A prefix of x is correctly recognized in the second step without performing*

Genome

q/m	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,768	65,536
1	0	0	0	0	0	0	0	0	0	0	0	0
2	4	1	0	0	0	0	0	0	0	0	0	0
4	25	44	66	77	62	28	6	1	0	0	0	0
6	25	53	99	165	234	264	232	155	74	23	4	0
8	24	55	114	223	410	686	986	1,153	1,090	817	492	229
10	22	54	117	240	476	914	1,660	2,775	4,009	4,742	4,528	3,477

Protein

q/m	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,768	65,536
1	5	2	1	0	0	0	0	0	0	0	0	0
2	21	30	33	24	12	6	2	0	0	0	0	0
4	28	58	113	209	358	530	594	443	232	131	74	40
6	26	55	106	189	292	357	297	185	128	102	84	68
8	24	55	116	231	439	770	1,177	1,436	1,204	742	515	417
10	22	54	117	241	483	937	1,759	3,088	4,751	5,762	4,825	2,961

English Text

q/m	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,768	65,536
1	8	8	8	6	6	5	5	4	4	3	2	1
2	22	32	40	42	39	34	31	29	29	27	25	21
4	27	54	95	164	256	377	502	604	654	649	607	565
6	26	55	106	192	325	508	701	824	798	684	581	503
8	24	55	113	221	411	743	1,278	2,006	2,813	3,362	3,340	2,877
10	22	54	115	232	453	855	1,582	2,891	5,051	8,039	11,237	13,306

Table 5.1: Average number of unique characters present in strings of increasing size, for several values of q. For display purposes all numbers have been rounded down. The three tables refer to random strings, extracted from a genome sequence, a protein sequence and a English text.

any additional comparison, since $\textsc{Pos}(\mathsf{nab}) = 0$ *and the text window is advanced by* $m - \mu = 3$ *positions.*

Table 5.1 present the average number of unique characters present in strings of increasing size, for several values of q. The three tables refer to random strings, extracted from a genome sequence, a protein sequence and an English text.

5.1.3 A Relaxed Variant of the UFM Algorithm

The main drawback of the UFM algorihtm consists in the necessity to look up the position p corresponding to the suffix of length μ of the current text window w through an hash table. Apart from the additional overhead from a practical perspective, the use of the hash table is also responsible for the $\mathcal{O}(m^2 n)$ worst case time complexity of the algorithm. In this section we present a useful strategy for

removing this bottleneck in order to improve the performance of the algorithm in practice.

Our strategy to accomplish this goal consists in *relaxing* the first step of the simulation. The pseudo-code of the resulting relaxed variant of the algorithm, named R-UFM (Relaxed Unique Factor Matcher), is shown in Figure 5.3.

As a matter of fact, what is really important during the first step is to quickly locate a unique character c of the pattern x in order to delegate the next part of the scan to the second step of the procedure. Then, after computing the value of μ at line 9, we could simply scan the suffix $w[m-\mu..m-1]$ until either a unique character of the pattern or a character not appearing in x is found.

If a unique character c is found, then we retrieve its position p inside x through a table (which can be implemented using a simple array); if a character not occuring inside x is encountered, we simply advance the current window.

We say that this approach is *relaxed* because, after the first step, the condition $p = \delta^*(w[m-\mu..m-1]^r)$ could also be unsatisfied. Then each candidate occurrence must be verified before reporting a match. However, since μ is expected to be very small in practice, the number of false positives is negligible with respect to the performance gain.

Indeed, the benefit of this strategy goes further than avoiding the hash table lookup (which may involve many string comparisons), since a stopping criteria may be actually met before the entire suffix $w[m-\mu..m-1]^r$ is scanned.

The R-UFM algorithm starts with a call to procedure R-Preprocessing, a simplified version of the Preprocessing procedure used in the standard UFM algorithm, which replaces the hash table Ht with a table P defined in the following way:

$$P[c] = \begin{cases} -1 & if\ f_x(c) = 0 \\ i \mid x[i] = c & if\ f_x(c) = 1 \\ -2 & if\ f_x(c) > 1, \end{cases}$$

```
R-UFM(x, m, y, n)
 1.  (D, P) ← R-Preprocessing(P, m)
 2.  j ← m − 1
 3.  while j < n do
 4.    d ← D(x[j])
 5.    if d < 0 then
 6.      j ← j + m
 7.      continue
 8.    μ ← d + 1, i ← j − μ + 1
 9.    while j > i and P[c] = −2 do
10.      j ← j − 1
11.    p ← P[j]
12.    if P[c] = −2 then j ← j − 1
13.    if p ≥ 0 then
14.      k ← 0
15.      while k < p and y[j − k − 1] = x[p − k − 1] do
16.        k ← k + 1
17.      if k = p do
18.        s ← μ − (j − i)
19.        if k + s = m and y[j..j + s] = x[p..p + s] then
20.          output j
21.        else
22.          j ← j − k
23.    j ← j + m
```

Figure 5.3: The pseudocode of the R-UFM algorithm.

where we remember that $f_x(c)$ is the absolute frequency of character c inside x.

Intuitively, P simply maps each unique character c to its position inside the pattern, also providing a way to distinguish between characters not appearing in x (associate to -1) and characters occurring more than once in x (associated to -2).

Such a definition of the table P allows to implement the scan of our *suffix-scan* approach with a very simple while loop (line 10), which executes until either the entire suffix has been scanned ($j > i$) or the current character c occurs at least twice inside x ($P[c] = -2$).

The overall structure of the algorithm remains the same, except for the additional naive verification at line 20, which is required to verify that the scanned suffix is a factor of x.

Example 12. *Assume, as usual, to match the pattern* $x =$ banana *against the text window* $w =$ anaban*, where* $\gamma(w) = 3$ *and* $\textsc{Pos}(\textsf{ban}) = 0$*. The preprocessing phase*

of the R-UFM algorithm sets $P[a] = P[n] = -2$ *and* $P[b] = 0$. *Then, in the first step at most* $\mu = 3$ *iterations are performed, scanning* w *from right to left, starting from the last character* n. *In this case, the while loop at line 9 halts before this occurs, because character* b *is encountered. At this point, the second step is entirely performed as in the standard UFM algorithm, and a prefix of size* μ *is recognized.*

We note that a nice consequence of the new approach is that the worst case complexity of the algorithm drops to $O(mn)$, since the loop at line 10 can be executed at most $O(m)$ times.

5.1.4 Improving the space usage

In the previous section, we showed how to implement the first step of the simulation in a more efficient manner without using an hash table. However, the necessity to use multiple lookup tables during each pass of the searching phase still remains a weak point of the algorithm. This conflicts, in particular, with an optimal use of processor's L1 cache, whose size typically ranges from 16 to 64 kilobytes. In Table 5.1, we showed that the value of μ is expected to be very small in practice. Since the unique characters are expected to be located at equal intervals inside the pattern string x on the average, we can also expect that the value $\hat{d} = \max_i D[i]$, with $0 < i \leq \sigma$, to be also small in practice. Thus, in order to further reduce the space overhead of the UFM algorithm, we can estimate μ using the constant value $\hat{d} + 1$. In this way, we save half the space during the searching phase, since table for D is not allocated. In Section 5.2, we will show that not only this overestimation doesn't impact on performance significantly, but it also reveals to be most effective in several situations.

5.2 Experimental Results

In this section, we report the results of an extensive experimental comparison of the new presented algorithms against the most efficient solutions known in the literature

for the online exact string matching problem, mostly focusing on those algorithms which make use of the suffix automaton. Experimental evaluation is performed in terms of search speed expressed as Gigabytes per second, excluding any preprocessing time.

5.2.1 Experimental Setting

In our experimental tests the following 12 algorithms (implemented in 53 variants, depending on the values of their parameters) have been compared:

- BNDM$_q$: the Backward-Nondeterministic-DAWG-Matching algorithm [12] implemented with q-grams, for $1 \leq q \leq 6$;
- LBNDM: the Long-BNDM algorithm [7], specifically designed for searching long patterns;
- BSX$_q$: the Backward-Nondeterministic-DAWG-Matching algorithm [12] with Extended Shift [8] implemented using q-grams, with $1 \leq q \leq 6$;
- FBNDM: the Factorized variant [25, 9] of the BNDM algorithm [12];
- BSDM$_q$: the Backward-SNR-DAWG-Matching algorithm [10] using condensed alphabets with groups of q characters, with $1 \leq q \leq 7$;
- WFR$_q$: the Weak Factors Recognition algorithm [15, 48], implemented using q-gram, with $3 \leq q \leq 7$;
- TWFR$_q$: the Tuned Weak Factors Recognition algorithm [48], implemented using q-gram, with $3 \leq q \leq 7$.
- EPSM: the Exact Packed String Matching algorithm [51, 52] based on SIMD instructions;*

*We notice that the EPSM algorithm is designed for simply counting the number of matching occurrences without reporting the corresponding positions.

- PBNDM: the Pruned-BNDM algorithm, introduced in Section 3.1;
- UFM$_q$: the UFM algorithm presented in Section 5.1 implemented with condensed alphabets, using groups of q characters, with $3 \leq q \leq 7$.
- R-UFM$_q$: the relaxed variant of the UFM algorithm presented in Section 5.1.3 implemented with condensed alphabets, using groups of q characters, with $3 \leq q \leq 7$.
- R-UFM$^*_q$: the relaxed variant of the UFM algorithm with fixed distance, presented in Section 5.1.4 and implemented with condensed alphabets, using groups of q characters, with $3 \leq q \leq 7$.

All algorithms have been implemented in the C programming language [†] and have been tested using the SMART tool [26]. All experiments have been executed locally on a computer running Linux Ubuntu 20.04.1 with an Intel Core i5 3.40 GHz processor and 8GB RAM.

Our tests have been run on a genome sequence, a protein sequence, and an English text (each of size 10MB). Such sequences are provided by the SMART research tool and are available online for download (additional details on the sequences can be found in Faro et al. [26]). In the experimental evaluation, patterns of length m were randomly extracted from the searched sequences, with m ranging over the set of values $\{2^i \mid 5 \leq i \leq 16\}$. In all cases, the mean over the search speed (expressed in Gigabytes per seconds) of 500 runs has been reported.

5.2.2 Evaluation

Table 5.2 summarises the results obtained in of our experimental evaluation. Each table is divided into two blocks: the first block (at the top) presents results relative to the most efficient algorithms known in the literature, mainly based on automata

[†]Source code is available at: `https://github.com/ostafen/unique-factor-matcher`

Genome Sequence

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
BNDM$_q$	**3.35**	3.24	3.29	3.22	3.24	3.13	3.26	3.05	3.24	3.20	3.41	3.29
LBNDM	1.69	1.93	2.09	1.07	0.24	0.24	0.27	0.28	0.29	0.29	0.29	0.29
BXS$_q$	3.08	3.24	3.20	3.08	3.24	3.15	3.02	3.26	3.07	3.20	3.26	3.22
FBNDM	1.69	2.49	2.53	2.37	2.40	2.46	2.50	2.48	2.38	2.57	2.57	2.69
BSDM$_q$	2.50	2.70	2.76	2.80	2.76	2.82	2.76	2.83	2.80	2.82	2.90	2.96
EPSM	**3.53**	**3.73**	**3.78**	**4.01**	**4.07**	4.16	4.10	4.19	4.47	4.92	5.38	5.43
WFR$_q$	3.05	**3.64**	3.53	3.69	3.99	4.16	**4.22**	4.28	7.51	11.72	7.71	2.50
TWFR$_q$	2.54	2.69	2.82	2.94	3.24	3.33	3.51	3.78	6.98	10.10	8.62	2.55
PBNDM	1.23	1.26	1.83	1.92	1.90	1.89	1.86	1.88	1.83	1.78	1.82	1.82
UFM$_q$	2.96	3.37	3.66	3.76	**4.07**	**4.19**	4.19	4.19	7.81	**14.6**	**25.4**	**30.8**
R-UFM$_q$	2.96	3.47	3.66	3.73	3.93	**4.19**	**4.25**	**4.41**	**8.14**	**14.6**	**26.6**	**30.8**
R-UFM$^*_q$	2.94	3.49	**3.73**	**3.85**	**4.10**	**4.19**	4.13	**4.44**	**8.25**	**14.6**	**25.4**	**30.8**

Protein Sequence

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
BNDM$_q$	2.51	2.55	2.49	2.55	2.49	2.50	2.53	2.57	2.48	2.53	2.50	2.49
LBNDM	1.98	2.25	2.44	2.57	2.68	2.49	1.26	0.59	0.48	0.46	0.46	0.47
BXS$_q$	2.51	2.50	2.54	2.51	2.51	2.57	2.49	2.53	2.55	2.51	2.45	2.46
FBNDM	2.21	2.41	2.58	2.57	2.58	2.56	2.55	2.50	2.56	2.51	2.56	2.54
BSDM$_q$	**2.54**	2.70	2.78	2.83	2.80	2.82	2.87	2.82	2.82	2.92	2.87	2.94
EPSM	**2.80**	**2.82**	**2.90**	**3.07**	**3.13**	3.20	3.24	3.24	3.47	3.78	4.22	4.73
WFR$_q$	2.34	2.66	2.82	2.86	3.10	3.18	**3.29**	3.18	5.80	8.88	**10.4**	5.80
TWFR$_q$	2.44	**2.71**	2.82	**2.87**	3.10	**3.29**	3.24	3.31	5.69	8.88	10.10	5.86
PBNDM	1.20	1.27	1.69	2.18	2.47	2.56	2.70	2.68	2.68	2.69	2.66	2.63
UFM$_q$	2.26	2.68	**2.86**	2.79	**3.12**	**3.24**	**3.31**	3.31	**6.23**	10.8	**18.9**	**26.6**
R-UFM$_q$	2.36	2.66	2.84	**2.87**	3.12	3.21	**3.31**	**3.53**	**5.9**	**11.2**	**18.9**	**27.9**
R-UFM$^*_q$	2.33	**2.71**	2.83	2.83	3.10	3.21	**3.31**	**3.45**	5.86	**11.1**	**18.9**	**27.9**

English Text

Algo\m	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
BNDM$_q$	2.48	2.56	2.51	2.54	2.50	2.53	2.27	2.48	2.55	2.55	2.55	2.48
LBNDM	1.68	2.16	2.44	2.53	2.65	2.60	2.50	1.87	1.21	0.89	0.71	0.63
BXS$_q$	2.46	2.49	2.55	2.55	2.51	2.48	2.53	2.53	2.50	2.47	2.48	2.49
FBNDM	1.96	2.21	2.50	2.47	2.50	2.46	2.44	2.48	2.44	2.50	2.46	2.43
BSDM$_q$	**2.57**	**2.74**	**2.80**	2.74	2.82	2.78	2.87	2.76	2.90	2.89	2.93	3.02
EPSM	**2.68**	**2.79**	**2.86**	**3.05**	**3.13**	3.20	3.26	3.24	3.57	3.99	4.41	4.97
WFR$_q$	2.33	2.60	2.74	**2.93**	3.04	3.08	**3.41**	3.24	5.80	8.49	8.14	4.16
TWFR$_q$	2.43	2.70	2.76	2.84	3.07	3.20	3.22	3.27	5.80	8.88	8.03	4.28
PBNDM	1.05	1.07	1.59	2.14	2.38	2.45	2.76	2.84	3.64	4.93	6.18	7.40
UFM$_q$	2.10	2.51	2.59	2.84	3.05	3.18	3.33	**3.35**	**6.17**	**11.5**	**20.2**	**27.9**
R-UFM$_q$	2.17	2.60	2.64	2.83	**3.12**	**3.29**	**3.37**	**3.35**	6.10	**11.7**	19.5	**29.3**
R-UFM$^*_q$	2.14	2.56	2.63	2.83	**3.12**	**3.24**	**3.37**	3.31	**6.17**	11.2	**20.2**	**27.9**

Table 5.2: Experimental results obtained for searching on a genome sequence, a protein sequence and an English text. Searching speed is reported in GB/s. The two best results have been boldfaced (in addition, the best result has been underlined).

simulation, while the second block (at the bottom) presents the algorithms introduced in this paper. Best results have been boldfaced for each value of m to ease their localization.

