UNIVERSITÀ DEGLI STUDI DI CATANIA
Facoltà di Scienze Matematiche Fisiche e Naturali
Dottorato di ricerca in Informatica

ADVANCEMENTS IN FINITE-STATE METHODS FOR STRING MATCHING

Emanuele Giaquinta

A dissertation submitted in partial fulfillment of the requirements for the degree of "Research Doctorate in Computer Science"

Tesi presentata per il conseguimento del titolo di "Dottorato di Ricerca in Informatica" (XXIII ciclo)

Coordinatore	Tutor
Prof. Domenico Cantone	*Prof. Domenico Cantone*

Printed December 2010

Acknowledgements

There are some people who have accompanied me during the last three years and whom I would like to thank at the end of this journey. First of all, my supervisor, Professor Domenico Cantone, who encouraged me to start this scientific project: his continuous guidance and support have been fundamental for my growth as a researcher. Dr. Simone Faro introduced me to the field of string algorithms and has assisted me all along. It has been a pleasure to work with him in all the activities that we have been developing together. I am also grateful to my colleagues and to all my friends, without whom these years would have surely been less enjoyable. I would also like to thank Alessandro Sergi for many interesting conversations and for his friendly encouragement as well as Mauro Ferrario for his help in a number of occasions. Finally, *grazie* to my family for their continuous and overwhelming support and love.

Contents

Chapter 1

Introduction

One of the oldest methods to represent information is by means of written texts. A text can be defined as a coherent sequence of symbols which encodes information in a specific language. One obvious example are natural languages, which are typically used by humans to communicate in oral or written form. Other examples are DNA, RNA and protein sequences; DNA and RNA are nucleic acids that carry the genetic information of living beings and can be represented as sequences of the nucleobases (**C**ytosine, **G**uanine, **A**denine and **T**hymine or **U**racil) of their nucleotides. Proteins are molecules made of amino acids that are fundamental constituents of living organisms and can be represented by the sequence of amino acids (20 in total) encoded in the corresponding gene.

A natural problem which arises when dealing with such sequences is identifying specific patterns; as far as natural language texts are concerned, one is often interested in finding the occurrences of a given word or sentence; on the other side, an important problem in computational biology is finding given features in DNA sequences or determining the degree of similarity of two sequences. Both problems are particular instances of the *string matching* problem which has been investigated in a formal way since 1970 [30, 57].

Among other applications of the *string matching* problem we recall data scanning problems, such as intrusion detection or anti-virus scanning, and searching of patterns within images (which can be modelled as two-dimensional sequences).

Formally, given an alphabet of symbols Σ of size σ, a text T of n symbols and a pattern P of m symbols, the *string matching* problem consists in finding all positions in T in which P occurs. In its *online* version only the pattern can be preprocessed before searching; instead, in the *offline* version the text can be preprocessed to build a suitable data structure that allows searching for arbitrary patterns without traversing the whole text.

In this thesis we deal with *online string matching* only. The first linear solution for the *string matching* problem is the Knuth-Morris-Pratt algorithm [47], which solves the problem with $\mathcal{O}(n)$ worst-case searching time complexity and $\mathcal{O}(m)$-space complexity. This algorithm is based on the deterministic finite automaton for the language $\Sigma^* P$ and scans the text from left to right. The first sublinear string matching algorithm is due to Boyer and Moore [14]. The Boyer-Moore algorithm processes the text in windows of size m and scans each window from right to left. The idea is to stop when a mismatch occurs and then shift the window based on the characters read. The algorithm is quadratic in the worst case though, on the average, it shows a sublinear behaviour. Another important solutiond based on finite automata is the Backward-DAWG-Matching algorithm (BDM) [30], which uses the deterministic automaton for the language $\mathit{Suff}(P)$ of the suffixes of P (suffix automaton) [11]; in this case the text is processed in windows of size m and windows are scanned from right to left, as the Boyer-Moore algorithm. When a mismatch occurs, the window is shifted according to the length of the longest recognized prefix of the pattern that is aligned with the right side of the window. The algorithm has $\mathcal{O}(mn)$ worst-case time complexity and $\mathcal{O}(m)$-space complexity.

The optimal average time complexity of the *string matching* problem, under a Bernoulli model of probability where all the symbols are equiprobable, is equal to $\mathcal{O}(n \log_\sigma(m)/m)$ [70]. Such a bound has been achieved by the BDM algorithm. Another important finite automaton for the string matching problem is the factor oracle [2]; this automaton recognizes at least the factors of the input string and can be used in place of the suffix automaton in the BDM algorithm (this variant is known as BOM). The advantage of this automaton is that it is lighter because it has $m + 1$ states, as opposed to the suffix automaton which can have up to $2m - 1$ states. Although the solutions based on finite automata are suitable from a theoretical point of view, in practice they do not always perform well. Indeed, the algorithm which is usually implemented in computer programs is a simplified version of the Boyer-Moore algorithm due to Horspool [39]. The principle of simplicity is very relevant in string matching algorithms.

The situation changed in 1992 when Baeza-Yates and Gonnet introduced the technique known as bit-parallelism to simulate the nondeterministic version of the Knuth-Morris-Pratt automaton [8]. They devised a new algorithm, named Shift-And, which is a version of the Knuth-Morris-Pratt algorithm based on the nondeterministic finite automaton (NFA) recognizing Σ^*P. The virtue of this algorithm is that it is extremely simple and succint, thus achieving very good performance in practice. Bit-parallelism is a representation, based on bit-vectors, that exploits the regularity of some nondeterministic automata. The space needed by this encoding is $\mathcal{O}(\sigma\lceil m/w \rceil)$, where w is the size in bits of a computer word. Later, Navarro and Raffinot extended this result to the nondeterministic version of the suffix automaton [56]. They presented an algorithm, named BNDM, which is the version of the BDM algorithm based on the NFA for the language $\mathit{Suff}(P)$. Despite being very efficient in practice, algorithms based on bit-parallelism suffer from an intrinsic limitation: they have a $\mathcal{O}(\lceil m/w \rceil)$ overhead in the corresponding time complexity, which is due to the use of a representation based on bit-vectors. Hence, they do not scale efficiently as the pattern length grows. A few techniques have been proposed to workaround this limitation [56, 59], but they all have some drawbacks.

A natural generalization of the *string matching* problem is the *multiple string matching* problem: given a set $\mathcal{P}$ of patterns and a text T, the *multiple string matching* problem is tantamount to finding all the occurrences in T of the patterns in $\mathcal{P}$. The first linear solution for this problem based on finite automata is due to Aho and Corasick [1]. The Aho-Corasick algorithm uses a deterministic incomplete finite automaton for the language $\Sigma^*\mathcal{P}$ based on the *trie* for the input patterns, and is basically a generalization of the Knuth-Morris-Pratt algorithm. The first sublinear solution is due to Commentz-Walter [27] and is a generalization of the Boyer-Moore algorithm based on the *trie* for the input patterns. Generalizations of the BDM and of the BOM algorithms can be found in [57]. The nondeterministic versions of finite automata for the *multiple string matching* problem are more difficult to simulate because, in general, it is not true that for every state the number of outgoing edges is at most 1, which is one of the properties exploited by bit-parallelism. Nevertheless, there exist generalizations of the Shift-And and of the BNDM algorithms [57] that are based on a simplified *trie* in which no factoring of prefixes occurs.

An important variant of the string matching problem is the *approximate string matching* problem, which consists in finding all the occurrences of a pattern in a text allowing for a finite number of errors. Errors are formalized by means of a distance function on strings which maps two strings into the minimal cost of a sequence of edit operations that are needed to convert the first string into the second string. This problem is useful to find, for example, DNA subsequences after mutations, or spelling errors. Well known distance functions for this problem are the *edit distance* [49] (also called the *Levenshtein distance*) or the *Damerau edit distance* [31]. The edit operations in the former edit distance are *insertion*, *deletion*, and *substitution* of characters; instead, in the second case, one allows for *swaps* of characters, i.e., transpositions of two adjacent characters[1]. *Approximate string matching* under the *Damerau* distance is also known as *string matching with swaps* and was introduced in 1995 as one of the open problems in non-standard string matching [52]. A variant of this problem, known as *approximate string matching with swaps*, consists in computing, for each occurrence, also the corresponding number of swaps.

Another important variant of the *string matching* problem is the *compressed string matching* problem, which consists in searching for a pattern in a text stored in a compressed form. Compression in this case means reducing the size of the text representation without any loss of information, i.e., the original text can be completely recovered from its compressed version. Despite the fact that the price of external memory has lowered dramatically in recent years, the interest in data compression has not withered; hence, being able to perform text processing directly on the compressed text remains an interesting task. The two compression methods that are investigated in this thesis are prefix codes and the Burrows-Wheeler transform. Prefix codes are variable-length codes with the property that no codeword is a prefix of any other codeword in the set. This compression method is also known as *Huffman coding* [40]. The problem which arises when performing string matching on prefix codes is that decoding must be performed from left to right and no bit can be skipped. The Burrows-Wheeler transform [15] (BWT) is a reversible transformation which yields a permutation of the text that can be better compressed using the combination of a locally-adaptive encoding, such as move-to-front [10], and statistical methods [40, 67]. It is not possible to search for a pattern in a BWT encoded text without preprocessing the text once at least. Hence, Existing algorithms [9] for *string matching* on BWT encoded texts are not strictly online. These algorithms are able to compute how many times a given pattern occurs in a text and all the positions in which it occurs, but require more than one iteration over the compressed text.

[1] For an in-depth survey on approximate string matching see [53].

1.1 Results

The main results presented in this thesis are:

- a new encoding, based on bit-vectors, for regular NFAs as those for the languages $\Sigma^* P$ and *Suff*(P). The new representation, based on a particular factorization of strings, requires $\mathcal{O}(\sigma\lceil k/w\rceil)$ space and adds a $\mathcal{O}(\lceil k/w\rceil)$ overhead to the time complexity of the algorithms based on it, where $\lceil \frac{m}{\sigma}\rceil \leq k \leq m$ is the size of the factorization and m is the length of the input string. We show that bit-parallel string matching algorithms based on this encoding scale much better as m grows.