Many of the tested algorithms have been implemented using q-grams, for different values of the parameter q (including most of our algorithms). For such algorithms we report only the best performance obtained among the variants in order not to burden the reading of the data.

Regarding patterns of small and medium size ($m \leq 2^8$), the EPSM algorithm achieves the best results, while the second best results are distributed between the remaining non bit-parallel algorithms and specifically BSDM and WFR. We would like to notice however that the EPSM algorithm is designed for simply counting the number of matching occurrences without reporting the corresponding positions. From this point of view, therefore, the comparison is slightly unfair and the BSDM and WFR algorithms would be the best if the positions of the occurrences of the pattern were required.

As the length of the pattern increases ($m \geq 2^8$), most of the time, best results are achieved by the UFM algorithm and its relaxed variants, with the WRF and TWFR algorithms winning in some situations. However, as the length of the pattern exceeds a certain threshold ($m \geq 2^{11}$), the performances of the UFM family of algorithms diverge rapidly from those observed for the rest of the algorithms and reach a search speed up to 10 times higher than the second best solution.

Among the UFM family of algorithms, it is important to notice that the new variants show equal or superior performances in practice, even if times never deviate too much with respect to the standard UFM algorithm. This result is particularly interesting, if we consider that the new variants also require less space, and have a better worst case complexity.

Regarding the bit-parallel based solutions, we notice that in general they are not competitive in practical cases against the remaining algorithms. However, it is worth to notice that for this class of algorithms our PBNDM algorithm is the best alternative for long strings (starting from $m = 2^{12}$), with the exception of genome sequences, where the approach fails, with BNDM$_q$ being the best choice among bit-parallel solutions.

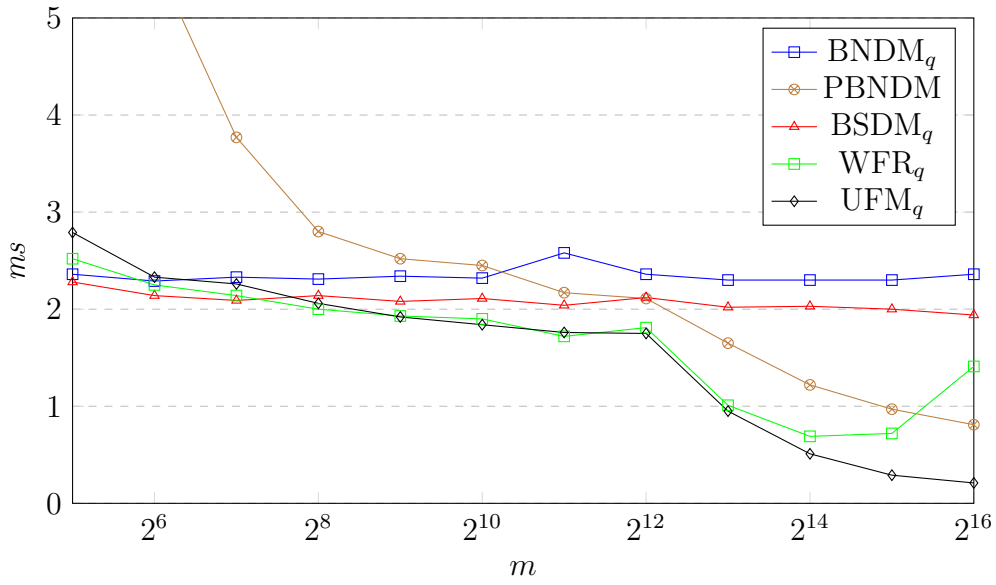

Figure 5.4: The running times (expressed in milliseconds) for several of the algorithms considered in our experiments. The pattern size m, which ranges from 2^5 to 2^{16}, has been reported in logscale to ease readability.

Finally, in order to provide a visual view of our results, we also plotted the running times obtained for an english text (in milliseconds) for some of the algorithms included in our setting in Figure 5.2.2. It immediately catches the eye how the UFM algorithm is the only one to retain its sublinear behaviour until the end. Interestingly, the PBNDM algorithm follows a similar trend (despite a worse performance in terms of time), while all the remaining algorithms maintain a stable performance, except for WFR, which even gets a performance drop after a certain point.

5.2.3 Chapter Summary

In this chapter, we presented a new family of algorithms, starting from the BSTM generic algorithm, based on a novel, two-step simulation of the suffix automaton. With the UFM algorithm and its relaxed variants, we provided concrete implementations of this approach, which not only turn out to be very competitive when compared with the most efficient algorithms known in literature (and, under certain circumstances, the fastest in practice) but also show a sublinear behaviour in

practice.

Future work include the possibility of finding alternative, more efficient strategies for actually implementing the BSTM algorithm, and to adapt the method even to other non standard string matching problems, such as the multiple pattern matching problem.

6

Conclusions

In this thesis we presented a variety of algorithms for solving the exact string matching problem and some of its variations. Each presented method has been extensively compared against the most effective and relevant solutions, depending on the specific sub-problem. Experiments show that the introduced algorithms are competitive in practice and can be easily adapted to variations of the exact string matching problem, thus providing a worthwhile contribution to the field of pattern matching algorithm.

The work presented still admits room for improvement, as outlined in detail in each section of the thesis. Future research directions include further extensions of the presented solutions to other non-standard string matching problems, fine tuning of parameters and improved complexity analysis.

Appendices

A

On the Longest Common Cartesian Substring Problem

A Cartesian tree is associated to a string of numbers and is structured as a heap from which the original string can be recovered. Although Cartesian trees have been introduced 40 years ago, the Cartesian tree matching problem appeared very recently. It consists in finding all substrings of a given text which have the same Cartesian tree as that of a given pattern. In this chapter, the problem of computing the longest common Cartesian substrings of two strings will be addressed and three methods for such problem will be presented. The first method is based on a classical suffix tree construction and solves the problem in randomized linear time and linear space, although the space overhead is quite prohibitive in the case of large strings. The second solution is based on classical dynamic programming, while our third solution is based on a constructive approach. Both of them run in quadratic worst case time but are more space economical in practice. From experimental results, it turns out that the second solution runs faster than the standard suffix tree solution for short strings, while the third solution is more suitable for large strings, when storing a full suffix tree becomes prohibitive.

A.1 Introduction

Cartesian trees have been introduced by Vuillemin [53]. They are associated to strings of numbers and are structured as heaps from which original strings can be

recovered by symmetrical traversal of the trees. It has been shown that they are connected to Lyndon trees [54, 55], to Range Minimum Queries [56] or to parallel suffix tree construction [57]. Recently new results on Cartesian pattern matching appeared [58, 59, 60, 61]. Such problem consists in finding substrings of a text that have the same Cartesian tree as a pattern. Recent studies concern finding periods in Cartesian tree matching [62].

In this chapter we are interested in computing the longest common Cartesian substrings (LCCS) of two strings which means common substrings of maximal length that share the same Cartesian tree. This is useful, for example, to discover interesting patterns and similarities in time series data of stock prices.

A usual linear time method for computing longest common substrings for classical strings consists of building the generalized suffix tree of the two strings and the deepest internal nodes (in terms of string depth) having leaves for suffixes of both strings identify longest common substrings. This method can be applied for computing longest common Cartesian substrings of two strings. However the suffix tree has to be built on the top of a particular representation of the two strings, which uniquely maps each Cartesian tree to a string of integers, the parent-distance representation, meaning that classical suffix tree construction algorithms cannot be used.

In this chapter we propose two quadratic worst case time algorithms for computing the longest Cartesian substrings of two strings that use only constant extra space in addition to the two strings and their parent-distance representation or their Cartesian trees. However, despite their quadratic worst case time complexity, from our experimental results it turns out that for short strings and in most practical settings our alternative based on dynamic programming is faster than the suffix tree based method, while our constructive solution is faster in the case of large strings.

The chapter is organized as follows. Section A.2 presents the notations and definitions used throughout the rest of the article. Section A.3 presents the method for constructing the Cartesian tree of a string. Section A.4 briefly presents the suffix

tree based method for computing longest common Cartesian substrings. Section A.5 and Section A.6 describe our new alternative solutions. Section A.7 presents experimental results. Section A.7.4 concludes the appendix.

A.2 Notations and Definitions

We assume that a string is a sequence of symbols drawn from an alphabet Σ, of size σ, which can be seen as a set of numeric values. We also assume that a comparison between any two symbols of the alphabet can be done in constant time.

We indicate the length of a string x with the symbol $|x|$. For a string x of length m, $x[i]$ represents the i-th symbol of x, for $1 \leq i \leq m$, and $x[i\,..\,j]$ represents the substring of x starting from position i and ending at position j, for $1 \leq i \leq j \leq m$. We denote by $x_i = x[1\,..\,i]$ the prefix of x of length i and by $x^i = x[m-i+1\,..\,m]$ the suffix of x of length i, with $1 \leq i \leq m$. For simplicity, we will simply write x_i^r instead of $(x_i)^r$ to denote the substring of length r ending at position i, i.e. $x[i-r+1..i]$, with $1 \leq i \leq m$ and $0 \leq r \leq i$.

We also emphasize that, to improve readability, we will use either the notation $x[i-r+1..i]$ or x_i^r, depending on the context, to denote the substring of x with length r ending at position i.

Given two strings x and y of length m and n, respectively, we indicate with symbol $lcp(x, y)$ the length of the longest prefix common to x and y.

Let x be a string of length m. The Cartesian tree $\textsc{Ct}(x)$ of x is the binary tree where:

- the root corresponds to the index i of the minimal element of x (if there are several occurrences of the minimal element, the leftmost one is chosen);
- the left subtree of the root corresponds to the Cartesian tree of $x_{i-1} = x[1\,..\,i-1]$;
- the right subtree of the root corresponds to the Cartesian tree of $x^{m-i} = x[i+1\,..\,m]$.

For simplicity in what follows we will use the symbol $x[i]$ to refer to both the i-th character of x and the node of $\textsc{Ct}(x)$ whose key is i, depending on the context. We will refer to the root of a Cartesian Tree T as $\textsc{root}[T]$ and we assume that such node can be always accessed in constant time. When T is empty, we agree, for simplicity, that $\textsc{root}[T] = \textsc{Nil}$.

The following definition of the *right path* of a binary tree is particularly relevant to this paper.

Definition 10 (Right path)**.** *The right path, $rp(T)$, of a binary tree T is the sequence of nodes encountered starting from the root of the tree and always going right.*

The right path of a Cartesian tree of a string x, with length m, always ends with the last character of the string, i.e. $x[m]$. We also refer to the rightmost node of a Cartesian Tree T as $\textsc{rmn}[T]$ and we also assume that such node can be always accessed in constant time. We use the notation $\textsc{size}[T]$ to indicate the number of nodes stored in a Cartesian tree T, assuming also that this information is always accessible in constant time.

In addition we indicate with symbol $\textsc{right}[n]$ and $\textsc{left}[n]$ the right and the left child of a node $n \in T$, respectively. The symbol $\textsc{index}[n]$ is used to indicate the index i of the character in x such that $x[i] = n$.

In the following, we use for simplicity the notation $rp(x)$ to indicate the right path of the Cartesian Tree of a string x, i.e. $rp(x) = rp(\textsc{Ct}(x))$ and write $len(rp(x))$ to denote its length. Similarly we use $\textsc{rmn}[x]$ to indicate $\textsc{rmn}[\textsc{Ct}(x)]$.

A.3 Building a Cartesian Tree

The construction of the Cartesian tree of a string x of length m can be done by means of an iterative procedure which iterates over the elements of x, proceeding from left to right, and computes the Cartesian tree of x_{i+1} from the Cartesian tree of x_i, for $1 \leq i < m$.

To better describe such approach we observe that the Cartesian tree of $x[1]$ consists of a single node, while $\textsc{Ct}(x_{i+1})$ can be computed from $\textsc{Ct}(x_i)$ by identifying the number of nodes that are on the right path of $\textsc{Ct}(x_i)$ but not on the right path of $\textsc{Ct}(x_{i+1})$.

Going deeper into the details, let $rp(\textsc{Ct}(x_i)) = \langle x[j_1], x[j_2], \ldots, x[j_k] \rangle$ be the right path of $\textsc{Ct}(x_i)$, with $1 \leq k \leq i$ and where $x[j_1] = \textsc{root}[\textsc{Ct}(x_i)]$ and $j_k = i$. We can distinguish three cases, as depicted in Figure A.1:

1. if $x[i+1] < x[j_1]$, then $x[i+1]$ becomes the new root of $\textsc{Ct}(x_{i+1})$;

2. if $x[j_u] < x[i+1] < x[j_{u+1}]$, for some $1 \leq u < k$, then $x[i+1]$ is the smallest value on the right of $x[j_u]$ and all elements in the substring $x[j_u + 1 \,..\, i]$ are greater than $x[i+1]$. Then $x[i+1]$ is inserted as the right child of $x[j_u]$ and the subtree rooted at $x[j_{u+1}]$ is made the left child of $x[i+1]$.

3. if $x[i+1] > x[j_k]$, then $x[i+1]$ is greater than the rightmost character in the right path of x_i, so that $x[i+1]$ is added as the right child of $x[j_k]$ in $\textsc{Ct}(x_{i+1})$.

The following lemma, which states the time complexity of the incremental construction, is particularly relevant for the analysis of our third algorithm presented in Section A.6.

Lemma 5 ([63])**.** *Given a numeric string x, of length m, the Cartesian tree of x can be computed in $O(m)$-time.*

Proof. In order to analyse the time complexity for the computation of the Cartesian tree of a string we refer to the algorithm whose pseudo-code, presented in Figure A.2, was described in [63].

The for loop of line 3 is executed $m-1$ times. The while loop of line 4 consists of scanning upward the right path of the tree. Each iteration of this loop decreases the current length of the right path by one and the scanned node will not be scanned again thus the overall number of iterations of the while loop over all the iterations

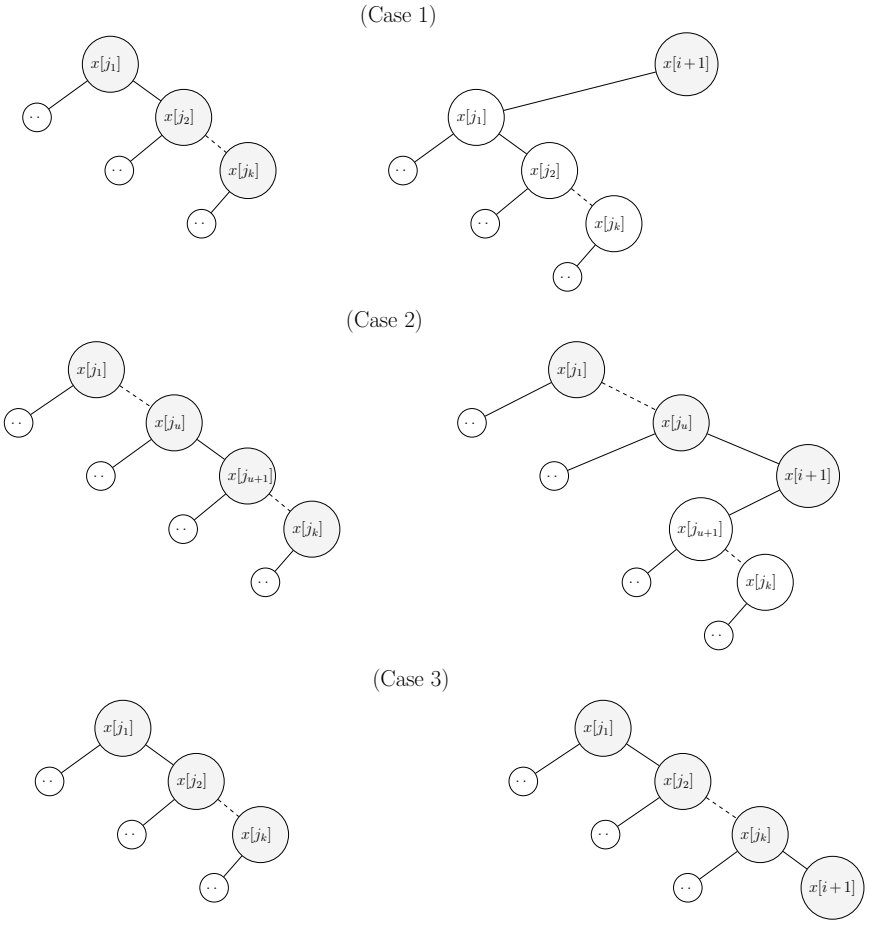

Figure A.1: The three cases occurring when computing $\textsc{Ct}(x_{i+1})$ (to the right) from $\textsc{Ct}(x_i)$ (to the left). (Case 1) $x[i+1]$ is less than the current root of the tree and it is added as the new root; (Case 2): we have $x[j_u] < x[i+1] < x[j_{u+1}]$; (Case 3): we have $x[i+1]$ is greater than $x[i]$. In all cases $x[i+1]$ becomes the last node of the right path of the tree. Nodes belonging to the right path are filled in gray.

of the for loop is bounded by m. All the other operations can be done in constant time. Therefore the time complexity of the algorithm for building the Cartesian tree of a string of length m is $O(m)$. □

Example 13. *Let* $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$ *be a numeric sequence of length* 8. *Figure A.3 shows the Cartesian trees computed by such incremental procedure.*

For the sake of completeness we point out that the Cartesian tree of a string x

```
TREE-EXTEND-RIGHT(T, e)
 1.  p ← New-Node()
 2.  Element(p) ← e
 3.  q ← RMN[T]
 4.  while q ≠ Nil and e < Element(q) do
 5.      q ← Parent(q)
 6.  if q = Nil then
 7.      Left(p) ← ROOT[T]
 8.      Parent(ROOT[T]) ← p
 9.      ROOT[T] ← p
10.  else
11.      if Right(q) ≠ Nil
12.          Parent(Right(q)) ← p
13.      Left(p) ← Right(q)
14.      Right(q) ← p
15.      Parent(p) ← q
16.  RMN[T] ← p
17.  return T

BUILD-CARTESIAN-TREE(x, m)
1.  T ← Empty-Tree()
2.  RMN[T] ← ROOT[T]
3.  for i ← 1 to m do
4.      T ← Tree-Extend-Right(T, x[i])
5.  return T
```

Figure A.2: The iterative procedure BUILD-CARTESIAN-TREE for building the Cartesian tree of a string x of length m. A node of the Cartesian tree has 4 components: PARENT, ELEMENT, LEFT and RIGHT, where ELEMENT refers to the integer value of the new inserted element. The function NEW-NODE() creates a new node and initializes its 4 components to NIL.

can be also computed by iterating over the elements of x and proceeding from right to left (instead of from left to right) by means of a symmetrical procedure.

For realizing the algorithm given in Figure A.2 the right path can be implemented as a stack so that there is no need to have a link to its parent for each node of the tree.

Instead of building the Cartesian tree for every position in the text to solve Cartesian tree matching, Park *et al.* [58] introduced the following representation for a Cartesian tree.

Definition 11 (Parent-distance representation)**.** *The parent-distance representation of a string $x[1\,..\,m]$ is a function PD_x, which is defined as follows:*

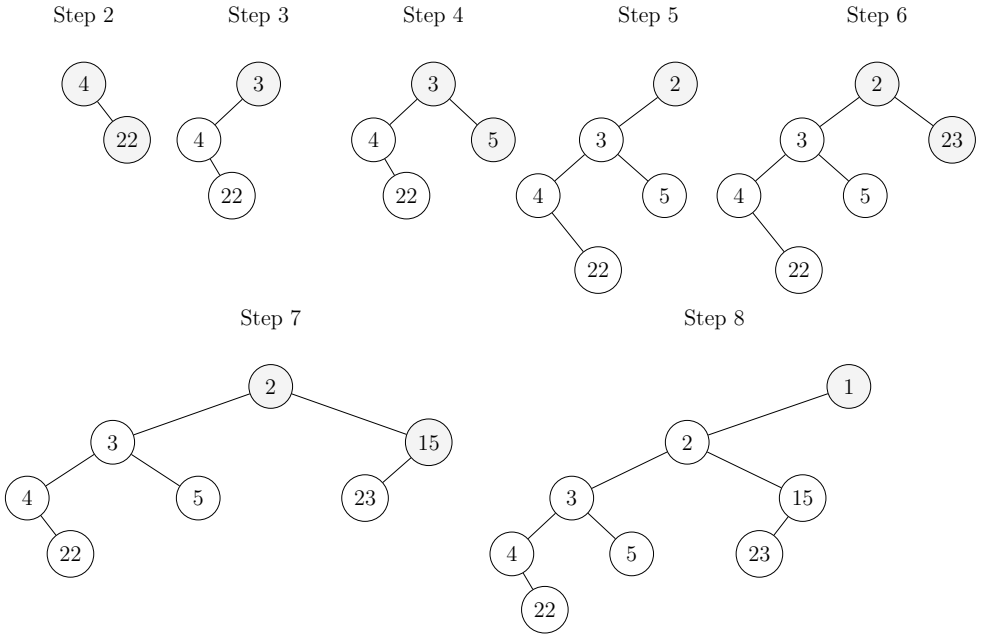

Figure A.3: Different steps of the construction of $\textsc{Ct}(x)$ when $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$.

$$PD_x(i) = \begin{cases} i - \max_{1 \le j < i}\{j \mid x[j] \le x[i]\} & \text{if such } j \text{ exists} \\ 0 & \text{otherwise.} \end{cases}$$

Example 14. *The following table gives the parent-distance representation for* $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$.

i	1	2	3	4	5	6	7	8
$x[i]$	4	22	3	5	2	23	15	1
$PD_x(i)$	0	1	0	1	0	1	2	0

Since the parent-distance representation has a one-to-one mapping to the Cartesian tree [58], it can replace the Cartesian tree without any loss of information. It can be computed and stored in a table in linear time and space using the algorithm given in Figure A.4 (see [58]).

```
COMPUTE-PARENT-DISTANCE(x, m)
 1.  St ← Empty-Stack()
 2.  for i ← 1 to m do
 3.      while St is not empty do
 4.          (value, index) ← St.top
 5.          if value ≤ x[i] then
 6.              PD_x[i] ← i − index
 7.              break
 8.          St.pop
 9.      if St is empty then
10.          PD_x[i] ← 0
11.      St.push((x[i], i))
12.  return PD_x
```

Figure A.4: Computation of the parent-distance representation for a string x of length m. While scanning the input string from left to right, characters are stored in a stack. The main idea of the procedure is that if two characters $x[i]$ and $x[j]$ for $i < j$ satisfy $x[i] > x[j]$, $x[i]$ cannot be the parent of $x[k]$ for any $k > j$. Therefore, each value $x[i]$ is kept in the stack only until, while scanning from left to right, no such $x[j]$ is found.

A.4 A Suffix Tree Based Approach

In this section, we present a randomized linear time algorithm for the computation of the longest common Cartesian substrings of two strings, based on the suffix tree data structure.

The Cartesian suffix tree of a string has to be built on the parent-distance representation of the string. The parent-distance representation of a substring of x can be easily computed as follows (see [58]):

$$PD_{x[i..j]}[k] = \begin{cases} 0 & \text{if } PD_x[i+k-1] \geq k \\ PD_x[i+k-1] & \text{otherwise.} \end{cases}$$

This can be used for getting all the suffixes of the parent-distance representation for building its suffix tree. However classical linear time suffix tree construction algorithms cannot be used because the distinct right context property should hold in order to apply these algorithms, which means that the suffix link of every internal node should point to an explicit node. In other words if $lcp(x[i\,..\,m], x[j\,..\,m]) = \ell$ then $lcp(x[i+1\,..\,m], x[j+1\,..\,m]) = \ell - 1$ for $1 \leq i, j \leq m$. The Carte-

sian suffix tree does not have the distinct right context property meaning that if $lcp(PD_x[i\,..\,m], PD_x[j\,..\,m]) = \ell$ then $lcp(PD_x[i+1\,..\,m], PD_x[j+1\,..\,m])$ can be greater than $\ell - 1$.

Example 15. *With* $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$,

$$lcp(PD_{x[5..8]}, PD_{x[6..8]}) = lcp(\langle 0, 1, 2, 0\rangle, \langle 0, 0, 0\rangle) = 1$$

and

$$lcp(PD_{x[6..8]}, PD_{x[7..8]}) = lcp(\langle 0, 0, 0\rangle, \langle 0, 0\rangle) = 2.$$

A randomized construction algorithm, running in linear time with high probability, for the suffix tree with missing suffix links was first given in [64]. It can be used for building Cartesian suffix trees. These Cartesian suffix trees can be used to compute longest common Cartesian substrings of two strings x and y: for instance, by building the generalized Cartesian suffix tree of PD_x and PD_y. Then the internal nodes with the largest string depth having leaves corresponding to both PD_x and PD_y identify longest common Cartesian substrings of x and y. This can be done during a traversal of the tree. Thus longest common Cartesian substrings of two strings can be computed in randomized linear time and in linear space. The space overhead, in addition to the two strings and their parent-distance representation, is constituted by the generalized suffix tree.

Theorem 1. *Given two strings x and y of numbers of length m and n respectively, the longest substrings of x and y having the same Cartesian tree can be computed in randomized linear time and in space $O(m + n)$.*

A.5 Computing the LCCS by Dynamic Programming

Let x and y be two strings of length m and n respectively. We are interested in finding the longest substrings of x and y having the same Cartesian tree. We will describe a solution based on dynamic programming. This solution also uses the

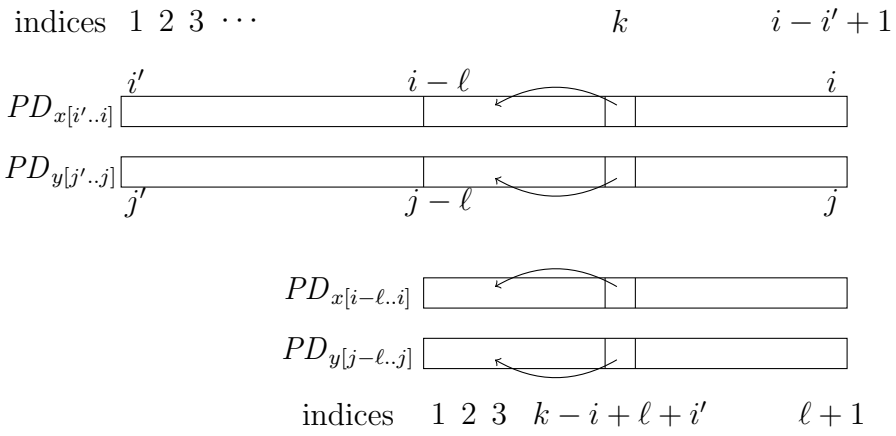

Figure A.5: $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] < k - i + \ell$

parent-distance representation. We will show that the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree can easily be computed from the longest suffixes of $x[1\,..\,i-1]$ and $y[1\,..\,j-1]$ having the same Cartesian tree. Let us first state that if two substrings have the same parent-distance so have their suffixes.

Fact 2. *If* $PD_{x[i'..i]} = PD_{y[j'..j]}$ *then* $PD_{x[i-\ell..i]} = PD_{y[j-\ell..j]}$ *for* $0 < \ell \leq i - i'$.

Proof. The statement trivially also holds for $\ell = i - i'$. Let $0 < \ell < i - i'$ and $i - \ell - i' + 1 < k < i - i'$, then only two cases have to be considered:

1. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] < k-i+\ell$ then $PD_{x[i-\ell..i]}[k-i+\ell+i'] = PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] = PD_{y[j-\ell..j]}[k - i + \ell + i']$ (see Figure A.5) or

2. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] \geq k - i + \ell$ then $PD_{x[i-\ell..i]}[k - i + \ell + i'] = 0 = PD_{y[j-\ell..j]}[k - i + \ell + i']$ (see Figure A.6).