- a new encoding, based on bit-vectors, of the NFA for the language $\bigcup_{P\in\mathcal{P}} \Sigma^* P$ induced from the *trie* data structure for $\mathcal{P}$ and the NFA for the language $\bigcup_{P\in\mathcal{P}}$ *Suff*(P) induced from the DAWG data structure for $\mathcal{P}$. The new representation requires $\mathcal{O}((\sigma+m)\lceil m/w\rceil)$ space, where m is the number of states of the automaton.

- practical bit-parallel variants of the Wide-Window algorithm that exploit the bit-level parallelism to simulate two automata in parallel. In one case this approach makes it possible to double the shift performed by the algorithm.

- the definition of a distance for *approximate string matching* based on edit operations that involve substrings of the string, namely swaps of equal length adjacent substrings and reversal of substrings; we also present an algorithm, based on dynamic programming and on the DAWG data structure, to solve the *approximate string matching* problem under this distance.

- a simple variant of an algorithm for the *string matching with swaps* problem that is able to count, for each occurrence of the pattern, the corresponding number of swaps without any time and space overhead.

- a general algorithm designed to adapt Boyer-Moore like algorithms for *compressed string matching* in Huffman encoded texts; the new algorithm is able to skip bits when decoding.

- a new type of preprocessing for *online string matching* on BWT encoded texts to count the occurrences of a pattern; the new algorithms require one iteration only over the compressed text and use less space, on average, in the case of moderately large alphabets.

This thesis includes material from my original publications [21, 17, 24, 25, 23, 22, 18].

1.2 Organization of the thesis

The thesis is divided into seven chapters. Chapter 2 provides the reader with the basic notions needed to properly follow the results presented in the subsequent chapters; in particular, it introduces the basic notions and notations used to work with strings and finite state automata and also specifies some basic data structures. Chapter 3 focuses on the basic *string matching* problem; it presents a formal derivation of the solutions for this problem based on nondeterministic finite automata and on the bit-parallelism technique and includes some novel results as well. Chapter 4 deals with the *multiple string matching* problem and presents existing and novel results concerning methods based on nondeterministic finite automata and bit-parallelism. Chapters 5 and 6 focus on more complex variants of the string matching problem such as *approximate string matching* and *compressed string matching*, respectively, and present novel results for both problems. The conclusions are finally drawn in Chapter 7.

Chapter 2

Basic notions and definitions

2.1 Strings

A string P of length $|P| = m$ over a given finite alphabet Σ of size σ is any sequence of m characters of Σ. For $m = 0$, we obtain the empty string ε. Σ^* is the collection of all finite strings over Σ. We denote by $P[i]$ the $(i+1)$-th character of P, for $0 \leq i < m$. Likewise, the substring (or factor) of P contained between the $(i+1)$-th and the $(j+1)$-th characters of P is denoted by $P[i\,..\,j]$, for $0 \leq i \leq j < m$. We also put $P_i =_{\text{Def}} P[0\,..\,i]$, for $0 \leq i < m$, and make the convention that P_{-1} denotes the empty string ε. It is common to identify a string of length 1 with the character occurring in it. We also put $\mathit{first}(P) = P[0]$ and $\mathit{last}(P) = P[|P|-1]$.

For any two strings P and P', we write $P.P'$ or, more simply, PP' to denote the concatenation of P' with P. We also write $P' \sqsupseteq P$ ($P' \sqsupset P$) to indicate that P' is a (proper) suffix of P, i.e., $P = P''.P'$ for some nonempty string P''. Analogously, $P' \sqsubseteq P$ ($P' \sqsubset P$) denotes that P' is a (proper) prefix of P, i.e., $P = P'.P''$ for some (nonempty) string P''. We denote by $\mathit{Fact}(P)$ the set of the factors of P and by $\mathit{Suff}(P)$ the set of the suffixes of P. We write P^{r} to denote the reverse of the string P, i.e., $P^{\mathsf{r}} = P[m-1]P[m-2]\ldots P[0]$. Given a finite set of patterns $\mathcal{P}$, let $\mathcal{P}^{\mathsf{r}} =_{\text{Def}} \{P^{\mathsf{r}} \mid P \in \mathcal{P}\}$ and $\mathcal{P}_l =_{\text{Def}} \{P[0\,..\,l-1] \mid P \in \mathcal{P}\}$. We also define $\mathit{size}(\mathcal{P}) =_{\text{Def}} \sum_{P \in \mathcal{P}} |P|$ and extend the maps $\mathit{Fact}(\cdot)$ and $\mathit{Suff}(\cdot)$ to $\mathcal{P}$ by putting $\mathit{Fact}(\mathcal{P}) =_{\text{Def}} \bigcup_{P \in \mathcal{P}} \mathit{Fact}(P)$ and $\mathit{Suff}(\mathcal{P}) =_{\text{Def}} \bigcup_{P \in \mathcal{P}} \mathit{Suff}(P)$.

2.2 Nondeterministic finite automata

A nondeterministic finite automaton (NFA) with ε-transitions is a 5-tuple $N = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the collection of final states, Σ is an alphabet, and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathscr{P}(Q)$ is the transition function ($\mathscr{P}(\cdot)$ is the powerset operator).[1] For each state $q \in Q$, the ε-closure of q, denoted as ECLOSE(q), is the set of states that are reachable from q by following zero or more ε-transitions. ECLOSE can be generalized to a set of states by putting ECLOSE$(D) =_{\text{Def}} \bigcup_{q\in D}$ ECLOSE(q). In the case of an NFA without ε-transitions, we have ECLOSE$(q) = \{q\}$, for any $q \in Q$.

The *extended* transition function $\delta^* : Q \times \Sigma^* \rightarrow \mathscr{P}(Q)$ induced by δ is defined recursively by

$$\delta^*(q,u) =_{Def} \begin{cases} \text{ECLOSE}(q) & \text{if } u = \varepsilon, \\ \bigcup_{p\in\delta^*(q,v)} \text{ECLOSE}(\delta(p,c)) & \text{if } u = v.c, \text{ for some } c \in \Sigma \text{ and } v \in \Sigma^*. \end{cases}$$

In particular, when no ε-transition is present, then

$$\delta^*(q,\varepsilon) = \{q\} \quad \text{and} \quad \delta^*(q, v.c) = \delta(\delta^*(q,v), c)\,.$$

Both the transition function δ and the extended transition function δ^* can be naturally generalized to handle set of states, by putting $\delta(D,c) =_{Def} \bigcup_{q\in D} \delta(q,c)$ and $\delta^*(D,u) =_{Def} \bigcup_{q\in D} \delta^*(q,u)$, respectively, for $D \subseteq Q$, $c \in \Sigma$, and $u \in \Sigma^*$. The *extended* transition function satisfies the following property:

$$\delta^*(q, u.v) = \delta^*(\delta^*(q,u), v), \text{ for all } u, v \in \Sigma^*\,. \tag{2.1}$$

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, we define a *reachable configuration* of N any subset $D \subseteq Q$ such that $D = \delta^*(q_0, u)$, for some $u \in \Sigma^*$.

2.3 Bitwise operators on computer words and the bit-vector data structure

We recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|", the `left shift` "≪" and `right shift`

[1] In the case of NFAs with no ε-transitions, the transition function has the form $\delta : Q \times \Sigma \rightarrow \mathscr{P}(Q)$. For the basics on NFAs, the reader is referred to [38].

"$\gg$" operators (which shifts to the left (right) its first argument by a number of bits equal to its second argument), and the unary bitwise `not` operator "$\sim$".

The functions that compute the first and the last bit set to 1 of a word x are $\lfloor \log_2(x \,\&\, ({\sim}\, x + 1)) \rfloor$ and $\lfloor \log_2(x) \rfloor$, respectively. Modern architectures include assembly instructions for this purpose; for example, the *x86* family provides the `bsf` and `bsr` instructions, whereas the *powerpc* architecture provides the `cntlzw` instruction. For a comprehensive list of machine-independent methods for computing the index of the first and last bit set to `1`, see [7].

A set $\mathcal{S} \subseteq \{1, 2, \ldots, m\}$ of integers can be conveniently represented as a vector D of m bits. The i-th bit of D is set to 1 if the element i belongs to $\mathcal{S}$, to 0 otherwise. If w is the size in bits of a computer word, $\lceil m/w \rceil$ words are needed to represent $\mathcal{S}$. Using this representation, typical operations on sets map onto simple bitwise operations:

- $i \in \mathcal{S} \iff (D \,\&\, (1 \ll i)) \neq 0$
- $\mathcal{S} \cup \{i\} \iff D \mid (1 \ll i)$
- $\mathcal{S}_1 \cup \mathcal{S}_2 \iff D_1 \mid D_2$
- $\mathcal{S}_1 \cap \mathcal{S}_2 \iff D_1 \,\&\, D_2$
- $\mathcal{S}_1 \setminus \mathcal{S}_2 \iff D_1 \,\&\, {\sim}\, D_2$
- $\mathcal{S}^c \iff {\sim}\, D\,.$

This representation allows to exploit the bit-level parallelism of the operations on computer words, cutting down the number of operations that an algorithm performs by a factor of w, as the time complexity of each operation is $\Theta(\lceil m/w \rceil)$.

2.4 The *trie* data structure

Given a set $\mathcal{P}$ of patterns over a finite alphabet Σ, the *trie* $\mathcal{T}_{\mathcal{P}}$ associated with $\mathcal{P}$ is a rooted directed tree, whose edges are labeled by single characters of Σ, such that

(i) distinct edges out of a same node are labeled by distinct characters;

(ii) all paths in $\mathcal{T}_{\mathcal{P}}$ from the root are labeled by prefixes of the strings in $\mathcal{P}$;

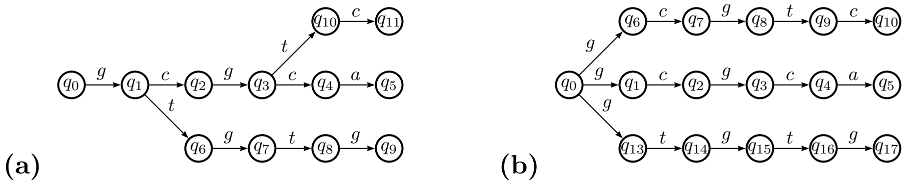


Figure 2.1: **(a)** *trie* and **(b)** *maximal trie* for the set of strings {**gcgca**,**gtgtg**,**gcgtc**}.