In both cases $PD_{x[i-\ell..i]}[p] = PD_{y[j-\ell..j]}[p]$ for $1 \leq p \leq \ell + 1$ thus $PD_{x[i-\ell..i]} = PD_{y[j-\ell..j]}$. □

We can now state the next lemma.

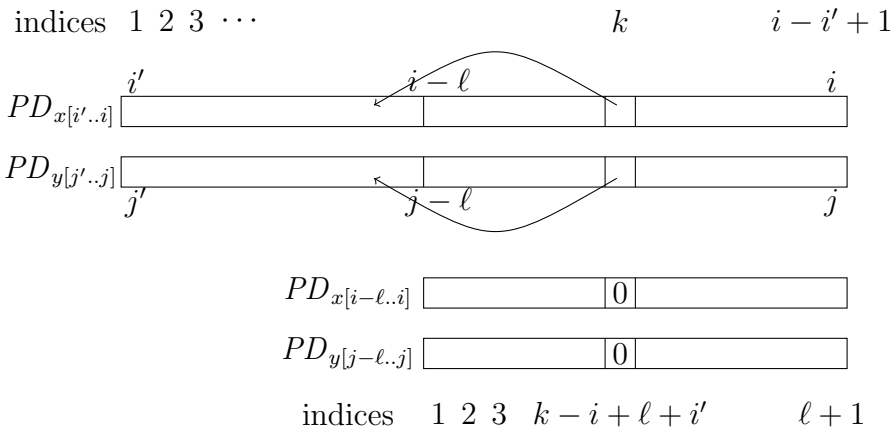

Figure A.6: $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] \geq k - i + \ell$

Lemma 6. *Let $x[i'..i-1]$ and $y[j'..j-1]$ be the longest suffixes of $x[1..i-1]$ and $y[1..j-1]$ having the same Cartesian tree. Let $\ell_x = PD_{x[i'..i]}[i - i' + 1]$, $\ell_y = PD_{y[j'..j]}[j - j' + 1]$ and $\ell = \min\{\ell_x, \ell_y\}$.*

The longest suffixes of $x[1..i]$ and $y[1..j]$ having the same Cartesian tree are:

1. *$x[i'..i]$ and $y[j'..j]$ if $\ell_x = \ell_y$;*
2. *$x[i - \ell_y + 1..i]$ and $y[j - \ell_y + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_x = 0$;*
3. *$x[i - \ell_x + 1..i]$ and $y[j - \ell_x + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_y = 0$;*
4. *$x[i - \ell + 1..i]$ and $y[j - \ell + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_x \neq 0$ and $\ell_y \neq 0$.*

Proof. If $x[i'..i-1]$ and $y[j'..j-1]$ have the same Cartesian tree then $PD_{x[i'..i-1]} = PD_{y[j'..j-1]}$. We will detail the 4 cases:

1. If $\ell_x = \ell_y$ then $PD_{x[i'..i]} = PD_{y[j'..j]}$ and thus $x[i'..i]$ and $y[j'..j]$ have the same Cartesian tree. Thus $x[i'..i]$ and $y[j'..j]$ are the longest suffixes of $x[1..i]$ and $y[1..j]$ having the same Cartesian tree. Longer suffixes with the same Cartesian tree would contradict the maximality of the length of $x[i'..i-1]$ and $y[j'..j-1]$.

2. If $\ell_x \neq \ell_y$ and $\ell_x = 0$ then $PD_{y[k..j]}[j-k+1] = \ell_y \neq \ell_x$ for $j' \leq k \leq j - \ell_y$ and $PD_{y[j-\ell_y+1..j]}[\ell_y] = 0 = \ell_x$. Thus by Fact 2, $x[i-\ell_y+1\,..\,i]$ and $y[j-\ell_y+1\,..\,j]$ are the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree.

3. Symmetric to 2.

4. If $\ell_x \neq \ell_y$ and $\ell_x \neq 0$ and $\ell_y \neq 0$ then $PD_{x[k..i]}[i-k+1] \neq PD_{y[k'..j]}[j-k'+1]$ for $i' \leq k \leq i - \ell$ and for $j' \leq k' \leq j - \ell$ and $PD_{x[i-\ell+1..i]}[\ell] = 0 = PD_{y[j-\ell+1..j]}[\ell]$. Thus by Fact 2, $x[i-\ell+1\,..\,i]$ and $y[j-\ell+1\,..\,j]$ are the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree.

□

A diagonal d corresponds to a pair of factors $x[i\,..\,s]$ and $y[j\,..\,t]$ such that $1 \leq i \leq s \leq m$, $1 \leq j \leq t \leq n$, $|x[i\,..\,s]| = |y[j\,..\,t]|$ and $d = j - i$. Since factors of length 1 always constitute common Cartesian substrings between two strings, the algorithm Compute-Longest-Cartesian-Substring given in Figure A.7 processes diagonals from $-m+2$ to $n-2$. For each diagonal it uses 4 indices i, i', j and j' to compare $x[i'\,..\,i]$ and $y[j'\,..\,j]$ starting with the first factors of length 2 of the current diagonal. It updates indices i, i', j and j' according to Lemma 6. It only computes the length ℓ of the longest common Cartesian substrings of x and y but could easily be computed to report two positions i in x and j in y such that $x[i\,..\,i+\ell-1]$ and $y[j\,..\,j+\ell-1]$ have the same Cartesian tree with the same complexities.

Theorem 3. *Given two strings x and y of numbers of length m and n respectively, the longest substrings of x and y having the same Cartesian tree can be computed in time $O(mn)$ and in space $O(m+n)$.*

Proof. Given x and y, the parent-distance representations PD_x and PD_y can be computed in space and time $O(m)$ and $O(n)$ respectively. Then the results of Lemma 6 can be applied on any pair of ending positions of substrings of x and y. There are

```
COMPUTE-LONGEST-CARTESIAN-SUBSTRING(x, m)
 1.  PD_x ← Compute-Parent-Distance(x, m)
 2.  PD_x ← Compute-Parent-Distance(y, n)
 3.  maxlength ← 1
 4.  for d ← −m + 2 to n − 2 do
 5.      (i, j) ← (2, 2)
 6.      if d < 0 then
 7.          i ← −d + 2
 8.      else if d > 0
 9.          j ← d + 2
10.      (i′, j′) ← (i − 1, j − 1)
11.      while i ≤ m and j ≤ n do
12.          (ℓ_x, ℓ_y) ← (PD_{x[i′..i][i−i′+1]}, PD_{y[j′..j][j−j′+1]})
13.          if ℓ_x ≠ ℓ_y then
14.              if ℓ_x = 0 then
15.                  ℓ = ℓ_y
16.              else if ℓ_y = 0 then
17.                  ℓ = ℓ_x
18.              else ℓ ← min(ℓ_x, ℓ_y)
19.              (i′, j′) ← (i − ℓ + 1, j − ℓ + 1)
20.          else
21.              maxlength ← max(maxlength, i − i′ + 1)
22.          (i, j) ← (i + 1, j + 1)
23.  return maxlength
```

Figure A.7: Computation of the length of the longest Cartesian substrings of x of length m and y of length n.

$O(mn)$ such pairs and the computation for one pair takes constant time if the result for the correct previous pair is available. The result follows by performing the computation diagonal-wise i.e. by considering increasing values i and j such that $j - i = d$ for $-m + 1 \leq d \leq n - 1$. □

Example 16. *$x = \langle 70, 84, 63, 74, 86, 97\rangle$ and $y = \langle 50, 83, 76, 39, 90, 67, 1, 6\rangle$. Then $PD_x = \langle 0, 1, 0, 1, 1, 1\rangle$ and $PD_y = \langle 0, 1, 2, 0, 1, 2, 0, 1\rangle$. Let us look at starting positions $i' = 3$ in x and $j' = 4$ in y:*

- *$PD_{x[3..3]}[1] = 0$ and $PD_{y[4..4]}[1] = 0$*
- *$PD_{x[3..4]}[2] = 1$ and $PD_{y[4..5]}[2] = 1$*
- *$PD_{x[3..5]}[3] = 1$ and $PD_{y[4..6]}[3] = 2$ then i' becomes 5 and j' becomes 6*
- *$PD_{x[5..6]}[2] = 1$ and $PD_{y[6..7]}[2] = 0$ then i' becomes 6 and j' becomes 7*

thus the longest Cartesian substring of $x[3\,..\,6]$ and $y[4\,..\,7]$ has length 2.

A.6 A Constructive Approach for the LCCS Problem

In this section, we present a third approach for finding the longest common Cartesian substrings of two equal length strings x and y. Such approach is based on the explicit construction of the Cartesian trees associated with the longest common Cartesian substrings of the two strings through the incremental approach presented in Section A.3.

As before, let x and y be two strings of length m and n, respectively. We indicate by $\textsc{Lcx}(x, y)$ the length r of the longest suffixes of x and y which share the same Cartesian tree, so that $\textsc{Ct}(x^r) = \textsc{Ct}(y^r)$. Moreover, we denote by $\textsc{LcCs}(x, y)$ the length of the longest common Cartesian substring of the two strings x and y. Specifically our solution is based on the following relation:

$$\textsc{LcCs}(x, y) = \max\{\textsc{Lcx}(x_i, y_j) \mid 1 \leq i, j \leq m\}, \tag{A.1}$$

Roughly speaking, we compute the length of the longest substring sharing the same Cartesian tree, between x and y, as the maximum length of the longest suffixes (sharing the same Cartesian tree) of any couple of prefixes x_i and y_j.

The main idea behind an efficient computation of such values is given in the following two technical lemmas which define how to efficiently compute $\textsc{Lcx}(x_i, y_j)$ from $\textsc{Lcx}(x_{i-1}, y_{j-1})$.

Lemma 7. *Let z and w be two strings of equal length m such that $\textsc{Ct}(z_{m-1}) = \textsc{Ct}(w_{m-1})$. If $len(rp(z)) = len(rp(w))$ then we have that $\textsc{Ct}(z) = \textsc{Ct}(w)$.*

Proof. Let $rp(z) = \langle z[i_1], z[i_2], ..., z[i_{h_1}]\rangle$ and $rp(w) = \langle w[j_1], w[j_2], ..., w[j_{h_2}]\rangle$ be the right paths of $\textsc{Ct}(z)$ and $\textsc{Ct}(w)$, respectively, and let $rp(z_{m-1}) = \langle z[l_1], z[l_2], ..., z[l_h]\rangle$ and $rp(w_{m-1}) = \langle w[g_1], w[g_2], ..., w[g_h]\rangle$ be the right paths of $\textsc{Ct}(z_{m-1})$ and $\textsc{Ct}(w_{m-1})$, respectively, where $1 \leq h \leq m-1$ and $1 \leq h_1, h_2 \leq h+1$ (see Figure A.8).

If $len(rp(z)) = len(rp(w))$ then we have that $h_1 = h_2$. This implies that $i_k = l_k$, for $1 \leq k \leq h_1 - 1$ and $j_k = g_k$, for $1 \leq k \leq h_2 - 1$. Thus $\textsc{Ct}(z)$ and $\textsc{Ct}(w)$ only

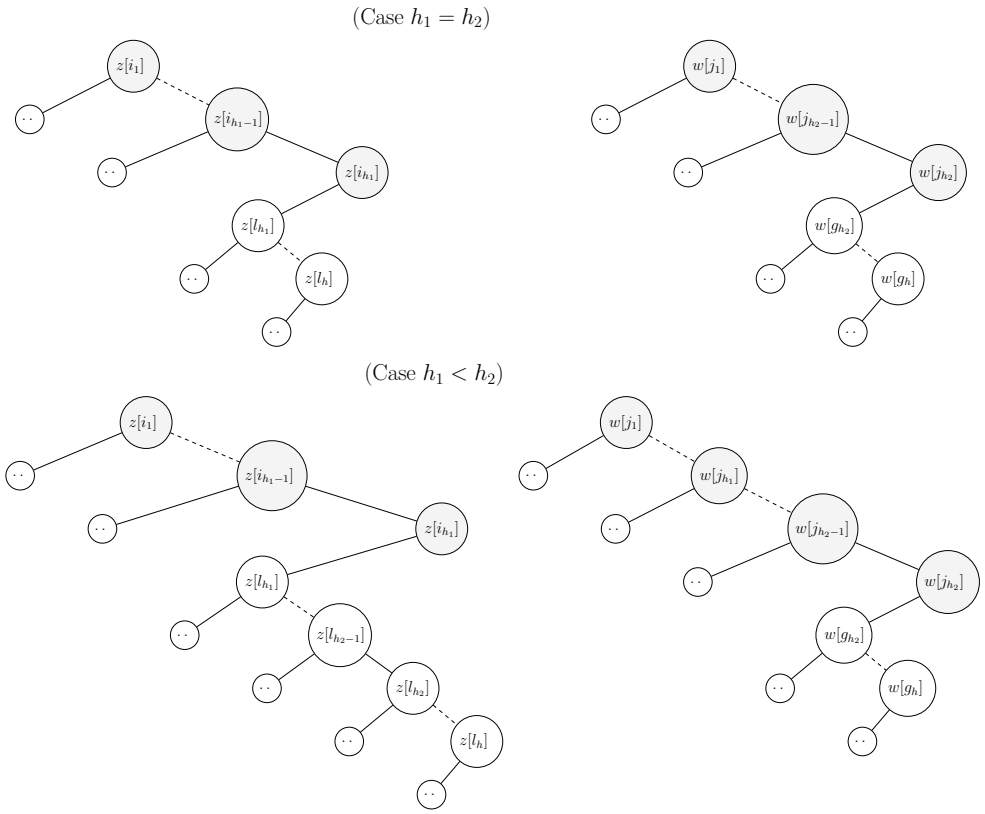

Figure A.8: The two Cartesian trees for z and w in the case of Lemma 7 and Lemma 8, respectively. When $h_1 = h_2$ (Lemma 7), the two Cartesian trees are equivalent. When $h_1 < h_2$ (Lemma 8), the subtree rooted at node $w[j_{h_2}]$ corresponds to the Cartesian tree for the longest suffixes of z and w having the same cartesian tree. Nodes belonging to the right path are filled in gray.

Figure A.9: The Cartesian trees for $\textsc{Ct}(z^{r+1})$ and $\textsc{Ct}(w^{r+1})$ which are at the base of the proof by contradiction of Lemma 8.

differ from $\text{CT}(z_{m-1})$ and $\text{CT}(w_{m-1})$ for the subtrees rooted at nodes $z[i_{h_1}] = z[m]$ and $w[j_{h_2}] = w[m]$.