(iii) for each string P in $\mathcal{P}$ there exists a path in $\mathcal{T}_{\mathcal{P}}$ from the root which is labeled by P.

For any node p in the trie $\mathcal{T}_{\mathcal{P}}$, we denote by $lbl(p)$ the string which labels the path from the root of $\mathcal{T}_{\mathcal{P}}$ to p and put $len(p) =_{\text{Def}} |lbl(p)|$. Plainly, the map lbl is injective. Additionally, for any edge (p, q) in $\mathcal{T}_{\mathcal{P}}$, the label of (p, q) is denoted by $lbl(p, q)$.

For a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ over an alphabet Σ, the *maximal trie* of $\mathcal{P}$ is the trie $\mathcal{T}_{\mathcal{P}}^{max}$ obtained by merging into a single node the roots of the linear tries $\mathcal{T}_{\mathcal{P}_1}, \mathcal{T}_{\mathcal{P}_2}, \ldots, \mathcal{T}_{\mathcal{P}_r}$ relative to the patterns $P_1, P_2, \ldots, P_r$, respectively. Strictly speaking, the maximal trie is a nondeterministic trie, as property (i) above may not hold at the root. An example of *trie* and *maximal trie* is shown in Fig. 2.1.

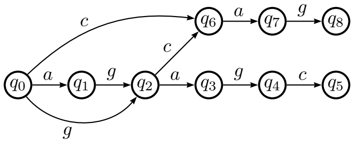
2.5 The DAWG data structure

The *directed acyclic word graph* (DAWG) [11, 28, 30] for a finite set of patterns $\mathcal{P}$ is a data structure representing the set $Fact(\mathcal{P})$. To describe it precisely, we need the following definitions. Let us denote by *end-pos*(u) the set of all positions in $\mathcal{P}$ where an occurrence of u ends, for $u \in \Sigma^*$; more formally, let

$$\textit{end-pos}(u) =_{\text{Def}} \{(P, j) \mid u \sqsupseteq P_j, \text{ with } P \in \mathcal{P} \text{ and } |u| - 1 \leq j < |P|\}\,.$$

For instance, we have *end-pos*$(\varepsilon) = \{(P, j) \mid P \in \mathcal{P}$ and $-1 \leq j < |P|\}$, since $\varepsilon \sqsupseteq P_j$, for each $P \in \mathcal{P}$ and $-1 \leq j < |P|$ (we recall that $P_{-1} = \varepsilon$, by convention).
[2]

[2] In the case of a single pattern, i.e., $|\mathcal{P}| = 1$, *end-pos* is just a set of positions rather than of pairs.



q	suf	R
q_0		$\{\epsilon\}$
q_1	q_0	$\{a\}$
q_2	q_0	$\{ag, g\}$
q_3	q_1	$\{aga, ga\}$
q_4	q_2	$\{agag, gag\}$
q_5	q_6	$\{agagc, gagc\}$
q_6	q_0	$\{agc, gc, c\}$
q_7	q_1	$\{agca, gca, ca\}$
q_8	q_2	$\{agcag, gcag, cag\}$

Figure 2.2: DAWG for the set of strings {**agagc**,**agcag**}.

We also define an equivalence relation $R_{\mathcal{P}}$ over Σ^* by putting

$$u \; R_{\mathcal{P}} \; v \iff_{\text{Def}} \textit{end-pos}(u) = \textit{end-pos}(v)\,, \tag{2.2}$$

for $u, v \in \Sigma^*$, and denote by $R_{\mathcal{P}}(u)$ the equivalence class of $R_{\mathcal{P}}$ containing the string u. Futhermore, let

$$\textit{val}(R_{\mathcal{P}}(u)) =_{\text{Def}} \text{the longest string in the equivalence class } R_{\mathcal{P}}(u)\,, \tag{2.3}$$

and $\textit{length}(R_{\mathcal{P}}(u)) =_{\text{Def}} |\textit{val}(R_{\mathcal{P}}(u))|$. Then the DAWG for a finite set $\mathcal{P}$ of patterns is a directed acyclic graph (V, E) with an edge labeling function $\textit{lbl}()$, where $V = \{R_{\mathcal{P}}(u) \,|\, u \in \textit{Fact}(\mathcal{P})\}$, $E = \{(R_{\mathcal{P}}(u), R_{\mathcal{P}}(uc)) \mid u \in \Sigma^*, c \in \Sigma, uc \in \textit{Fact}(\mathcal{P})\}$, and $\textit{lbl}(R_{\mathcal{P}}(u), R_{\mathcal{P}}(uc)) = c$, for $u \in \Sigma^*$, $c \in \Sigma$ such that $uc \in \textit{Fact}(\mathcal{P})$ (cf. [12]).

We also define a failure function, $\textit{suf} : \textit{Fact}(\mathcal{P}) \setminus \{\varepsilon\} \rightarrow \textit{Fact}(\mathcal{P})$, named *suffix link*, by putting

$$\textit{suf}(u) =_{\text{Def}} \text{ the longest } v \in \textit{Suff}(u) \text{ such that } y \not{R}_{\mathcal{P}} x \tag{2.4}$$

for $u \in \textit{Fact}(\mathcal{P}) \setminus \{\varepsilon\}$.

The $\textit{suf}(\cdot)$ and $\textit{end-pos}(\cdot)$ functions can be extended to the equivalence classes of $R_{\mathcal{P}}$ not containing ε, by putting for all $q \in V \setminus \{R_{\mathcal{P}}(\varepsilon)\}$

$$\begin{aligned} \textit{suf}(q) &=_{\text{Def}} R_{\mathcal{P}}(\textit{suf}(\textit{val}(q))) \\ \textit{end-pos}(q) &=_{\text{Def}} \textit{end-pos}(\textit{val}(q))\,. \end{aligned}$$

An example of DAWG is shown in Fig. 2.2. The DAWG for a finite set of patterns $\mathcal{P}$ naturally induces the deterministic automaton $\mathscr{F}(\mathcal{P}) = (Q, \Sigma, \delta, \textit{root}, F)$ whose language is $\textit{Fact}(\mathcal{P})$, where

- $Q = \{R_{\mathcal{P}}(u) : u \in \mathit{Fact}(\mathcal{P})\}$ is the set of states,
- Σ is the alphabet of the characters in P,
- $\delta : Q \times \Sigma \to Q$ is the transition function defined by:

$$\delta(R_{\mathcal{P}}(u), c) = \begin{cases} \{R_{\mathcal{P}}(uc)\} & \text{if } uc \in \mathit{Fact}(\mathcal{P}) \\ \emptyset & \text{otherwise} \end{cases}$$

- $\mathit{root} = R_{\mathcal{P}}(\varepsilon)$ is the initial state,
- $F = Q$ is the set of final states.

2.6 Experimental framework

The experimental results presented in this thesis have been obtained using the following setup: all the algorithms have been implemented in the `C` programming language and have been compiled with the `GNU C Compiler` 4.0, using the optimization options `-O2 -fno-guess-branch-probability`; the running times have been measured with a high resolution timer by first copying the whole input in memory and then taking the mean over a certain number of runs of the time needed by the algorithm to end. The word size is 32 in all the tests. The corpus used to carry out the tests consists of the following files:

(i) the English King James version of the "Bible" ($n = 4,047,392, \sigma = 63$),

(ii) the CIA World Fact Book ($n = 2,473,400, \sigma = 94$),

(iii) a DNA sequence of the *Escherichia coli* genome ($n = 4,638,690, \sigma = 4$),

(iv) a protein sequence of the *Saccharomyces cerevisiae* genome ($n = 2,900,352, \sigma = 20$),

(v) the Spanish novel "Don Quixote" by Cervantes ($n = 2347740, \sigma = 86$).

where n denotes the number of characters and σ denotes the alphabet size. Files (i), (ii), and (iii) are from the Canterbury Corpus [3], file (iv) is from the Protein Corpus [4], and file (v) is from Project Gutenberg [5].

[3] `http://corpus.canterbury.ac.nz/`

[4] `http://data-compression.info/Corpora/ProteinCorpus/`

[5] `http://www.gutenberg.org/`

Chapter 3

The string matching problem

Given a pattern P of length m and a text T of length n, both drawn from a common finite alphabet Σ of size σ, the *string matching* problem consists in finding all the occurrences of P in T. The optimal average time complexity of the problem is equal to $\mathcal{O}(n \log_\sigma(m)/m)$ [70].

The most efficient solutions for the string matching problem are based on finite automata. As already recalled, the first linear-time solution is the Knuth-Morris-Pratt algorithm [47]. This algorithm uses an implicit representation of the automaton for the language Σ^*P based on the *border* function of the pattern [30] and solves the problem with $\mathcal{O}(n)$ worst-case searching time complexity and $\mathcal{O}(m)$-space complexity. The lower bound on average has been achieved by the BDM algorithm [30], that is based on the deterministic automaton for the language $\mathit{Suff}(P)$. The worst-case searching time complexity of this algorithm is $\mathcal{O}(mn)$ while its space complexity is $\mathcal{O}(m)$. Baeza-Yates and Gonnet [8] introduced a technique, known as bit-parallelism, to simulate the nondeterministic versions of these automata. They devised a new algorithm, named Shift-And, that is the version of the Knuth-Morris-Pratt algorithm based on the NFA recognizing Σ^*P. Later, Navarro and Raffinot [56] presented the BNDM algorithm, which is the version of the BDM algorithm based on the nondeterministic suffix automaton for the language $\mathit{Suff}(P)$. Albeit algorithms based on bit-parallelism are very efficient and compact, they have an $\mathcal{O}(\lceil m/w \rceil)$ overhead, where w is the size in bits of a computer word, as compared to the corresponding algorithms based on a deterministic automaton. This limitation is intrinsic, since the bit-parallelism encoding is based on bit-vectors.

In this chapter we present new results concerning bit-parallelism. In Section 3.1 we formally define the NFAs for the *string matching* problem and in Section 3.2 we introduce the bit-parallelism technique and the two main algorithms based on it. Then, in Section 3.3 we present a new encoding, based on bit-vectors, for regular NFAs as those for the languages $\Sigma^* P$ and $\mathit{Suff}(P)$. By exploiting a particular factorization of strings, the new representation allows to encode automata by smaller bit-vectors resulting in faster algorithms. In Section 3.4 we present a method to increase the parallelism in bit-parallel algorithms; more precisely, we introduce variants of the Wide-Window algorithm [37] that simulate two automata in parallel. In one case this approach makes it possible to double the shift performed by the algorithm at each iteration.