Moreover, $z[l_{h_1}]$ is the left child of $z[m]$ in $\text{CT}(z)$ and $w[g_{h_2}]$ is the left child of $w[m]$ in $\text{CT}(w)$. Since, by hypothesis, $\text{CT}(z_{m-1}) = \text{CT}(w_{m-1})$, then the Cartesian trees rooted in $z[l_{h_1}]$ and $w[g_{h_2}]$ are equal. Hence, the subtrees rooted at $z[i_{h_1}]$ and $w[j_{h_2}]$ are also equal, proving that $\text{CT}(z) = \text{CT}(w)$. □

Lemma 8. *Let z and w be two strings of equal length m such that $\text{CT}(z_{m-1}) = \text{CT}(w_{m-1})$. If $len(rp(z)) \neq len(rp(w))$, then the length of the longest common Cartesian substrings of z and w is given by the number of nodes in the subtree rooted in the rightmost node of T', where*

$$T' = \begin{cases} \text{CT}(z) & \text{if } len(rp(z)) > len(rp(w)) \\ \text{CT}(w) & \text{otherwise.} \end{cases}$$

Proof. Let $rp(z) = \langle z[i_1], z[i_2], ..., z[i_{h_1}] \rangle$ and $rp(w) = \langle w[j_1], w[j_2], ..., w[j_{h_2}] \rangle$ be the right paths of $\text{CT}(z)$ and $\text{CT}(w)$, and also let $rp(z_{m-1}) = \langle z[l_1], z[l_2], ..., z[l_h] \rangle$ and $rp(w_{m-1}) = \langle w[g_1], w[g_2], ..., w[g_h] \rangle$ be the right paths of $\text{CT}(z_{m-1})$ and $\text{CT}(w_{m-1})$, respectively, where $1 \leq h \leq m-1$ and $1 \leq h_1, h_2 \leq h+1$.

If $h_1 \neq h_2$, we can suppose without loss of generality that $h_1 < h_2$. Let T be the Cartesian tree rooted at node $w[j_{h_2}] = w[m]$ in $\text{CT}(w)$ and let r be the number of nodes contained in such a tree (see Figure A.8).

Let us first show that T is the Cartesian tree corresponding to both the suffixes of z and w of length r, i.e. $T = \text{CT}(z^r) = \text{CT}(w^r)$. Since $\text{CT}(z_{m-1}) = \text{CT}(w_{m-1})$, then $\text{CT}(z_{m-1}^{r-1})$ and $\text{CT}(w_{m-1}^{r-1})$ are also equal and their roots are $w[g_{h_2}]$ and $z[l_{h_2}]$, respectively.

Since $h_1 < h_2$, then $z[i_{h_1}] = z[m] < z[l_{h_2}]$ must hold and, consequently, $z[l_{h_2}]$ is the left child of $z[m]$ in $\text{CT}(z^r)$. On the other hand, $w[g_{h_2}]$ is the left child of $w[m]$ in $\text{CT}(w^r)$. Since $z[m]$ and $w[m]$ are the rightmost nodes of $\text{CT}(z^r)$ and $\text{CT}(w^r)$, their right subtrees are also equal, being empty, and hence $T = \text{CT}(z^r) = \text{CT}(w^r)$.

To complete the proof, we show that no other $r' > r$ could exist such that

$\textsc{Ct}(z^{r'}) = \textsc{Ct}(w^{r'})$, i.e. z^r and w^r are the longest suffixes of z and w having the same Cartesian tree.

By contradiction, suppose that $\textsc{Ct}(z^{r'}) = \textsc{Ct}(w^{r'})$ for some $r' \geq r+1$. Then, $\textsc{Ct}(z^{r+1}) = \textsc{Ct}(w^{r+1})$ too, i.e. the suffixes of z and w of size $r+1$ have the same Cartesian tree.

Consider now $\textsc{Ct}(w^{r+1})$. It can be obtained by adding T as the right subtree of the node $w[j_{h_2-1}]$, since $w[j_{h_2-1}]$ is the parent of $w[j_{h_2}]$ in $\textsc{Ct}(w)$, and thus $w[j_{h_2-1}] \leq w[j_{h_2}]$. Moreover, since $\textsc{Ct}(w^{r+1}) = \textsc{Ct}(z^{r+1})$, it should be possible to obtain $\textsc{Ct}(z^{r+1})$ by making T the right subtree of node $z[l_{h_2-1}]$, thus by making $z[i_{h_1}]$ its right child. However, since $h_1 < h_2$, we know that at least $z[l_{h_2-1}]$ is contained on the left subtree of node $z[i_{h_1}]$ in $\textsc{Ct}(z)$ (Fig. A.8), meaning that $z[i_{h_1}] < z[l_{h_2-1}]$. This implies that $z[l_{h_2-1}]$ must be the left child of $z[i_{h_1}] = z[m]$ in $\textsc{Ct}(z^{r+1})$ (Fig. A.9), directly leading to $\textsc{Ct}(z^{r+1}) \neq \textsc{Ct}(w^{r+1})$, which is a contradiction.

Thus, we can conclude that z^r and w^r are the longest suffixes of z and w having the same Cartesian tree. □

Given two strings x and y of length n and m, Lemma 7 and Lemma 8 provide us a systematic method to compute the longest substrings of x and y having the same Cartesian tree. The algorithm is shown in detail in Figure A.11. It consists of the two procedures Constructive-Cartesian-Substring and LcCs-Scan.

Procedure LcCs-Scan takes as input two strings, x and y of length m and n, respectively, and computes $\textsc{Lcx}(x_i, y_i)$, for $1 \leq i \leq min(m, n)$, by performing a linear scan of the two strings, from left to right.

Specifically the i-th iteration of the main for loop of line 4 computes the value $r = \textsc{Lcx}(x_i, y_i)$. This is done by building the two Cartesian trees $T_{x_i} = \textsc{Ct}(x_i^r)$ and $T_{y_i} = \textsc{Ct}(y_i^r)$, built on the longest common suffixes of x_i and y_i sharing the same Cartesian tree.

The procedure begins by creating the two empty trees, T_{x_0} and T_{y_0}, corresponding to the empty prefixes of x and y. Then the i-th iteration of the for loop, for $1 \leq i \leq$

$min(n, m)$, extends the Cartesian trees $T_{x_{i-1}}$ and $T_{y_{i-1}}$ by adding the two characters $x[i]$ and $y[i]$, respectively, by means of a call to procedure TREE-EXTEND-RIGHT described in Section A.3.

After the two trees have been extended, procedure LONGEST-COMMON-SUFFIX computes the Cartesian trees T_{x_i} and T_{y_i} for the longest common suffixes of x_i and y_i according to Lemma 7 and Lemma 8.

The pseudo-code of procedure LONGEST-COMMON-SUFFIX is shown in Figure A.10. Assuming that the size of the left child of the rightmost node of a tree can be accessed in constant time, in order for procedure LONGEST-COMMON-SUFFIX to run in constant time, each update on T_x and T_y must also take constant time. Specifically, this is required when, after extending each tree with a new element, the two resulting trees are no longer the same.

Thus suppose that just before calling procedure LONGEST-COMMON-SUFFIX we have $len(rp(T_x)) < len(rp(T_y))$. Then, in order to compute the length of the longest suffixes having the same Cartesian tree, i.e. the new SIZE$[T_y]$, the number of nodes contained in the left subtree of the rightmost node of T_y, $y[m]$, must be known.

An efficient method to achieve this goal is to keep an extra field for each node, telling the number of nodes stored in its left subtree. Such a field, can be easily updated during the incremental construction. Specifically to update T_x and T_y in order to match the Cartesian trees for x_i^r and y_i^r, where r is the total number of nodes contained in the subtree rooted at node $y[m]$, it is enough to set the number of left nodes for node $x[m]$ to r, without performing any additional operation modifying the structure of the tree*.

Procedure CONSTRUCTIVE-CARTESIAN-SUBSTRING simply calls LCCS-SCAN for each of the $m + n$ values $(1, j)$, for $1 \leq j \leq n$, and $(i, 1)$, for $2 \leq i \leq m$, according to equation (A.1).

The correctness of the resulting algorithm is stated in the following Lemma 9

*We notice that we could also get rid of storing the left subtree of each node, since only the total number of nodes it contains is actually relevant for the computation.

```
LONGEST-COMMON-SUFFIX(T_x, T_y)
 1.  if len(rp(T_x)) = len(rp(T_y))
 2.      return (T_x, T_y, Size[T_x])
 3.  m ← Size[T_x]
 4.  if len(rp(T_x)) > len(rp(T_y)) then
 5.      Root[T_x] ← x[m]
 6.      i ← Index[Right[x[m]]]
 7.      Root[T_y] ← y[m]
 8.      Left[y[m]] ← y[i]
 9.  else
10.      Root[T_y] ← y[m]
11.      i ← Index[Right[y[m]]]
12.      Root[T_x] ← x[m]
13.      Left[x[m]] ← x[i]
14.  return (T_x, T_y, Size[T_x])
```

Figure A.10: The pseudo-code of the procedure LONGEST-COMMON-SUFFIX, which updates either the structure of T_x or T_y (lines 4-8 and 10-12) in order to match the longest suffix of the strings x and y having the same Cartesian tree.

```
LCCS-SCAN(x, y, m, n, lcs)
 1.  T_{x_0} ← Empty-Tree()
 2.  T_{y_0} ← Empty-Tree()
 3.  r ← 0
 4.  for i ← 1 to min(m, n) do
 5.      E_{x_i} ← Tree-Extend-Right(T_{x_{i-1}}, x[i])
 6.      E_{y_i} ← Tree-Extend-Right(T_{y_{i-1}}, y[i])
 7.      (T_{x_i}, T_{y_i}, r) ← Longest-Common-Suffix(E_{x_i}, E_{y_i})
 8.      if r > lcs then
 9.          lcs ← r
10.  return lcs

CONSTRUCTIVE-CARTESIAN-SUBSTRING(x, y, m, n)
 1.  lcs ← 1
 2.  for j ← 1 to n do
 3.      lcs ← LcCs-Scan(x, y^{n-j+1}, m, n - j + 1, lcs)
 4.  for i ← 2 to m do
 5.      lcs ← LcCs-Scan(x^{m-i+1}, y, m - i + 1, n, lcs)
 6.  return lcs
```

Figure A.11: The pseudo-code of the algorithm CONSTRUCTIVE-CARTESIAN-SUBSTRING for the computation of the longest Cartesian substrings of two strings x and y, of length m and n respectively, and its auxiliary procedure LCCS-SCAN.

and Lemma 10.

Lemma 9. *At the end of the i-th iteration of procedure* LCCS-SCAN, $T_{x_i} = \text{CT}(x_i^r)$ *and* $T_{y_i} = \text{CT}(y_i^r)$ *hold, where r is the length of the longest common suffix of x_i and*

y_i having the same Cartesian tree.

Proof. We proceed by induction on $i \geq 0$. For $i = 0$, the lemma trivially holds, since $T_{x_1} = \textsc{Ct}(x[1])$ and $T_{y_1} = \textsc{Ct}(y[1])$. Let now $i > 1$. For inductive hypothesis at the end of iteration $i-1$, $T_{x_{i-1}}$ and $T_{y_{i-1}}$ correspond to the longest suffixes of x_{i-1} and y_{i-1} having the same Cartesian tree. Thus, by Lemma 7 and Lemma 8 at the end of the i-th iteration, $T_{x_i} = \textsc{Ct}(x_i^r)$ and $T_{y_i} = \textsc{Ct}(y_i^r)$ hold, where r is the length of the longest suffixes of x_i and y_i having the same Cartesian tree. □

Lemma 10. *Procedure* Constructive-Cartesian-Substring *correctly computes the longest common Cartesian substrings of two strings x and y.*

Proof. Let x and y be two strings of length m and n, respectively, and let $x[i_1..i_2]$ and $y[j_1..j_2]$ be the longest common Cartesian substrings of x and y, where $1 \leq i_1 \leq i_2 \leq m$ and $1 \leq j_1 \leq j_2 \leq n$.

Assume now that $j_1 \geq i_1$ (the case where $i_1 > j_1$ can be easily derived). Thus $j_1 = i_1 + s$ and $j_2 = i_2 + s$, for some $0 \leq s \leq n - i_1$. Let us consider the call to the procedure LcCs-Scan$(x, y[s..n], m, n - s + 1, lcs)$. Then, after $i = i_2 - 1$ iterations of the for loop in line 4, $i = i_2$ and $i + s = i_2 + s = j_2$.

By Lemma 9, at the end of the i-th iteration of the for loop at line 4, $T_{x_i} = \textsc{Ct}(x[i_1..i_2])$ and $T_{y_i} = y[j_1..j_2]$, and, consequently, the length of the longest common Cartesian substrings is computed in line 8. □

Regarding the space and time complexity of our new presented algorithm, we observe that each iteration of the for loop of line 4 in procedure LcCs-Scan performs two incremental insertions, each relative to a different Cartesian tree. As stated in Section A.3, by using the incremental construction, the Cartesian tree for a string x of length m can be computed in $O(m)$ time: thus each call to procedure Tree-Extend-Right takes $O(1)$ amortized time. Since the remaining operations performed in a single iteration of the loop, including the call to procedure Longest-Common-Suffix, require constant time, then each iteration of the for loop of line

4 in procedure LcCs-Scan takes $O(1)$ amortized time, which results in an overall cost of $O(\min(n, m))$ for the entire procedure LcCs-Scan. The additional space required to store T_x and T_y is instead $O(m + n)$.

Finally, by observing that LcCs-Scan is executed $O(m + n)$ times, we conclude that procedure Constructive-Cartesian-Substring correctly computes the longest common Cartesian substrings of two strings x and y of length m and n, respectively, in $O(mn)$-time and $O(m + n)$-space.

A.6.1 A Backward Approach Over the Constructive Solution

In this section, we introduce an efficient heuristic which aims at speeding-up the execution of the constructive based algorithm presented in Section A.6. The idea is to perform, under suitable condition, a jump to the right while scanning the two strings, followed by a backward scan of the strings, in order to avoid reading many characters. As it turns out from our experimental results presented in Section A.7, the new algorithm completely outperforms the original one, and in particular, for large strings, it is also faster than the dynamic programming based algorithm presented in Section A.5.

Suppose that at the beginning of a generic iteration i of the for loop at line 4 of procedure LcCs-Scan, T_{x_i} and T_{y_i} correspond to the longest suffixes x_i^r and y_i^r having the same Cartesian tree, where r is the length of such suffixes. Assume that lcs is the length of the longest common Cartesian substring which has been found up to iteration i.