3.1 Automata based solutions for the string matching problem

There are two main automata which are the core building blocks in different algorithms for the *string matching* problem. Let $P \in \Sigma^m$ be a string of length m. The first automaton is the one for the language $\Sigma^* P$, i.e., the automaton that recognizes all the strings that have P as suffix. The second one is the so called *suffix automaton*, which is the automaton that recognizes the language $\mathit{Suff}(P)$ of the suffixes of P. It is important to observe that the nondeterministic versions of these automata are very regular. We indicate with $\mathscr{A}(P) = (Q, \Sigma, \delta, q_0, F)$ the nondeterministic automaton for the language $\Sigma^* P$, where:

- $Q = \{q_0, q_1, \dots, q_m\}$ (q_0 is the initial state)
- the transition function $\delta : Q \times \Sigma \longrightarrow \mathscr{P}(Q)$ is defined by:

$$\delta(q_i, c) =_{\text{Def}} \begin{cases} \{q_0, q_1\} & \text{if } i = 0 \text{ and } c = P[0] \\ \{q_0\} & \text{if } i = 0 \text{ and } c \neq P[0] \\ \{q_{i+1}\} & \text{if } 1 \leq i < m \text{ and } c = P[i] \\ \emptyset & \text{otherwise} \end{cases}$$

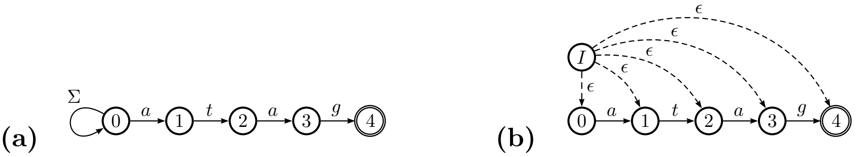


Figure 3.1: **(a)** Nondeterministic automaton and **(b)** nondeterministic suffix automaton for the pattern **atag**.

- $F = \{q_m\}$ (F is the set of final states).

Likewise, we denote by $\mathscr{S}(P) = (Q, \Sigma, \delta, I, F)$ the nondeterministic suffix automaton with ε-transitions for the language *Suff*(P), where:

- $Q = \{I, q_0, q_1, \ldots, q_m\}$ (I is the initial state)

- the transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \mathscr{P}(Q)$ is defined by:

$$\delta(q, c) =_{\mathrm{Def}} \begin{cases} \{q_{i+1}\} & \text{if } q = q_i \text{ and } c = P[i] \quad (0 \leq i < m) \\ Q & \text{if } q = I \text{ and } c = \varepsilon \\ \emptyset & \text{otherwise} \end{cases}$$

- $F = \{q_m\}$ (F is the set of final states).

An example of both automata is shown in Fig. 3.1.

The algorithms based on the automata $\mathscr{A}(P)$ and $\mathscr{S}(P)$ for searching a pattern P in a text T work by moving a logical window of size $|P|$ over T. The method based on the automaton $\mathscr{A}(P)$ computes, for a given window ending at position j in T, all the prefixes of P that are suffixes of T_j. In particular, there is an occurrence of P if the prefix of length m is found. The algorithm always shifts the current window by one position to the right, i.e., it checks every window. Instead, the method based on the automaton $\mathscr{S}(P)$ computes, for a given window ending at position j in T, the longest prefix of P that is a suffix of T_j by reading the window from right to left with $\mathscr{S}(P^r)$. As in the previous method, there is an occurrence of P if the length of the longest prefix found is m. The algorithm shifts the current window by a number of positions that depends on the length of the longest *proper* prefix of P recognized. This approach makes it possible to skip windows.

Given a pattern P of length m, the automaton $\mathscr{A}(P)$ can be used to find the occurrences of P in a given text T, by observing that P has an occurrence in T ending at position i, i.e., $P \sqsupseteq T_i$, if and only if $\delta^*_A(q_0, T[0\,..\,i])$ contains the final state q_m. Thus, to find all the occurrences of P in T, it suffices to compute the set $\delta^*_A(q_0, T_i) \cap F$, for $i = 0, 1, \ldots, |T| - 1$.

Given a pattern P of length m, the automaton $\mathscr{S}(P^{\mathsf{r}})$ can be used to find the occurrences of P in a text T by observing that P has an occurrence in T ending at position i, i.e., $P \sqsupseteq T_i$, if and only if $\delta^*_{S^{\mathsf{r}}}(q_0, (T[i-m+1\,..\,i])^{\mathsf{r}})$ contains the final state q_m. Hence, to find all the occurrences of P in T, one can compute $\delta^*_{S^{\mathsf{r}}}(q_0, (T[i-m+1\,..\,i])^{\mathsf{r}}) \cap F$, for $i = m-1, m, \ldots, |T|-1$. With this approach it is possible to skip windows: in fact, for a window of T of size m ending at position i, let l be the length of the longest proper suffix of $T[i-m+1\,..\,i]$ such that $\delta^*_{S^{\mathsf{r}}}(q_0, (T[i-l+1\,..\,i])^{\mathsf{r}}) \cap F \neq \emptyset$. It is easy to see that l is the length of the longest prefix of P that is a suffix of T_i. Then, the windows at positions $i, i+1, \ldots i+m-l-1$ can be safely skipped.

3.2 The bit-parallelism technique

Bit-parallelism is a technique, based on bit-vectors, that was introduced by Baeza-Yates and Gonnet in [8] to simulate efficiently nondeterministic automata. The first algorithms based on this technique are the well-known Shift-And [8] and BNDM [56]. The trivial way to encode a nondeterministic automaton of m states is by i) finding a linear ordering of its states, ii) representing the automaton configurations as bit vectors, such that bit i is set iff state with position i is active, and iii) by tabulating the transition function $\delta(D, c)$, for $D \subseteq Q$, $c \in \Sigma$. However, the main problem of this representation is that the space in bits needed to represent the transition function is $(2^m \cdot \sigma) \cdot m$, which is exponential in m. Bit-parallelism takes advantage of the regularity of the automata $\mathscr{A}(P)$ and $\mathscr{S}(P)$ to efficiently encode the transition function in a different way. The construction can be derived by starting from a result that was first formalized for the Glushkov automaton and that can be immediately generalized to a certain class of NFAs as follows (cf. [58]).

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA with ε-transitions such that up to the ε-transitions, for each state $q \in Q$, either

(i) all the incoming transitions in q are labeled by a same character, or

B=

σ	q_1	q_2	q_3	q_4
a	1	0	1	0
t	0	1	0	0
g	0	0	0	1

Figure 3.2: The map $B(\cdot)$ of the automaton $\mathscr{A}(\mathbf{atag})$.

(ii) all the incoming transitions in q originate from a unique state.

Let $B(c)$, for $c \in \Sigma$, be the set of states of N with an incoming transition labeled by c, i.e.,

$$B(c) \ =_{\mathrm{Def}} \{q \in Q \mid q \in \delta(p, c), \text{ for some } p \in Q\} \,.$$

Likewise, let *Follow*(q), for $q \in Q$, be the set of states reachable from state q with one transition over a character in Σ, i.e.,

$$\mathit{Follow}(q) \ =_{\mathrm{Def}} \bigcup_{c \in \Sigma} \delta(q, c) \,.$$

and let

$$\Phi(D) \ =_{\mathrm{Def}} \bigcup_{q \in D} \mathit{Follow}(q) \,,$$

for $D \subseteq Q$. Then the following result holds.

Lemma 3.1 (cf. [58])**.** *For every $q \in Q$, $D \subseteq Q$, and $c \in \Sigma$, we have*

(a) $\delta(q, c) = \mathit{Follow}(q) \cap B(c)$;

(b) $\delta(D, c) = \Phi(D) \cap B(c)$.

Proof. Concerning (a), we notice that $\delta(q, c) \subseteq \mathit{Follow}(q) \cap B(c)$ holds plainly. To prove the converse inclusion, let $p \in \mathit{Follow}(q) \cap B(c)$. Then $p \in \delta(q, c') \cap \delta(q', c)$, for some $c' \in \Sigma$ and $q' \in Q$. If p satisfies condition (i), then $c' = c$, and therefore $p \in \delta(q, c)$. On the other hand, if p satisfies condition (ii), then $q = q'$ and therefore $p \in \delta(q, c)$ follows again.

From (a), we obtain immediately (b), since

$$\begin{aligned} \delta(D, c) &= \bigcup_{q \in D} \delta(q, c) = \bigcup_{q \in D} (\mathit{Follow}(q) \cap B(c)) \\ &= \bigcup_{q \in D} \mathit{Follow}(q) \cap B(c) = \Phi(D) \ \cap \ B(c) \,. \end{aligned}$$

□

```
Shift-And ($P$, $m$, $T$, $n$)
for $c \in \Sigma$ do $B[c] \leftarrow 0^m$
for $i \leftarrow 0$ to $m-1$ do
    $B[P[i]] = B[P[i]] \mid (0^{m-1}1 \ll i)$
$D \leftarrow 0^m$
for $j \leftarrow 0$ to $n-1$ do
    $D \leftarrow ((D \ll 1) \mid 0^{m-1}1) \& B[T[j]]$
    if $D \& 10^{m-1} \neq 0^m$ then Output($j$)
```

Figure 3.3: The Shift-And algorithm.

Provided that one finds an efficient way of storing and accessing the maps $\Phi(\cdot)$ and $B(\cdot)$, equation (b) of Lemma 3.1 is particularly suitable for a representation based on bit-vectors, as set intersection can be readily implemented by the bitwise and operation.

The map $B(\cdot)$ can be encoded with $\sigma \cdot m$ bits, independently of the automaton structure, using an array B of σ bit-vectors, each of size m, where the i-th bit of $B[c]$ is set if there is an incoming transition in state with position i labeled by c. Instead, a generic encoding of the $\Phi(\cdot)$ map requires $2^m \cdot m$ bits, which is exponential in the number m of states.

While this result allows to reduce the total bits needed to represent the transition function to $(2^m + \sigma) \cdot m$, it can still be improved. In fact, depending on the automaton structure, it is possible to find a more efficient way to compute the map $\Phi(\cdot)$.