In order to improve the current value for the length of the longest common Cartesian substrings, at least $t = lcs - r + 1$ additional elements should be scanned without falling back into a mismatch (this last case is handled by procedure Longest-Common-Suffix by applying Lemma 8).

The new algorithm is based on the alternation of two separate steps. The algorithm begins with the following Step 1:

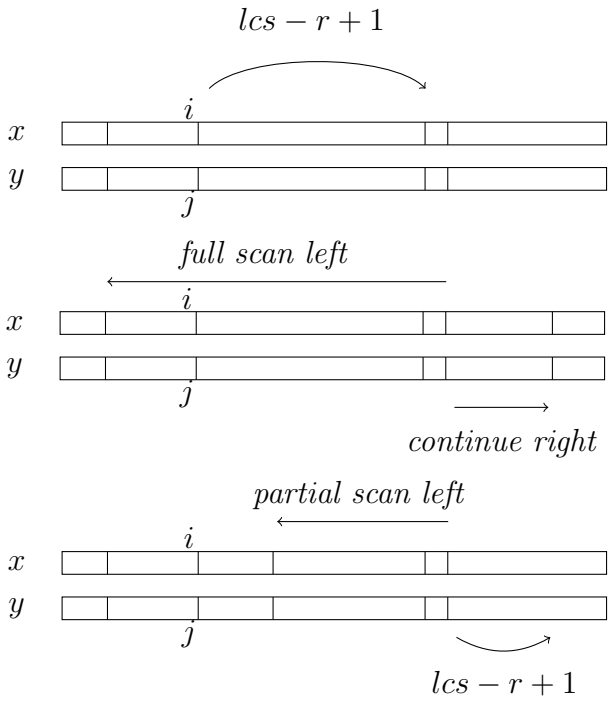

Figure A.12: The three steps which may occur during the execution of algorithm Jumping-LcCs-Scan. **Step 1:** an incremental construction is performed from left to right until a mismatch is detected among T_{x_i} and T_{y_i}. Then a jump to the right of $t = lcs + r - 1$ positions is performed. **Step 2.1:** a full scan is performed proceeding from right to left without any mismatch and the value of lcs is incremented by one. We then continue scanning right with Step 1. **Step 2.2:** We fall in a mismatch before reaching the length lcs. Then we jump right again and continue with Step 2.

- **Step 1**: incremental construction is performed from left to right until the hypothesis of Lemma 8 occurs, i.e. a mismatch is detected among T_{x_i} and T_{y_i} (see Figure A.12, on the top).

At the end of Step 1, after T_{x_i} and T_{y_i} are made equal again by a call to Longest-Common-Suffix, we jump right by $t = lcs - r + 1$ elements. Starting from these elements, Step 2 begins, which, starting from scratch, performs an incremental construction of the trees $T_{x_{i+t}}$ and $T_{y_{i+t}}$ proceeding from right to left while the two Cartesian trees match. Two further steps may arise at this point.

- **Step 2.1**: incremental construction proceeds from right to left without any

mismatch occurring before elements $x[i + t - lcs - 1]$ and $y[i + t - lcs - 1]$ are reached. In this case the value of lcs has been successfully incremented by one. We then continue with Case 1 again (see Figure A.12, in the middle).

- **Step 2.2**: while proceeding from right to left a mismatch occurs between $T_{x_{i+t}}$ and $T_{y_{i+t}}$ before elements $x[i + t - lcs - 1]$ and $y[i + t - lcs - 1]$ are reached. This means that we are not able to increment the value of lcs by continue scanning the strings from right to left. Thus we return at the end of Case 1, when a new jump is performed, and continue with Case 2 again (see Figure A.12, on the bottom).

Figure A.13 shows the pseudo-code of procedure JUMPING-LCCS-SCAN described above. The fact that jumps are performed while scanning elements from left to right results in a drastic speed improvement of the original constructive algorithm. The only drawback of this approach is that, when step 2.1 occurs, the first l elements are processed twice. However, in the next section, we will give experimental evidence that, in practice, step 2.1 doesn't have a significant performance impact.

A.7 Experimental Results

In this section we present an extensive experimental evaluation in order to compare, in terms of running times, our alternative LCCS algorithms against the standard Suffix Tree based solution.

The four algorithms presented above have been implemented in C programming language[†]. The experiments were performed on a computer running Linux Ubuntu 20.04.1 with an Intel Core i5 3.40 GHz processor and 8GB RAM.

All algorithms are assumed to compute the length of the longest common Cartesian substring of two input strings, x and y, of length m and n, respectively. We assume without loss of generality that $m < n$.

Specifically, our experimental environment is formed by the following algorithms:

[†]Source code is available at the following link `https://github.com/ostafen/lccs`

```
JUMPING-LCCS-SCAN(x, y, m, n, lcs)
 1.  T_{x_0} ← Empty-Tree()
 2.  T_{y_0} ← Empty-Tree()
 3.  r ← 0
 4.  r' ← −1
 5.  i ← lcs + 1
 6.  while i ≤ min(m, n) do
 7.      while r ≤ lcs and r > r' do
 8.          r' ← r
 9.          E_{x_i} ← Tree-Extend-Right(T_{x_i}, x[i − r])
10.          E_{y_i} ← Tree-Extend-Right(T_{y_i}, y[i − r])
11.          (T_{x_i}, T_{y_i}, r) ← Longest-Common-Suffix(E_{x_i}, E_{y_i})
12.      if r > lcs then
13.          while i < min(m, n) and r > r' do
14.              i ← i + 1
15.              lcs ← r
16.              r' ← r
17.              E_{x_i} ← Tree-Extend-Right(T_{x_{i−1}}, x[i])
18.              E_{y_i} ← Tree-Extend-Right(T_{y_{i−1}}, y[i])
19.              (T_{x_i}, T_{y_i}, r) ← Longest-Common-Suffix(E_{x_i}, E_{y_i})
20.      i ← i + lcs − r + 1
21.  return lcs
```

Figure A.13: The pseudo-code of procedure JUMPING-LCCS-SCAN. At each step, the algorithm tries to improve the length of the current longest Cartesian substring, by alternating both leftmost and rightmost construction of the Cartesian trees. Under some suitable conditions, when a mismatch of the two trees occurs, forwards jumps are performed.

- ST: the generalized suffix-tree based algorithm presented in Section A.4;
- DP: the algorithm of Section A.5, based on parent-distance representation and dynamic programming;
- IC and BIC: the left to right incremental construction algorithm and its backward variant, respectively, presented in Section A.6.

A.7.1 Implementation Details

In our solution based on generalized suffix-trees (ST) the implementation with missing suffix links from [64] has been used without back propagation and with a simple hashing function. Although our implementation is not linear in the worst case we argue that for short strings it is faster than the randomized linear method which is quite intricate and could lead to an increase in running times for short strings.

The method for finding the longest Cartesian substring between two strings, based on Suffix Trees, is then the following: the parent distance representations PD_x and PD_y are computed. Then we construct $PD_{yx} = PD_y\$_1 PD_x\$_2$ where $\$_1$ and $\$_2$ are terminators that do not occur in PD_x and PD_y. Then the suffix tree of PD_{yx} is build for the part corresponding of the longest string y. For the remaining part the suffix tree is scanned to determine the fork where to insert the tail of the current suffix. The string depth of this fork is used to compute the length of the longest common Cartesian substring. The tail is not inserted since it is not necessary for our purposes and will only lead to a loss of time.

For our dynamic programming solution (Dp), long diagonals (with $\min\{m, n\}$ elements) are processed first. Shortest diagonals corresponding to prefixes and suffixes of y are processed in a second time. During the computation of a diagonal, when the length of the remaining part of the diagonal is too short and will not possibly contribute to a longest common Cartesian substring, we processed to the next diagonals.

Regarding our methods based on incremental construction (Ic and Bic), we choose to implement a Cartesian tree as an array of fixed size equal to $\max(m, n)$. In order to avoid reallocation, the array corresponding to a Cartesian tree is managed as a circular buffer, so that old array locations are gradually overwritten. Moreover, procedure Longest-Common-Suffix have been efficiently implemented according to the implementation notes discussed in Section A.6.

A.7.2 Results on Random and Real Data

We conducted experiments both on random and real data. Concerning random data, we built random strings of integers, each drawn with uniform distribution from the set $\{0, 1, ..., \sigma - 1\}$, with $\sigma = 10, 10^2, 10^3, 10^4$. Real data sequences have been randomly drawn from the well-known *Seoul Temperatures* dataset provided by the KMA National Climate Data Center (`https://data.kma.go.kr/resources/html/`

σ/k	1	2	3	4	5	6	7	8	9	10
10	0.34	0.17	0.07	0.03	0.01	$0.42 \cdot 10^{-1}$	$0.18 \cdot 10^{-2}$	$0.7 \cdot 10^{-3}$	$0.24 \cdot 10^{-3}$	$0.84 \cdot 10^{-4}$
10^2	0.35	0.17	0.07	0.02	0.01	$0.39 \cdot 10^{-2}$	$0.13 \cdot 10^{-2}$	$0.43 \cdot 10^{-3}$	$0.16 \cdot 10^{-3}$	$0.72 \cdot 10^{-4}$
10^3	0.34	0.17	0.07	0.03	0.01	$0.42 \cdot 10^{-2}$	$0.15 \cdot 10^{-2}$	$0.59 \cdot 10^{-3}$	$0.26 \cdot 10^{-3}$	$0.11 \cdot 10^{-3}$
10^4	0.35	0.17	0.07	0.02	0.01	$0.42 \cdot 10^{-2}$	$0.14 \cdot 10^{-2}$	$0.56 \cdot 10^{-3}$	$0.21 \cdot 10^{-4}$	$0.80 \cdot 10^{-4}$

Table A.1: Relative frequency of common Cartesian substring of length k among the number of k-mers of two random generated strings of integers x and y. Each row refers to a specific value of σ.

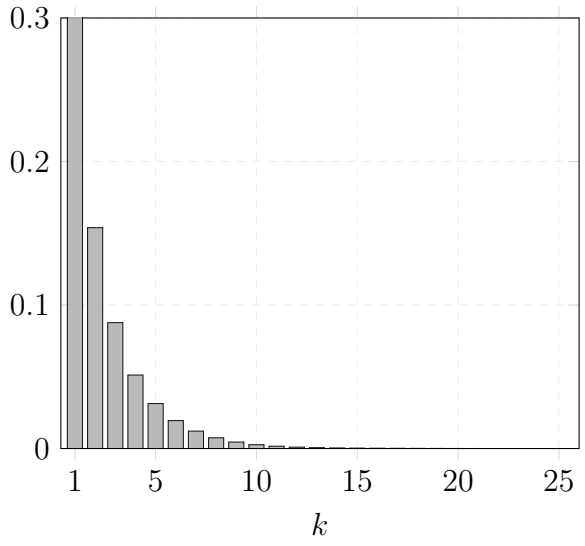

Figure A.14: Histogram showing the relative frequency (on the y-axis) of common Cartesian substring of length k (on the x-axis) among the number of k-mers of two strings of integers x and y, randomly extracted from the Seoul Temperature dataset.

		$n = 16$				$n = 32$				$n = 64$			
	Algo/m	2	4	8	16	4	8	16	32	8	16	32	64
$\sigma = 10$	St	6.97	4.27	5.26	6.29	7.73	8.27	9.98	**12.1**	14.5	16.3	19.1	**23.7**
	Dp	1.34	**0.83**	**1.09**	**1.93**	**0.96**	**1.25**	**2.45**	**6.29**	**1.40**	**2.56**	**7.16**	**22.9**
	Ic	**1.02**	**0.89**	**2.08**	4.87	**0.94**	**2.11**	6.23	19.4	**2.11**	5.97	21.3	77.8
	Bic	**1.09**	1.00	2.19	**4.47**	1.08	2.26	**5.43**	13.2	2.28	**5.21**	**13.5**	39.1
$\sigma = 10^2$	St	3.90	4.16	5.02	6.19	7.65	8.60	10.0	**12.0**	15.51	17.1	19.8	**23.6**
	Dp	0.76	**0.83**	**1.08**	**1.93**	**1.02**	**1.31**	**2.55**	**6.64**	**1.52**	**2.82**	**7.97**	**24.7**
	Ic	**0.58**	**0.90**	**2.01**	4.84	**1.00**	**2.24**	6.25	19.4	**2.33**	6.43	22.5	78.0
	Bic	**0.62**	0.99	2.15	**4.37**	1.13	2.41	**5.53**	13.2	2.45	**5.65**	**14.5**	39.4
$\sigma = 10^3$	St	3.88	4.14	4.95	6.22	7.34	8.23	9.81	**11.5**	14.66	17.6	20.8	**25.3**
	Dp	0.73	**0.87**	**1.13**	**1.98**	**1.00**	**1.27**	**2.51**	**6.40**	**1.41**	**2.85**	**8.36**	**26.3**
	Ic	**0.57**	**0.87**	**2.03**	4.82	**1.01**	**2.12**	6.01	18.9	**2.12**	6.47	23.4	81.7
	Bic	**0.62**	**0.98**	2.13	**4.37**	1.09	2.31	**5.24**	12.9	2.31	**5.66**	**15.0**	41.2
$\sigma = 10^4$	St	3.93	4.15	4.97	6.54	7.20	8.09	9.49	**11.7**	15.3	17.3	20.0	**24.7**
	Dp	0.72	**0.82**	**1.09**	**2.00**	**0.91**	**1.24**	**2.41**	**6.41**	**1.55**	**2.78**	**8.12**	**26.0**
	Ic	**0.58**	**0.92**	**2.03**	5.01	**0.92**	**2.05**	5.93	18.8	**2.32**	6.40	22.6	81.2
	Bic	**0.60**	0.97	2.16	**4.54**	1.07	2.18	**5.14**	13.0	2.43	**5.71**	**14.5**	40.9

Table A.2: Running times of LcCs algorithms on random data and for $n = 16, 32$ and 64. Times are expressed in milliseconds.

`en/aowdp.html`), collecting temperature data from 1907 to 2019. In the dataset, the percentage of duplicated strings is around 5%, for strings of length $k \leq 3$, $0.07 - 0.014\%$ for $4 \leq k \leq 6$, and rapidly tends to 0 for $k \geq 7$.

In both cases, we set n to increasing powers of two in the range $[2^4..2^{12}]$ and, for each value of n, we use 4 values of m: $n/8$, $n/4$, $n/2$ and n, respectively.

For a better understanding of the data-sets we report in Table A.1 and Figure A.14, respectively, the histograms of the relative frequency (on the y-axis) of each length k (on the x-axis) for all common Cartesian substrings which are present inside two strings of integers x and y, for the Random dataset and the Seoul Temperature dataset.

Table A.2, Table A.3 and Table A.4 show the running times of the four algorithms in tabular form, in the case of random data. In the two tables the range of possible values for n have been divided into three sub-ranges, $[2^4..2^6]$, $[2^7..2^9]$ and $[2^{10}..2^{12}]$, respectively. Each table is organized into multiple sub-tables, one for each value of n, with columns representing increasing values of m, and this structure is replicated for each of the possible values of σ.

		$n = 128$				$n = 256$				$n = 512$			
	Algo/m	16	32	64	128	32	64	128	256	64	128	256	512
$\sigma = 10$	St	2.95	3.28	**4.00**	**4.72**	5.95	7.19	**7.93**	**9.90**	**4.47**	4.94	**5.67**	**6.89**
	Dp	**0.31**	**0.79**	**2.77**	**9.38**	**0.87**	**2.88**	**10.83**	**40.10**	**3.30**	**11.18**	**43.92**	164.9
	Ic	0.63	2.23	8.91	33.13	2.27	9.06	36.22	141	8.84	34.51	141.0	557.8
	Bic	**0.57**	**1.42**	4.17	12.44	**1.48**	**4.28**	13.01	42.74	4.49	13.29	43.97	**49.7**
$\sigma = 10^2$	St	2.87	3.16	**3.77**	**4.67**	6.10	6.91	**8.13**	**10.42**	4.50	**4.90**	**5.79**	**7.25**
	Dp	**0.30**	**0.80**	**2.79**	**9.83**	**0.94**	**3.06**	**11.54**	**42.51**	**3.37**	**11.80**	46.33	174.0
	Ic	0.60	2.15	8.43	32.32	2.30	8.98	36.43	142	8.49	34.68	141.5	566.1
	Bic	**0.54**	**1.39**	4.02	12.31	**1.52**	**4.30**	13.32	43.93	**4.38**	13.62	**44.98**	**155.8**
$\sigma = 10^3$	St	2.84	3.24	**3.85**	**5.00**	6.32	7.25	**8.04**	**10.13**	4.49	**4.97**	**5.71**	**6.84**
	Dp	**0.31**	**0.83**	**2.90**	**10.42**	**0.95**	**3.13**	**11.41**	**42.84**	**3.46**	**12.23**	47.73	175.4
	Ic	0.61	2.21	8.71	34.13	2.34	9.07	35.92	142	8.70	35.44	144.3	564.4
	Bic	**0.55**	**1.43**	4.16	13.14	**15.5**	**4.36**	13.10	43.92	**4.47**	14.06	**46.05**	**154.3**
$\sigma = 10^4$	St	3.01	3.35	**3.84**	**4.98**	6.19	7.00	**8.04**	**10.46**	**4.50**	**4.96**	**5.58**	**6.90**
	Dp	**0.32**	**0.85**	**2.89**	**10.45**	**0.95**	**3.08**	**11.70**	**42.62**	**3.52**	**11.93**	46.96	174.2
	Ic	0.64	2.27	8.69	34.37	2.35	9.01	36.53	142	8.77	34.45	142.5	562.3
	Bic	**0.57**	**1.47**	4.14	13.18	**1.56**	**4.34**	13.32	43.61	4.52	13.56	**45.66**	**154.9**

Table A.3: Running times of LCCS algorithms on random data and for $n = 128, 256$ and 512. Times are expressed in centiseconds.

		$n = 1024$				$n = 2048$				$n = 4096$			
	Algo/m	128	256	512	1024	256	512	1024	2048	512	1024	2048	4096
$\sigma = 10$	St	**0.09**	**0.10**	**0.11**	**0.14**	**0.20**	**0.22**	**0.25**	**0.30**	**0.41**	**0.45**	**0.48**	**0.61**
	Dp	**0.12**	0.45	1.80	6.75	0.45	1.78	7.29	27.13	1.78	7.08	28.35	108.66
	Ic	0.36	1.48	6.00	23.65	1.46	5.91	24.21	94.66	5.83	23.43	94.10	378.52
	Bic	0.13	**0.43**	**1.49**	**5.14**	**0.43**	**1.46**	**5.23**	**18.36**	**1.45**	**5.08**	**18.24**	**66.21**
$\sigma = 10^2$	St	**0.09**	**0.10**	**0.12**	**0.15**	**0.19**	**0.21**	**0.24**	**0.31**	**0.39**	**0.43**	**0.49**	**0.62**
	Dp	**0.12**	0.46	1.87	6.95	0.48	1.86	7.34	28.00	1.83	7.37	29.27	109.98
	Ic	0.34	1.46	5.95	23.42	1.46	5.89	23.37	93.89	5.75	23.39	92.86	368.09
	Bic	0.13	**0.43**	**1.50**	**5.23**	**0.44**	**1.49**	**5.17**	**18.65**	**1.46**	**5.18**	**18.31**	**66.12**
$\sigma = 10^3$	St	**0.09**	**0.10**	**0.12**	**0.15**	**0.20**	**0.21**	**0.25**	**0.30**	**0.39**	**0.43**	**0.49**	**0.62**
	Dp	**0.13**	0.47	1.87	6.98	0.48	1.88	7.42	27.99	1.82	7.24	29.05	109.73
	Ic	0.37	1.47	5.93	23.39	1.45	5.91	23.48	93.37	5.69	22.87	91.88	365.72
	Bic	0.14	**0.44**	**1.50**	**5.20**	**0.44**	**1.50**	**5.18**	**18.52**	**1.44**	**5.06**	**18.24**	**65.65**
$\sigma = 10^4$	St	**0.09**	**0.11**	**0.13**	**0.15**	**0.19**	**0.21**	**0.25**	**0.31**	**0.39**	**0.43**	**0.49**	**0.62**
	Dp	**0.12**	0.46	1.83	6.82	0.46	1.83	7.36	28.04	1.81	7.21	28.93	109.79
	Ic	0.36	1.44	5.79	22.85	1.41	5.76	23.29	93.47	5.65	22.76	91.43	365.57
	Bic	0.13	**0.43**	**1.47**	**5.14**	**0.43**	**1.46**	**5.17**	**18.54**	**1.43**	**5.04**	**18.08**	**65.68**

Table A.4: Running times of LCCS algorithms on random data and for $n = 1024, 2048$ and 4096. Times are expressed in seconds.

	$n = 16$				$n = 32$				$n = 64$			
Algo/m	2	4	8	16	4	8	16	32	8	16	32	64
St	3.27	3.45	3.94	4.67	5.82	6.51	7.34	**8.76**	11.29	12.16	**13.74**	**16.92**
Dp	0.73	**0.79**	**1.10**	**1.58**	**0.94**	**1.41**	**2.49**	**4.51**	**1.75**	**3.90**	**8.75**	**17.61**
Ic	**0.63**	**1.12**	**2.13**	**3.78**	**1.31**	**3.50**	6.84	13.40	**5.23**	12.59	26.43	55.19
Bic	**0.72**	1.29	2.35	**3.78**	1.59	3.71	**6.21**	9.95	5.49	**10.41**	16.80	28.22

	$n = 128$				$n = 256$				$n = 512$			
Algo/m	16	32	64	128	32	64	128	256	32	64	128	256
St	21.87	**23.65**	**27.28**	**33.06**	**45.01**	**49.81**	**54.68**	**67.73**	**88.51**	**95.15**	**108.43**	**129.72**
Dp	**6.20**	**15.86**	**35.90**	**72.56**	**29.31**	**74.18**	**154.42**	302.56	**132.01**	294.66	608.61	1184.35
Ic	22.91	51.58	111.95	238.75	102.76	239.65	495.27	1055.59	454.13	1000.18	2118.29	4390.09
Bic	**17.93**	29.49	49.25	85.94	54.06	94.83	155.92	**274.41**	162.75	**277.97**	**497.55**	**899.94**

	$n = 1024$				$n = 2048$				$n = 4096$			
Algo/m	16	32	64	128	32	64	128	256	32	64	128	256
St	**0.18**	**0.19**	**0.23**	**0.29**	**0.39**	**0.42**	**0.46**	**0.55**	**0.77**	**0.85**	**0.92**	**1.15**
Dp	0.60	1.26	2.57	4.96	2.57	5.26	10.38	19.83	10.29	21.16	41.40	79.25
Ic	2.04	4.31	9.09	18.50	8.87	18.31	37.04	74.13	35.65	73.95	148.47	298.25
Bic	**0.52**	**0.92**	**1.73**	**3.28**	**1.75**	**3.28**	**6.19**	**11.92**	**6.15**	**11.99**	**23.16**	**43.65**

Table A.5: Running times of LcCs algorithms on Seoul Temperature data for $n \in [2^4..2^{12}]$. Times are expressed in milliseconds for $n \in [2^4..2^9]$ and in seconds for $n \in [2^{10}..2^{12}]$.

Experiments on real data are shown in Table A.5.

In all cases, the two best results have been boldfaced (in addition, the best result has been underlined).

Each algorithm shows a similar behaviour both on random and real data. From the results, it clearly turns out that the quadratic solutions are faster than St in the case of sufficiently short string ($n \in [2^4..2^6]$). In particular, Dp achieves the best performance up to $n = 2^7$, while Ic and Bic compete for the second best result. As n increases or when m gets very close to n, the difference in speed between the quadratic algorithms and the linear solution is gradually lowered, with St becoming faster starting from $n = 256$, and Bic achieving the second best result (starting from $n = 512$).

It is also worth to notice that, as the values of n and m increase, specifically for $n \geq 2^9$, the running times of the Bic algorithm undergo a milder surge with respect to the remaining quadratic solutions, thanks to its jump-based heuristic.

Table A.6 shows the space consumption of the proposed solutions for strings of increasing length, where $m = n$. With a space requirement that exceeds of about

Algo/m	2	4	8	16	32	64	128	256	512	1024	2048	4096
St	1.20	1.97	3.50	6.56	12.69	24.94	49.44	98.44	196.44	392.44	784.44	1568.44
Dp	0.05	0.09	0.19	0.38	0.75	1.50	3.00	6.00	12.00	24.00	48.00	96.00
Ic	0.05	0.09	0.19	0.38	0.75	1.50	3.00	6.00	12.00	24.00	48.00	96.00
Bic	0.06	0.12	0.25	0.50	1.00	2.00	4.00	8.00	16.00	32.00	64.00	128.00

Table A.6: Space consumption expressed in kilobytes of the presented solutions for strings of equal size (i.e. $m = n$).

12-20 times the remaining solutions, the St algorithm is certainly the most space demanding. All the other algorithms show a similar behaviour in terms of space, with Dp and Ic being the cheapest alternatives. We also notice that Dp and Ic take exactly the same space, while Bic is more space demanding than Ic. This is due to the fact that the Bic algorithm performs the incremental construction of the Cartesian trees both from left to right and from right to left, and thus additional information with respect to Ic are needed.

To conclude, our experimental analysis suggests that no algorithm is suitable for every possible scenario; however, two algorithms stand out naturally from our experiments, depending on the length of the strings considered. In particular, Dp is clearly the best choice for strings of small size ($m, n \leq 64$), even if all the quadratic algorithms show good performances in such case, while St is superior in every other case.

Finally, we also signal the Bic algorithm as a good alternative to the St algorithm for large strings, since its running time tends to degrade less noticeably with respect to other quadratic solutions as the length of the input increases, while maintaining the space requirements low.

A.7.3 Results on Real Data

In the experimental results shown in this section we use real data from the first wave COVID-19 pandemic occurred starting from March 2020.[‡] Specifically we extracted the numbers of cases and numbers of deaths for the 15 most infected countries at

[‡]We downloaded data from `https://opendata.ecdc.europa.eu/covid19/casedistribution/csv` on 27th April 2020

♯	Country	Cases	Deaths
1	China	102	98
2	France	62	53
3	Germany	62	49
4	Iran	68	68
5	Italy	66	65
6	Korea	72	67
7	Spain	63	54
8	Turkey	45	40
9	UK	105	98
10	USA	67	58
11	Russia	47	32
12	Brazil	54	41
13	Canada	63	42
14	Belgium	57	47
15	Netherlands	60	50

Table A.7: Country number of cases and deaths from the first wave COVID-19 pandemic occurred from 1st March 2020 to 27th April 2020

that time. Data were given in reverse chronological order. We trimmed the data for the tailing runs of 0s and 1s at the end.

Table A.7 reports the number of daily cases and deaths for fifteen different countries. For each pair of countries, we computed the length of the longest common Cartesian substrings.

Figure A.15 (on top) shows the results for the number of cases. These results have been computed in:

- 3.843 μs for the ST algorithm
- 3.678 μs for the DP algorithm
- 11.996 μs for the IC algorithm
- 5.861 μs for the BIC algorithm

Similarly Figure A.15 (on bottom) shows the results for the number of deaths. These results have been computed in:

- 3.629 μs for the ST algorithm

Number of COVID-19 cases

	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	11	8	11	11	10	10	13	9	13	14	9	9	10	11
2		9	12	11	8	9	16	11	11	10	9	11	7	12
3			8	11	8	11	9	11	9	8	12	8	8	11
4				11	10	9	12	8	14	12	8	9	8	10
5					9	10	10	13	11	11	10	12	9	10
6						9	7	9	8	8	8	9	7	8
7							9	10	9	9	9	9	11	11
8								9	12	15	8	9	7	12
9									10	10	9	14	12	11
10										14	9	10	9	10
11											9	9	8	10
12												11	10	8
13													9	8
14														9

Number of COVID-19 deaths

	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	9	8	14	7	10	10	10	10	8	10	8	9	13	10
2		10	10	9	7	7	10	7	9	9	9	9	12	14
3			9	9	8	11	9	11	10	8	7	8	10	7
4				8	9	10	13	11	9	9	8	9	10	8
5					8	8	7	10	14	10	11	10	8	11
6						8	7	9	7	9	8	7	8	8
7							10	8	9	9	10	9	9	8
8								7	9	10	7	7	11	8
9									8	9	10	10	9	9
10										8	11	8	11	9
11											9	8	10	9
12												7	7	9
13													9	10
14														8

Figure A.15: Length of the longest common Cartesian substring between countries for the number of cases (on top) and the number of deaths (on bottom) of the COVID-19.

- 2.892 μs for the Dp algorithm
- 9.507 μs for the Ic algorithm
- 5.073 μs for the Bic algorithm

Again, these results show that for such short strings our Dp alternative solution is faster than the suffix tree based method.

In our real data experiments, Dp is faster than the suffix tree solution in 38 cases out of 50 cases; when it is faster, Dp is 3.81 times faster on average (up to 8.43 times

for the maximum), and when it is slower, it is 1.3 times slower on average than the suffix tree solution (up to 1.82 for the maximum).