3.2.1 The Shift-And algorithm

The Shift-And algorithm simulates the nondeterministic automaton that recognizes the language $\Sigma^* P$, for a given string P of length m. An Automaton configuration $\delta^*(q_0, S)$ on input $S \in \Sigma^*$ is encoded as a bit-vector D of m bits (the initial state does not need to be represented as it is always active), where the i-th bit of D is set to 1 iff state q_{i+1} is active, i.e., $q_{i+1} \in \delta^*(q_0, S)$, for $i = 0, \ldots, m-1$. The map $B(\cdot)$ is encoded, as described above, using an array B of σ bit-vectors, each of size m, where the i-th bit of $B[c]$ is set iff $\delta(q_i, c) = q_{i+1}$ or equivalently iff $P[i] = c$, for $c \in \Sigma$, $0 \leq i < m$. An example of the $B(\cdot)$ map is shown in Fig. 3.2. As far as the $\Phi(\cdot)$ map is concerned, observe that *Follow*$(q_i) = \{q_{i+1}\}$, for $i = 1, \ldots, m-1$, and *Follow*$(q_0) = \{q_0, q_1\}$. Thus, for any automaton configuration D,

$$\Phi(D) = \bigcup_{q_i \in D} \{q_{i+1}\} \cup \{q_0\}\,,$$

which becomes $\bigcup_{q_i \in D}\{q_{i+1}\} \cup \{q_1\}$ if we do not represent the initial state. Hence, if D is represented as a bit-vector, we can compute $\Phi(D)$ with a bitwise `left shift` operation of one unit and a bitwise `or` with 1 (represented as $0^{m-1}1$).

For a configuration D of the NFA, a transition on character c can then be implemented by the bitwise operations

$$D \leftarrow ((D \ll 1) \mid 1) \ \&\ B[c]\,.$$

When a search starts, the initial configuration D is initialized to 0^m. Then, the automaton configuration is updated for each text character, as described before, by reading the text from left to right. For each position j in T, if D is the automaton configuration after having read the j-th character, it holds that $P_i \sqsupseteq T_j$ iff bit i is set in D. Thus, to verify if there is a match at position j it suffices to check that the $(m-1)$-th bit is set in D. The worst-case searching time complexity of the Shift-And algorithm is $\mathcal{O}(n\lceil m/w \rceil)$ while the space complexity is $\mathcal{O}(\sigma\lceil m/w \rceil)$. The pseudocode of the algorithm is shown in Fig. 3.3.

3.2.2 The Backward-Nondeterministic-DAWG-Matching algorithm

The Backward-Nondeterministic-DAWG-Matching algorithm (BNDM, for short) simulates the nondeterministic suffix automaton for P^r with the bit-parallelism technique, using an encoding similar to that described above for the Shift-And algorithm. The automaton configuration is encoded again as a bit vector D of m bits. The i-th bit of D is set to 1 iff state q_{i+1} is active, for $i = 0, 1, \ldots, m-1$, and D is initialized to 1^m, since after the ε-closure of the initial state I all states q_i represented in D are active. The first transition on character c is implemented as $D \leftarrow (D \ \&\ B[c])$, while any subsequent transition on character c can be implemented as

$$D \leftarrow ((D \ll 1) \ \&\ B[c])\,.$$

This algorithm works by shifting a window of length m over the text. Specifically, for each window alignment, it searches the pattern by scanning the current window backwards and updating the automaton configuration accordingly.

```
BNDM $(P, m, T, n)$
for $c \in \Sigma$ do $B[c] \leftarrow 0^m$
for $i \leftarrow 0$ to $m - 1$ do
    $c \leftarrow P[m - 1 - i]$
    $B[c] = B[c] \mid (0^{m-1}1 \ll i)$
$j \leftarrow m - 1$
while $j < n$ do
    $k \leftarrow 0, last \leftarrow 0$
    $D \leftarrow 1^m$
    while $D \neq 0^m$ do
        $D \leftarrow D \,\&\, B[T[j - k]]$
        $k \leftarrow k + 1$
        if $D \,\&\, 10^{m-1} \neq 0^m$ then
            if $k > 0$ then
                $last \leftarrow k$
            else Output$(j)$
        $D \leftarrow D \ll 1$
    $j \leftarrow j + m - last$
```

Figure 3.4: The Backward-Nondeterministic-DAWG Matching algorithm.

Let j be the ending position of the current window. At each iteration k, for $k = 1, \dots, m$, the automaton configuration is updated by performing a transition on character $T[j - k + 1]$, as described before. After having performed the k-th transition, it holds that

$$T[j - k + 1, \dots, j] = P[m - 1 - i, \dots, m - 2 - i + k]$$

iff bit i is set in D, for $i = k - 1, \dots, m - 1$.

Each time a suffix of P^r (i.e., a prefix of P) is found, i.e., when prior to the left shift the $(m - 1)$-th bit of D & $B[c]$ is set, the suffix length k is recorded in a variable *last*. A search ends when either D becomes zero (i.e., when no factor of P can be recognized) or when the algorithm has performed m iterations (i.e., when a match has been found). The window is then shifted to the starting position of the longest recognized proper prefix, i.e., to position $j + m - last$. The worst-case searching time complexity of the BNDM algorithm is $\mathcal{O}(nm\lceil m/w \rceil)$, while its space complexity is $\mathcal{O}(\sigma \lceil m/w \rceil)$. The algorithm BNDM is optimal on average as BDM. The pseudocode of the algorithm is shown in Fig. 3.4.

3.2.3 Bit-parallelism limitations

When the pattern size m is larger than w, the configuration bit-vector and all auxiliary bit-vectors need to be splitted over $\lceil m/w \rceil$ multiple words. For this reason the performance of the Shift-And and BNDM algorithms, and, more in general, of bit-parallel algorithms degrade considerably as $\lceil m/w \rceil$ 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 possible candidate occurrences of the pattern. When an occurrence of the selected substring is found, a subsequent naive verification phase allows to establish whether it 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 w, which may be much smaller than m.

3.3 Tighter packing for bit-parallelism

We present a new encoding of the configurations of the nondeterministic (suffix) automaton for a given pattern P of length m, which on the average requires less than m bits and can be used within the bit-parallel framework. The effect is that bit-parallel string matching algorithms based on this encoding scale much better as m grows, at the price of a larger space complexity. We will illustrate such a point experimentally with the Shift-And and the BNDM algorithms, but our proposed encoding can also be applied to other variants of the BNDM algorithm.

Our encoding will have the form (D, a), where D is a k-bit vector, with $k \leqslant m$ (on the average k is much smaller than m), and a is an alphabet symbol (the last text character read) that will be used as a parameter in the bit-parallel simulation with the vector D.

The encoding (D, a) is obtained by suitably factorizing the simple bit-vector encoding for NFA configurations presented in the previous section. More specifically, it is based on the following pattern factorization:

Definition 3.1 (1-factorization)**.** *Let $P \in \Sigma^m$. A 1-factorization $\boldsymbol{u}$ of size k of P is a sequence $\langle u_1, u_2, \ldots, u_k \rangle$ of nonempty substrings of P such that:*

(a) $P = u_1 u_2 \ldots u_k$ *;*

(b) each factor u_j in $\boldsymbol{u}$ contains at most one *occurrence of any of the characters in the alphabet Σ, for $j = 1, \ldots, k$.*

For a given 1-factorization $\boldsymbol{u} = \langle u_1, u_2, \ldots, u_k \rangle$ of P, we put

$$r_j^{\boldsymbol{u}} \; =_{\mathit{Def}} |u_1 u_2 \ldots u_{j-1}| \,, \tag{3.1}$$

for $j = 1, 2, \ldots, k+1$ *(so that* $r_1^{\boldsymbol{u}} = 0$ *and* $r_{k+1}^{\boldsymbol{u}} = m$*) and call the numbers* $r_1^{\boldsymbol{u}}, r_2^{\boldsymbol{u}}, \ldots, r_{k+1}^{\boldsymbol{u}}$ *the* indices of $\boldsymbol{u}$*. Plainly,* $r_j^{\boldsymbol{u}}$ *is the index in* P *of the first character of the factor* u_j*, for* $j = 1, 2, \ldots, k$*.*

A 1*-factorization of* P *is* minimal *if such is its size.*

Observe that the size k of a 1-factorization $\boldsymbol{u}$ of a string $P \in \Sigma^m$ satisfies the condition

$$\left\lceil \frac{m}{\sigma} \right\rceil \leq k \leq m\,.$$

Indeed, as the length of any factor in $\boldsymbol{u}$ is limited by the size σ of the alphabet Σ, then $m \leq k\sigma$, which implies $\lceil \frac{m}{\sigma} \rceil \leq k$. The second inequality is immediate and occurs when P has the form a^m, for some $a \in \Sigma$, in which case P has only the 1-factorization of size m whose factors are all equal to the single character string a.

As we shall show below, the size of the bit-vector D in our encoding depends on the size of the 1-factorization used; as a result, a minimal 1-factorization yields the smallest vector. A greedy approach to construct a 1-factorization of smallest size for a string P consists in computing the longest prefix u_1 of P containing no repeated characters and then recursively 1-factorize the string P deprived of its prefix u_1, as in the procedure Greedy-1-Factorize shown below.

```
Greedy-1-Factorize($P$)
  if $P$ is the empty string $\varepsilon$ then
    return the empty sequence $\langle\,\rangle$
  else
    $u_1 \leftarrow$ longest prefix of $P$ containing no repeated character
    $P' \leftarrow$ the suffix of $P$ such that $P = u_1 P'$
    return the sequence obtained by prepending the factor $u_1$ to the
            sequence Greedy-1-Factorize($P'$)
  endif
```

The correctness of the procedure Greedy-1-Factorize is shown in the following lemma:

Lemma 3.1. *The call* Greedy-1-Factorize(P)*, for a string* $P \in \Sigma^m$*, computes a minimal 1-factorization of* P*.*

Proof. Let $\boldsymbol{u} = \langle u_1, u_2, \ldots, u_k \rangle$ be the 1-factorization computed by the call Greedy-1-Factorize(P) and let $\boldsymbol{v} = \langle v_1, v_2, \ldots, v_h \rangle$ be any 1-factorization of P. We just need to show that $k \leq h$.

By construction, the character $\mathit{first}(u_{i+1})$ occurs in u_i, for $i = 1, 2, \ldots, k-1$, otherwise the factor u_i could have been extended by at least one more character.

We say that the factor v_j of $\boldsymbol{v}$ *covers* the factor u_i of $\boldsymbol{u}$ if $r_j^{\boldsymbol{v}} \leq r_i^{\boldsymbol{u}} < r_{j+1}^{\boldsymbol{v}}$ (see (3.1)), i.e., if j is the largest index such that the string $v_1 v_2 \ldots v_{j-1}$ is a prefix of the string $u_1 u_2 \ldots u_{i-1}$.

Plainly, each factor of $\boldsymbol{u}$ is covered by exactly one factor of $\boldsymbol{v}$. Thus, for our purposes, it is enough to show that each factor of $\boldsymbol{v}$ can cover at most one factor of $\boldsymbol{u}$, so that the number of factors in $\boldsymbol{v}$ must be at least as large as the number of factors in $\boldsymbol{u}$. Indeed, if this were not the case then

$$r_j^{\boldsymbol{v}} \leq r_i^{\boldsymbol{u}} < r_{i+1}^{\boldsymbol{u}} < r_{j+1}^{\boldsymbol{v}} \,,$$

for some $i \in \{1, 2, \ldots, h\}$ and $j \in \{1, 2, \ldots, k\}$. But then, the string $u_i.\mathit{first}(u_{i+1})$, which contains two occurrences of the character $\mathit{first}(u_{i+1})$, would be a factor of v_j, which yields a contradiction. □

A 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of a given pattern $P \in \Sigma^*$ induces naturally a partition $\{Q_1, \ldots, Q_k\}$ of the set $Q \setminus \{q_0\}$ of states of the automaton $\mathscr{A}(P) = (Q, \Sigma, \delta, q_0, F)$ for the language $\Sigma^* P$, where

$$Q_i \ =_{\text{Def}} \left\{q_{r_i+1}, \ldots, q_{r_{i+1}}\right\} \ , \ \text{for } i = 1, \ldots, k \,,$$

and $r_1, r_2, \ldots, r_{k+1}$ are the indices of $\langle u_1, u_2, \ldots, u_k \rangle$.

Notice that the labels of the arrows entering the states

$$q_{r_i+1}, \ldots, q_{r_{i+1}} \,,$$

in that order, form exactly the factor u_i, for $i = 1, \ldots, k$. Hence, if for any alphabet symbol a we denote by $Q_{i,a}$ the collection of states in Q_i with an incoming arrow labeled a, it follows that $|Q_{i,a}| \leq 1$ since, by condition (b) of the above definition of 1-factorization, no two states in Q_i can have an incoming transition labeled by the same character. When $Q_{i,a}$ is nonempty, we write $q_{i,a}$ to indicate the unique state q of $\mathscr{A}(P)$ for which $q \in Q_{i,a}$, otherwise $q_{i,a}$ is undefined. Upon using $q_{i,a}$ in any expression, we also implicitly assert that $q_{i,a}$ is defined.

For any valid configuration $\delta^*(q_0, Sa)$ of the automaton $\mathscr{A}(P)$ on some input of the form $Sa \in \Sigma^*$, we have that $q \in \delta^*(q_0, Sa)$ only if the state q has an incoming transition labeled a. Therefore, $Q_i \cap \delta^*(q_0, Sa) \subseteq Q_{i,a}$ and, consequently, $|Q_i \cap \delta^*(q_0, Sa)| \leq 1$, for each $i = 1, \ldots, k$. The configuration $\delta^*(q_0, Sa)$ can then be encoded by the pair (D, a), where D is the bit-vector of size k such that $D[i]$ is set iff Q_i contains an active state, i.e., $Q_i \cap \delta^*(q_0, Sa) \neq \emptyset$, iff $q_{i,a} \in \delta^*(q_0, Sa)$. Indeed, if $i_1, i_2, \ldots, i_l$ are all the indices i for which $D[i]$ is set, we have that $\delta^*(q_0, Sa) = \{q_{i_1,a}, q_{i_2,a}, \ldots, q_{i_l,a}\}$ holds, which shows that the above encoding (D, a) can be inverted.

To illustrate how to compute D' in a transition $(D, a) \xrightarrow{\mathscr{A}} (D', c)$ on character c using bit-parallelism, it is convenient to give some further definitions.

For $i = 1, \ldots, k-1$, we put $\overline{u}_i = u_i.\mathit{first}(u_{i+1})$. We also put $\overline{u}_k = u_k$ and call each set $\overline{u}_i$ the *closure* of u_i.

Plainly, any 2-gram can occur at most once in the closure $\overline{u}_i$ of any factor of our 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of P. We can then encode the 2-grams present in the closure of the factors u_i by a $\sigma \times \sigma$ matrix B of k-bit vectors, where the i-th bit of $B[c_1][c_2]$ is set iff the 2-gram $c_1 c_2$ is present in $\overline{u}_i$ or, equivalently, iff

$$\begin{aligned} &(\mathit{last}(u_i) \neq c_1 \land q_{i,c_2} \in \delta(q_{i,c_1}, c_2)) \lor \\ &\qquad (i < k \land \mathit{last}(u_i) = c_1 \land q_{i+1,c_2} \in \delta(q_{i,c_1}, c_2))\,, \end{aligned} \tag{3.2}$$

for every 2-gram $c_1 c_2 \in \Sigma^2$ and $i = 1, \ldots, k$.

To properly take care of transitions from the last state in Q_i to the first state in Q_{i+1}, it is also useful to have an array L, of size σ, of k-bit vectors encoding, for each character $c \in \Sigma$, the collection of factors ending with c. More precisely, the i-th bit of $L[c]$ is set iff $\mathit{last}(u_i) = c$, for $i = 1, \ldots, k$.

We show next that the matrix B and the array L, which in total require $(\sigma^2 + \sigma)k$ bits, are all is needed to compute the transition $(D, a) \xrightarrow{\mathscr{A}} (D', c)$ on character c. To this purpose, we first state the following basic property, which can easily be proved by induction.

Lemma 3.2 (Transition Lemma). *Let $(D, a) \xrightarrow{\mathscr{A}} (D', c)$, where (D, a) is the encoding of the configuration $\delta^*(q_0, Sa)$ for some string $S \in \Sigma^*$, so that (D', c) is the encoding of the configuration $\delta^*(q_0, Sac)$.*

Then, for each $i = 1, \ldots, k$, $q_{i,c} \in \delta^*(q_0, Sac)$ if and only if either

(i) $\mathit{last}(u_i) \neq a$, $q_{i,a} \in \delta^*(q_0, Sa)$, and $q_{i,c} \in \delta(q_{i,a}, c)$, or

(ii) $i \geq 1$, $\mathit{last}(u_{i-1}) = a$, $q_{i-1,a} \in \delta^*(q_0, Sa)$, and $q_{i,c} \in \delta(q_{i-1,a}, c)$. □

```
F-PREPROCESS$(P, m)$
    for $c \in \Sigma$ do $S[c] \leftarrow 0$
    for $c \in \Sigma$ do $L[c] \leftarrow 0$
    for $c, c' \in \Sigma$ do $B[c][c'] \leftarrow 0$
    $b \leftarrow 0$, $e \leftarrow 0$, $k \leftarrow 0$
    while $e < m$ do
        while $e < m$ and $S[P[e]] < k + 1$ do
            $S[P[e]] \leftarrow k + 1$, $e \leftarrow e + 1$
        for $i \leftarrow b + 1$ to $e - 1$ do
            $B[P[i-1]][P[i]] \leftarrow B[P[i-1]][P[i]] \mid (1 \ll k)$
        $L[P[e-1]] \leftarrow L[P[e-1]] \mid (1 \ll k)$
        if $e < m$ then
            $B[P[e-1]][P[e]] \leftarrow B[P[e-1]][P[e]] \mid (1 \ll k)$
        $b \leftarrow e$
        $k \leftarrow k + 1$
    return $(B, L, k)$
```

Figure 3.5: Preprocessing procedure for the construction of the arrays B and L relative to a minimal 1-factorization of the pattern.

Now observe that, by definition, the i-th bit of D' is set iff $q_{i,c} \in \delta^*(q_0, Sac)$ or, equivalently by the Transition Lemma and (3.2), iff (for $i = 1, \ldots, k$)

$$\begin{aligned}
&(D[i] = 1 \wedge B[a][c][i] = 1 \wedge \sim L[a][i] = 1) \vee \\
&\qquad (i \geq 1 \wedge D[i-1] = 1 \wedge B[a][c][i-1] = 1 \wedge L[a][i-1] = 1) && \text{iff} \\
&((D \mathbin{\&} B[a][c] \mathbin{\&} \sim L[a])[i] = 1 \vee (i \geq 1 \wedge (D \mathbin{\&} B[a][c] \mathbin{\&} L[a])[i-1] = 1)) && \text{iff} \\
&((D \mathbin{\&} B[a][c] \mathbin{\&} \sim L[a])[i] = 1 \vee ((D \mathbin{\&} B[a][c] \mathbin{\&} L[a]) \ll 1)[i] = 1) && \text{iff} \\
&((D \mathbin{\&} B[a][c] \mathbin{\&} \sim L[a]) \mid ((D \mathbin{\&} B[a][c] \mathbin{\&} L[a]) \ll 1))[i] = 1\,.
\end{aligned}$$

Hence $D' = (D \mathbin{\&} B[a][c] \mathbin{\&} \sim L[a]) \mid ((D \mathbin{\&} B[a][c] \mathbin{\&} L[a]) \ll 1)$, so that D' can be computed by the following bitwise operations:

$$\begin{aligned}
D &\leftarrow D \mathbin{\&} B[a][c] \\
H &\leftarrow D \mathbin{\&} L[a] \\
D &\leftarrow (D \mathbin{\&} \sim H) \mid (H \ll 1)\,.
\end{aligned}$$

To check whether the final state q_m belongs to a configuration encoded as (D, a), we have only to verify that $q_{k,a} = q_m$. This test can be broken into two steps: first, one checks if any of the states in Q_k is active, i.e., $D[k] = 1$; then, one verifies that the last character read is the last character of u_k, i.e., $L[a][k] = 1$.