A.7.4 Conclusions

In this chapter we presented a classical suffix tree based solution for computing the longest Cartesian substrings between two strings. This solution is based on the parent-distance representations of the two strings and runs in randomized linear time and linear extra-space in addition to the two strings and their parent-distance representation. Then we proposed two alternative solutions, based on dynamic programming and incremental Cartesian tree construction, that run in quadratic time in the worst case and in multiplicative constant extra-space in addition to the two strings and their parent-distance representation. Moreover, we introduced a jump-based heuristic which improves the performance of our constructive approach, especially for large strings. We presented experiments, both on random and real data, showing the effectiveness of our dynamic programming and constructive solutions for strings of short and large size, respectively. Further works would include the search for the longest approximate common Cartesian substrings between two strings but the notion of approximation in this context has to be defined.

Bibliography

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[2] S. Faro and T. Lecroq, "The exact online string matching problem: A review of the most recent results," *ACM Comput. Surv.*, vol. 45, no. 2, pp. 13:1–13:42, 2013.

[3] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977.

[4] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, 1977.

[5] M. Crochemore and W. Rytter, *Text Algorithms*. Oxford University Press, 1994.

[6] A. C. Yao, "The complexity of pattern matching for a random string," tech. rep., Stanford, CA, USA, 1977.

[7] H. Peltola and J. Tarhio, "Alternative algorithms for bit-parallel string matching," in *String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003*, vol. 2857 of *LNCS*, pp. 80–94, Springer, 2003.

[8] B. Durian, H. Peltola, L. Salmela, and J. Tarhio, "Bit-parallel search algorithms for long patterns," in *Experimental Algorithms, 9th International Symposium, SEA 2010*, vol. 6049 of *LNCS*, pp. 129–140, Springer, 2010.

[9] D. Cantone, S. Faro, and E. Giaquinta, "A compact representation of nondeterministic (suffix) automata for the bit-parallel approach," *Inf. Comput.*, vol. 213, pp. 3–12, 2012.

[10] S. Faro and T. Lecroq, "A fast suffix automata based algorithm for exact online string matching," in *Implementation and Application of Automata - 17th International Conference, CIAA 2012*, vol. 7381 of *LNCS*, pp. 149–158, Springer, 2012.

[11] R. A. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Commun. ACM*, vol. 35, no. 10, pp. 74–82, 1992.

[12] G. Navarro and M. Raffinot, "A bit-parallel approach to suffix automata: Fast extended string matching," in *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Proceedings* (M. Farach-Colton, ed.), vol. 1448 of *LNCS*, pp. 14–33, Springer, 1998.

[13] S. Faro, "A very fast string matching algorithm based on condensed alphabets," in *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016, Proceedings* (R. Dondi, G. Fertin, and G. Mauri, eds.), vol. 9778 of *Lecture Notes in Computer Science*, pp. 65–76, Springer, 2016.

[14] S. Faro and S. Scafiti, "A weak approach to suffix automata simulation for exact and approximate string matching," *Theor. Comput. Sci.*, vol. 933, pp. 88–103, 2022.

[15] D. Cantone, S. Faro, and A. Pavone, "Speeding up string matching by weak factor recognition," in *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017* (J. Holub and J. Zdárek, eds.), pp. 42–50, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017.

[16] C. Allauzen, M. Crochemore, and M. Raffinot, "Factor oracle: A new structure for pattern matching," in *SOFSEM '99, Theory and Practice of Informatics, 26th Conference on Current Trends in Theory and Practice of Infor-*

matics, Milovy, Czech Republic, November 27 - December 4, 1999, Proceedings (J. Pavelka, G. Tel, and M. Bartosek, eds.), vol. 1725 of *Lecture Notes in Computer Science*, pp. 295–310, Springer, 1999.

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[18] S. Faro and S. Scafiti, "Pruned BNDM: extending the bit-parallel suffix automata to large strings," in *Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021* (C. S. Coen and I. Salvo, eds.), vol. 3072 of *CEUR Workshop Proceedings*, pp. 328–340, CEUR-WS.org, 2021.

[19] S. Faro and S. Scafiti, "Compact suffix automata representations for searching long patterns," *Theor. Comput. Sci.*, vol. 940, no. Part, pp. 254–268, 2023.

[20] S. Faro and S. Scafiti, "The range automaton: An efficient approach to text-searching," in *Combinatorics on Words - 13th International Conference, WORDS 2021, Rouen, France, September 13-17, 2021, Proceedings* (T. Lecroq and S. Puzynina, eds.), vol. 12847 of *Lecture Notes in Computer Science*, pp. 91–103, Springer, 2021.

[21] S. Faro and S. Scafiti, "Efficient string matching based on a two-step simulation of the suffix automaton," in *Implementation and Application of Automata - 25th International Conference, CIAA 2021, Virtual Event, July 19-22, 2021, Proceedings* (S. Maneth, ed.), vol. 12803 of *Lecture Notes in Computer Science*, pp. 165–177, Springer, 2021.

[22] N. Uratani and M. Takeda, "A fast string-searching algorithm for multiple patterns," *Inf. Process. Manag.*, vol. 29, no. 6, pp. 775–792, 1993.

[23] M. J. Fischer and M. S. Paterson, "String matching and other products," in *Complexity of Computation (SYAM-AMS Proceedings 7)* (R. Karp, ed.), vol. 7, (USA), pp. 113,125, Massachusetts Institute of Technology, 1974.

[24] R. Y. Pinter, "Efficient string matching with don't-care patterns," in *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.), (Berlin, Heidelberg), pp. 11–29, Springer Berlin Heidelberg, 1985.

[25] D. Cantone, S. Faro, and E. Giaquinta, "A compact representation of nondeterministic (suffix) automata for the bit-parallel approach," in *Combinatorial Pattern Matching, CPM 2010, Proceedings*, vol. 6129 of *LNCS*, pp. 288–298, Springer, 2010.

[26] S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio, "The string matching algorithms research tool," in *Proceedings of the Prague Stringology Conference, 2016* (J. Holub and J. Zdárek, eds.), pp. 99–111, 2016.

[27] S. Muthukrishnan, "New results and open problems related to non-standard stringology," in *Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95, Espoo, Finland, July 5-7, 1995, Proceedings* (Z. Galil and E. Ukkonen, eds.), vol. 937 of *Lecture Notes in Computer Science*, pp. 298–317, Springer, 1995.

[28] D. Cantone and S. Faro, "Pattern matching with swaps for short patterns in linear time," in *SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 24-30, 2009. Proceedings* (M. Nielsen, A. Kucera, P. B. Miltersen, C. Palamidessi, P. Tuma, and F. D. Valencia, eds.), vol. 5404 of *Lecture Notes in Computer Science*, pp. 255–266, Springer, 2009.

[29] M. Campanelli, D. Cantone, and S. Faro, "A new algorithm for efficient pattern matching with swaps," in *Combinatorial Algorithms, 20th International*

Workshop, IWOCA 2009, Hradec nad Moravicí, Czech Republic, June 28-July 2, 2009, Revised Selected Papers (J. Fiala, J. Kratochvíl, and M. Miller, eds.), vol. 5874 of *Lecture Notes in Computer Science*, pp. 230–241, Springer, 2009.

[30] S. Faro, "Swap matching in strings by simulating reactive automata," in *Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013* (J. Holub and J. Zdárek, eds.), pp. 7–20, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2013.

[31] S. Faro and A. Pavone, "An efficient skip-search approach to swap matching," *Comput. J.*, vol. 61, no. 9, pp. 1351–1360, 2018.

[32] M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen, "A linear time algorithm for consecutive permutation pattern matching," *Inf. Process. Lett.*, vol. 113, no. 12, pp. 430–433, 2013.

[33] J. Kim, P. Eades, R. Fleischer, S. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama, "Order-preserving matching," *Theor. Comput. Sci.*, vol. 525, pp. 68–79, 2014.

[34] D. Cantone, S. Faro, and M. O. Külekci, "An efficient skip-search approach to the order-preserving pattern matching problem," in *Proceedings of the Prague Stringology Conference 2015, Prague, Czech Republic, August 24-26, 2015* (J. Holub and J. Zdárek, eds.), pp. 22–35, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2015.

[35] S. Faro and M. O. Külekci, "Efficient algorithms for the order preserving pattern matching problem," in *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016,*

Proceedings (R. Dondi, G. Fertin, and G. Mauri, eds.), vol. 9778 of *Lecture Notes in Computer Science*, pp. 185–196, Springer, 2016.

[36] S. Faro and M. O. Külekci, "Efficient algorithms for the order preserving pattern matching problem," in *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016, Proceedings* (R. Dondi, G. Fertin, and G. Mauri, eds.), vol. 9778 of *Lecture Notes in Computer Science*, pp. 185–196, Springer, 2016.

[37] D. Cantone, S. Faro, and M. O. Külekci, "The order-preserving pattern matching problem in practice," *Discret. Appl. Math.*, vol. 274, pp. 11–25, 2020.

[38] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka, "Approximating LZ77 via small-space multiple-pattern matching," *CoRR*, vol. abs/1504.06647, 2015.

[39] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[40] S. Dori and G. M. Landau, "Construction of aho corasick automaton in linear time for integer alphabets," *Inf. Process. Lett.*, vol. 98, no. 2, pp. 66–72, 2006.

[41] G. Navarro and K. Fredriksson, "Average complexity of exact and approximate multiple string matching," *Theor. Comput. Sci.*, vol. 321, no. 2-3, pp. 283–290, 2004.

[42] G. Navarro and M. Raffinot, *Flexible pattern matching in strings - practical online search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[43] S. Wu and U. Manber, "Fast text searching allowing errors," *Commun. ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[44] G. Navarro and M. Raffinot, "Fast and flexible string matching by combining bit-parallelism and suffix automata," *ACM J. Exp. Algorithmics*, vol. 5, p. 4, 2000.

[45] E. Rivals, L. Salmela, P. Kiiskinen, P. Kalsi, and J. Tarhio, "mpscan: Fast localisation of multiple reads in genomes," in *Algorithms in Bioinformatics, 9th International Workshop, WABI 2009, Philadelphia, PA, USA, September 12-13, 2009. Proceedings* (S. Salzberg and T. J. Warnow, eds.), vol. 5724 of *Lecture Notes in Computer Science*, pp. 246–260, Springer, 2009.

[46] S. Faro and M. O. Külekci, "Fast multiple string matching using streaming SIMD extensions technology," in *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings* (L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, eds.), vol. 7608 of *Lecture Notes in Computer Science*, pp. 217–228, Springer, 2012.

[47] C. Ryu, T. Lecroq, and K. Park, "Fast string matching for DNA sequences," *Theor. Comput. Sci.*, vol. 812, pp. 137–148, 2020.

[48] D. Cantone, S. Faro, and A. Pavone, "Linear and efficient string matching algorithms based on weak factor recognition," *ACM J. Exp. Algorithmics*, vol. 24, no. 1, pp. 1.8:1–1.8:20, 2019.

[49] S. Faro and M. O. Külekci, "Fast and flexible packed string matching," *J. Discrete Algorithms*, vol. 28, pp. 61–72, 2014.

[50] L. Salmela, J. Tarhio, and J. Kytöjoki, "Multipattern string matching with q-grams," *ACM J. Exp. Algorithmics*, vol. 11, 2006.

[51] S. Faro and M. O. Külekci, "Fast packed string matching for short patterns," in *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments,*

ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013 (P. Sanders and N. Zeh, eds.), pp. 113–121, SIAM, 2013.

[52] S. Faro and M. O. Külekci, "Fast and flexible packed string matching," *J. Discrete Algorithms*, vol. 28, pp. 61–72, 2014.

[53] J. Vuillemin, "A unifying look at data structures," *Commun. ACM*, vol. 23, no. 4, pp. 229–239, 1980.

[54] C. Hohlweg and C. Reutenauer, "Lyndon words, permutations and trees," *Theor. Comput. Sci.*, vol. 307, no. 1, pp. 173–178, 2003.

[55] M. Crochemore and L. M. S. Russo, "Cartesian and Lyndon trees," *Theor. Comput. Sci.*, vol. 806, pp. 1–9, 2020.

[56] E. D. Demaine, G. M. Landau, and O. Weimann, "On Cartesian trees and Range Minimum Queries," *Algorithmica*, vol. 68, no. 3, pp. 610–625, 2014.

[57] J. Shun and G. E. Blelloch, "A simple parallel Cartesian tree algorithm and its application to parallel suffix tree construction," *ACM Trans. Parallel Comput.*, vol. 1, no. 1, pp. 8:1–8:20, 2014.

[58] S. G. Park, A. Amir, G. M. Landau, and K. Park, "Cartesian tree matching and indexing," in *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy.* (N. Pisanti and S. P. Pissis, eds.), vol. 128 of *LIPIcs*, pp. 16:1–16:14, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[59] S. Song, C. Ryu, S. Faro, T. Lecroq, and K. Park, "Fast Cartesian tree matching," in *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings* (N. R. Brisaboa and S. J. Puglisi, eds.), vol. 11811 of *Lecture Notes in Computer Science*, pp. 124–137, Springer, 2019.

[60] G. Gu, S. Song, S. Faro, T. Lecroq, and K. Park, "Fast multiple pattern Cartesian tree matching," in *WALCOM: Algorithms and Computation - 14th International Conference, WALCOM 2020, Singapore, March 31 - April 2, 2020, Proceedings* (M. S. Rahman, K. Sadakane, and W. Sung, eds.), vol. 12049 of *Lecture Notes in Computer Science*, pp. 107–119, Springer, 2020.

[61] P. Gawrychowski, S. Ghazawi, and G. M. Landau, "On indeterminate strings matching," in *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark* (I. L. Gørtz and O. Weimann, eds.), vol. 161 of *LIPIcs*, pp. 14:1–14:14, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[62] M. Bataa, S. G. Park, A. Amir, G. M. Landau, and K. Park, "Finding periods in Cartesian tree matching," in *Combinatorial Algorithms - 30th International Workshop, IWOCA 2019, Pisa, Italy, July 23-25, 2019, Proceedings* (C. J. Colbourn, R. Grossi, and N. Pisanti, eds.), vol. 11638 of *Lecture Notes in Computer Science*, pp. 70–84, Springer, 2019.

[63] H. N. Gabow, J. L. Bentley, and R. E. Tarjan, "Scaling and related techniques for geometry problems," in *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA* (R. A. DeMillo, ed.), pp. 135–143, ACM, 1984.

[64] R. Cole and R. Hariharan, "Faster suffix tree construction with missing suffix links," *SIAM J. Comput.*, vol. 33, no. 1, pp. 26–42, 2003.