The whole test can then be implemented with the bitwise test

$$D \,\&\, 10^{k-1} \,\&\, L[a] \neq 0^k\,.$$

The same considerations also hold for the suffix automaton $\mathscr{S}(P)$. The only difference is in the handling of the initial state. In the case of the automaton $\mathscr{A}(P)$, state q_0 is always active, so we have to activate state q_1 when the current text symbol is equal to $P[0]$. To do so it is enough to perform a bitwise or of D with $0^{k-1}1$ when $a = P[0]$, as $q_1 \in Q_1$. Instead, in the case of the suffix automaton $\mathscr{S}(P)$, as the initial state has an ε-transition to each state, all the bits in D must be set, as in the BNDM algorithm.

A drawback of the new encoding is that the handling of the self-loop and of the acceptance condition is more complex with respect to the original encoding. However, it is possible to simplify them at the expense of an overhead of at most two bits in the representation, by forcing the first and last factor in a 1-factorization to have length 1. Note that the handling of the self-loop is relevant for the automaton $\mathscr{A}(P)$ only, while the acceptance condition concerns both the $\mathscr{A}(P)$ and $\mathscr{S}(P)$ automata. In particular, to simplify the handling of the self-loop, we compute a factorization where the length of the first factor is equal to 1. Let $\langle v_1, v_2, \ldots, v_l\rangle$ be a minimal 1-factorization of $P[1\,..\,m-1]$; we define the following 1-factorization $\langle u_1, u_2, \ldots, u_{l+1}\rangle$ of P, where

$$u_i = \begin{cases} P[0] & \text{if } i = 1 \\ v_{i-1} & \text{if } 2 \leq i \leq l+1. \end{cases}$$

Observe that the size of Q_1 in the corresponding partition is 1; it follows that to handle the self-loop one does not need to perform the check $a = P[0]$ but just perform a bitwise or with $0^{k-1}1$, as there is one state only in the first subset and thus $q_{1,a}$ is undefined for all $a \neq P[0]$.

In a similar way, in order to simplify the acceptance condition, we compute a factorization where the length of the last factor is equal to 1; let $\langle v_1, v_2, \ldots, v_l\rangle$ be a minimal 1-factorization of $P[0\,..\,m-2]$; we define the following 1-factorization $\langle u'_1, u'_2, \ldots, u'_{l+1}\rangle$ of P where

$$u'_i = \begin{cases} v_i & \text{if } 1 \leq i \leq l \\ P[m-1] & \text{if } i = l+1. \end{cases}$$

```
F-Shift-And$(P, m, T, n)$
    $(B, L, k) \leftarrow$ F-PREPROCESS$(P, m)$
    $D \leftarrow 0^k$
    $a \leftarrow T[0]$
    for $j \leftarrow 1$ to $n - 1$
        if $a = P[0]$ then $D \leftarrow D \mid 0^{k-1}1$
        $D \leftarrow D \,\&\, B[a][T[j]]$
        $H \leftarrow D \,\&\, L[a]$
        $D \leftarrow (D \,\&\, \sim H) \mid (H \ll 1)$
        $a \leftarrow T[j]$
        if $(D \,\&\, 10^{k-1} \,\&\, L[a]) \neq 0^k$
            then Output$(j)$
```

```
F-BNDM$(P, m, T, n)$
    $(B, L, k) \leftarrow$ F-PREPROCESS$(P^r, m)$
    $j \leftarrow m - 1$
    while $j < n$ do
        $i \leftarrow 1, l \leftarrow 0$
        $D \leftarrow 1^k$, $a \leftarrow T[j]$
        while $D \neq 0^k$ do
            if $(D \,\&\, 10^{k-1} \,\&\, L[a]) \neq 0^k$ then
                if $i < m$ then
                    $l \leftarrow i$
                else Output$(j)$
            $D \leftarrow D \,\&\, B[a][T[j - i]]$
            $H \leftarrow D \,\&\, L[a]$
            $D \leftarrow (D \,\&\, \sim H) \mid (H \ll 1)$
            $a \leftarrow T[j - i]$
            $i \leftarrow i + 1$
        $j \leftarrow j + m - l$
```

Figure 3.6: Variants of Shift-And (**left**) and BNDM (**right**) based on the 1-factorization encoding.

In this case, the bitwise and with $L[a]$ in the acceptance condition test is not needed anymore, as there is only one state in the last subset.

Let k be the size of a minimal 1-factorization of P. Plainly, in both cases, $k - 1 \leq l \leq k$; in particular, if $l = k$, we have an overhead of 1 bit. If we combine the two techniques, the overhead in the representation is of at most two bits.

The preprocessing procedure which builds the arrays B and L described above and relative to a minimal 1-factorization of the given pattern $P \in \Sigma^m$ is reported in Fig. 3.5. Its time complexity is $\mathcal{O}(\sigma^2 + m)$. The variants of the Shift-And and BNDM algorithms based on our encoding of the configurations of the automata $\mathscr{A}(P)$ and $\mathscr{S}(P)$ are reported in Fig. 3.6 (algorithms F-Shift-And and F-BNDM, respectively). Their worst-case time complexities are $\mathcal{O}(n\lceil k/w \rceil)$ and $\mathcal{O}(nm\lceil k/w \rceil)$, respectively, while their space complexity is $\mathcal{O}(\sigma^2\lceil k/w \rceil)$, where k is the size of a minimal 1-factorization of the pattern.

3.3.1 q-grams based 1-factorization

It is possible to a achieve higher compactness by transforming the pattern into a sequence of overlapping q-grams and computing the 1-factorization of the resulting string. This technique has been extensively used to boost the performance of several string matching algorithms [36, 60]. More precisely, given a pattern P of length m defined over an alphabet Σ of size σ, the q-gram encoding of P is the string $P_0^{(q)} P_1^{(q)} \dots P_{m-q}^{(q)}$, defined over the alphabet Σ^q of the q-grams of Σ, where

$$P_i^{(q)} = P[i]P[i+1]\dots P[i+q-1]\,,$$

i.e., the pattern is transformed into the sequence of its $m-q+1$ overlapping substrings of length q, where each substring $P_i^{(q)}$ is regarded as a symbol belonging to Σ^q. Clearly, the size of the 1-factorization of the resulting string is at least $\lceil \frac{m-q+1}{\sigma^q} \rceil$. Hence, the size of the 1-factorization can be significantly reduced by using a q-grams representation. The only drawback is that the space needed for the tables B and L can grow up to $(\sigma^{2q} + \sigma^q)k$, where k is the size of the 1-factorization, if one follows a naive approach. Observe that, for each pair $\left(P_i^{(q)}, P_{i+1}^{(q)}\right)$, with $i = 0, \dots, m-q-1$, the corresponding transition must be encoded into the table B. However, as we are using overlapping q-grams, we have that $P_i^{(q)}[1\,..\,q-1] \sqsubset P_{i+1}^{(q)}$, i.e., the last $q-1$ symbols of $P_i^{(q)}$ are equal to the first $q-1$ symbols of $P_{i+1}^{(q)}$. Thus, there is no need to use the full q-gram $P_{i+1}^{(q)}$ as index in the table, rather we can use only its last symbol. More precisely, let $\langle u_1^q, u_2^q, \dots, u_k^q \rangle$ be a 1-factorization of the q-gram encoding of P; we encode the substrings of length $2q$ (or, equivalently, the 2-grams over Σ^q) present in the closure of the factors u_i^q by a $\Sigma^q \times \Sigma$ matrix B of k bit vectors, where the i-th bit of $B[C_1][c_2]$ is set iff the substring $C_1.C_1[1\,..\,q-1].c_2$ is present in $\overline{u}_i^q$, for every $C_1 \in \Sigma^q, c_2 \in \Sigma$. This method reduces the space complexity to $(\sigma^{q+1} + \sigma^q)k$. In general, however, it is not feasible to use a direct access table for the tables B and T with this encoding.

For values of σ^q still suitable for a direct access table, a useful technique is to lazily allocate only the rows of B that have at least one nonzero element. Let $g_q(P)$ be the number of distinct q-grams in P; then $g_q(P) \leq \min(\sigma^q, m-q+1)$. We can have at most $g_q(P) - 1$ nonzero rows in B (there is no transition starting from the last q-gram), which can be significantly less than σ^q. The resulting space complexity is then $\mathcal{O}(\sigma^q + (g_q(P)\sigma + \sigma^q)k)$.

An approach suitable to store the tables B and T, for arbitrary values of q, is to use a hash table, where the keys are the q-grams of the pattern. The main problem which arises when engineering a hash table is to choose a good hash function. Moreover, as we require that lookup in the searching phase be as fast as possible, the hash function must also be very efficient to compute. Since we are using overlapping q-grams, a given q-gram shares $q-1$ symbols with the previous one. To exploit this redundancy, we need a hash function that allows a recursive computation of the hash value of a generic q-gram Xb starting from the hash value of the previous q-gram aX, for $|X| \in \Sigma^{q-1}$ and $a, b \in \Sigma$. A method that satisfies such a requirement is hashing by integer division, which has been used in the Karp-Rabin string matching algorithm [42]. In this case, the hash function has the following definition

$$h(s_0, s_1, \ldots, s_{q-1}) = \sum_{j=0}^{q-1} r^{q-j-1} ord(s_j) \mod n\,, \tag{3.3}$$

where the radix r and n are parameters and $ord : \Sigma \to \mathbb{N}$ is a function that maps a symbol to a number in the range $0, \ldots, r-1$. For a given string s, the hash values of its overlapping q-grams can then be computed by using the following recursive definition

1. $h(s_0^q) = \sum_{j=0}^{q-1} r^{q-j-1} ord(s_j) \mod n$

2. $h(s_i^q) = (rh(s_{i-1}^q) + ord(s_{i+q-1}) - r^q ord(s_{i-1})) \mod n$

for $i = 1, \ldots, |s| - q$. The radix r is usually chosen in $\mathbb{Z}_n^*$ in such a way that the cycle length $\min\{k \mid r^k \equiv 1 \pmod{n}\}$ is maximal (if the cycle length is smaller than the length of the string, permutations of the same string could have the same hash value). As our domain is the set of strings of fixed length q, in this case it is enough to ensure that the cycle length is at least q. Another possibility would be to use hashing by cyclic polynomial, which is described in [26] together with an in-depth survey of recursive hashing functions for q-grams. The space complexity of this approach is $\mathcal{O}(g_q(P)((\sigma+1)k+q))$. If we handle collisions with chaining, the time complexity gets an additional multiplicative term equal to $(1+\alpha)q$, where α is the load factor. The q term in both the space and time complexities is due to the fact that, for each inserted q-gram, we have to store also the original string and, on searching, when an entry's hash matches, we have to compare the full q-gram. For small values of q, this overhead is negligible. If $q \log \sigma \in \mathcal{O}(w)$, where w is the word size in bits, the check can be performed in constant time by storing in the hash table, for each inserted q-gram $s_0 s_1, \ldots, s_{q-1}$, its signature $h(s_0, s_1, \ldots, s_{q-1})$, with $r = \sigma$ and $n = \sigma^q$, instead of the original string and then computing incrementally the signatures of the q-grams of the text as shown above.

3.3.2 Experimental evaluation

In this section we present and comment the experimental results relative to an extensive comparison of the BNDM and F-BNDM algorithms and of the Shift-And and F-Shift-And algorithms. In particular, in the BNDM case we have implemented two variants for each algorithm, named *single word* and *multiple words*, respectively. Single word variants are based on the automaton for a suitable substring of the pattern whose configurations can fit in a computer word; a naive check is then used to verify whether any occurrence of the subpattern can be extended to an occurrence of the complete pattern: specifically, in the case of the BNDM algorithm, the prefix pattern of length $\min(m, w)$ is chosen, while in the case of the F-BNDM algorithm the longest substring of the pattern which is a concatenation of at most w consecutive factors is selected. Multiple words variants are based on the automaton for the complete pattern whose configurations are splitted, if needed, over multiple machine words. The resulting implementations are referred to in the tables below as BNDM* and F-BNDM*.We also implemented versions of F-BNDM, named F-BNDM$_q$, that use the q-grams based 1-factorization, for $q \in \{2, 3, 4\}$. q-grams are indexed using a hash table of size 2^{12}, with collisions resolution by chaining; the hash function used is (3.3), with parameters $r = 131$ and $n = 2^w$.

We have also included in our tests the LBNDM algorithm [59]. When the alphabet is considerably large and the pattern length is at least two times the word size, the LBNDM algorithm achieves larger shift lengths. However, the time for its verification phase grows proportionally to m/w, so there is a treshold beyond which its performance degrades significantly.

For the Shift-And case, only test results relative to the multiple words variant have been included in the tables below, since the overhead due to a more complex bit-parallel simulation in the single word case is not paid off by the reduction of the number of calls to the verification phase.

The main two factors on which the efficiency of BNDM-like algorithms depends are the maximum shift length and the number of words needed for representing automata configurations. For the variants of the first case, the shift length can be at most the length of the longest substring of the pattern that fits in a computer word. This, for the BNDM algorithm, is plainly equal to $\min(w, m)$: hence, the word size is an upper bound for the shift length whereas, in the case of the F-BNDM algorithm, it is generally possible to achieve shifts of length larger

Algorithm	32	64	128	256	512	1024	2048	4096
LBNDM	2.44	1.46	1.13	0.77	0.62	0.55	1.94	10.81
BNDM	2.30	2.33	2.35	2.34	2.35	2.34	2.33	2.33
BNDM*	2.44	2.77	2.38	2.21	2.05	2.25	2.91	5.40
F-BNDM	2.44	1.62	1.27	1.03	1.03	0.99	0.98	0.93
F-BNDM*	2.53	1.64	1.23	1.72	1.49	1.50	1.60	2.44
F-BNDM$_2$	3.26	2.24	1.63	1.04	0.68	0.67	0.68	0.68
F-BNDM$_3$	2.49	1.61	1.26	0.84	0.59	0.55	**0.55**	**0.56**
F-BNDM$_4$	**2.29**	**1.37**	**1.05**	**0.70**	**0.53**	**0.51**	0.57	0.58
Shift-And*	8.85	51.45	98.42	142.27	264.21	508.71	997.19	1976.09
F-Shift-And*	22.20	22.20	22.21	92.58	147.79	213.70	354.32	662.06

Table 3.1: Running times (ms) on the King James version of the Bible ($\sigma = 63$).

Algorithm	32	64	128	256	512	1024	2048	4096
LBNDM	1.24	0.84	0.60	0.48	0.38	0.52	7.83	36.84
BNDM	1.34	1.35	1.37	1.37	1.36	1.34	1.36	1.36
BNDM*	1.39	1.48	1.22	1.22	1.12	1.25	1.80	4.18
F-BNDM	**1.19**	0.77	0.56	0.49	0.48	0.48	0.48	0.47
F-BNDM*	1.36	0.84	0.83	0.89	0.77	0.79	1.00	1.92
F-BNDM$_2$	1.42	0.96	0.67	0.48	0.35	**0.35**	**0.36**	**0.36**
F-BNDM$_3$	1.26	**0.71**	0.47	0.32	0.32	0.40	0.63	0.76
F-BNDM$_4$	1.36	0.71	**0.45**	**0.30**	**0.30**	0.40	0.73	2.92
Shift-And*	6.33	38.41	70.59	104.42	189.16	362.83	713.87	1413.76
F-Shift-And*	15.72	15.70	40.75	73.59	108.33	170.52	290.24	541.53

Table 3.2: Running times (ms) on a protein sequence of the *Saccharomyces cerevisiae* genome ($\sigma = 20$).

than w, as our encoding allows to pack more state configurations per bit on the average as shown in a table below. In the multi-word variants, the shift lengths for both algorithms, denoted BNDM* and F-BNDM*, are always equal, as they use the same automaton; however, the 1-factorization based encoding involves a smaller number of words on the average, especially for long patterns, thus providing a considerable speedup.

The tests have been performed on a 2.33 GHz Intel Core 2 Duo. We used the input files (i), (iii), (iv) (see Section 2.6).

For each input file, we have generated sets of 100 patterns of fixed length m randomly extracted from the text, for m ranging over the values 32, 64, 128, 256, 512, 1024, 2048, 4096. For each set of patterns we report the mean over the running times of 100 runs.

Concerning the BNDM-like algorithms, the experimental results show that in

Algorithm	32	64	128	256	512	1024	2048	4096
LBNDM	3.90	2.88	2.78	8.63	90.43	109.34	101.86	98.87
BNDM	**3.03**	3.02	3.02	3.01	3.01	3.02	3.02	3.03
BNDM*	3.25	4.58	4.23	3.89	3.42	3.56	4.27	6.72
F-BNDM	4.44	2.47	2.25	2.19	2.13	2.10	2.08	2.06
F-BNDM*	4.20	2.85	4.20	3.15	2.85	2.78	3.23	4.89
F-BNDM$_2$	7.17	4.40	2.97	2.64	2.59	2.58	2.59	2.60
F-BNDM$_3$	5.13	3.38	2.53	1.50	1.33	1.42	1.33	1.31
F-BNDM$_4$	3.29	**2.22**	**1.99**	**1.24**	**0.84**	**0.76**	**0.77**	**0.77**
Shift-And*	10.19	59.00	93.97	162.79	301.55	579.92	1131.50	2256.37
F-Shift-And*	25.04	42.93	114.22	167.11	281.37	460.37	839.32	1728.71

Table 3.3: Running times (ms) on a DNA sequence of the *Escherichia coli* genome ($\sigma = 4$).

(A)	ecoli	protein	bible
32	32	32	32
64	63	64	64
128	72	122	128
256	74	148	163
512	77	160	169
1024	79	168	173
1536	80	173	176
2048	80	174	178
4096	82	179	182

(B)	ecoli	protein	bible
32	15	8	6
64	29	14	12
128	59	31	26
256	119	60	50
512	236	116	102
1024	472	236	204
1536	705	355	304
2048	944	473	407
4096	1882	951	813

(C)	ecoli	protein	bible
32	2.13	4.00	5.33
64	2.20	4.57	5.33
128	2.16	4.12	4.92
256	2.15	4.26	5.12
512	2.16	4.41	5.01
1024	2.16	4.33	5.01
1536	2.17	4.32	5.05
2048	2.16	4.32	5.03
4096	2.17	4.30	5.03

Table 3.4: (A) The length of the longest substring of the pattern fitting in w bits; (B) the size of the minimal 1-factorization of the pattern; (C) the ratio between m and the size of the minimal 1-factorization of the pattern.

the case of long patterns both variants based on the 1-factorization encoding are considerably faster than their corresponding variants BNDM and BNDM*. In the first test suite, with $\sigma = 63$, the LBNDM algorithm turns out to be faster than F-BNDM, except for very long patterns, as the treshold on large alphabets is quite high. In the second test suite, with $\sigma = 20$, LBNDM is still competitive but, in the cases in which it beats the F-BNDM algorithm, the difference is small. In almost all the tests, the q-grams versions of F-BNDM achieve the best running times. It is worth observing that, in the case of file (ii), the number of distinct q-grams in the patterns is very high on average, and thus a small value of q yields more stable results. Instead, in the case of file (iii), where the alphabet size is small and thus also the number of distinct q-grams is small, the version with $q = 4$ is the fastest one. It turns out that F-BNDM$_4$ is the fastest algorithm also in the case of file (i) (natural language), where, despite the large alphabet size, the number of distinct q-grams in the patterns is small on average.

Likewise, the F-Shift-And variant is faster than the classical Shift-And algorithm in all cases, for $m \geq 64$.

3.4 Increasing the parallelism in bit-parallel algorithms

In this section we present two different approaches which lead to a higher level of parallelism in bit-parallel algorithms. By way of demonstration we apply them to a bit-parallel version of the Wide-Window algorithm,[1] but our approaches can be applied to other (more efficient) solutions based on bit-parallelism as well.

The two approaches can be summarized as follows:

- **first approach:** if the algorithm searches for the pattern in fixed-size text windows then, at each attempt, process simultaneously two (adjacent or partially overlapping) text windows by using in parallel two copies of the same automaton;

- **second approach:** if each search attempt of the algorithm can be divided into two steps (which possibly make use of two different automata) then execute simultaneously the two steps, by running the two automata in parallel.

[1] We chose the Wide-Window algorithm in our case study since its structure makes its parallelization simpler.

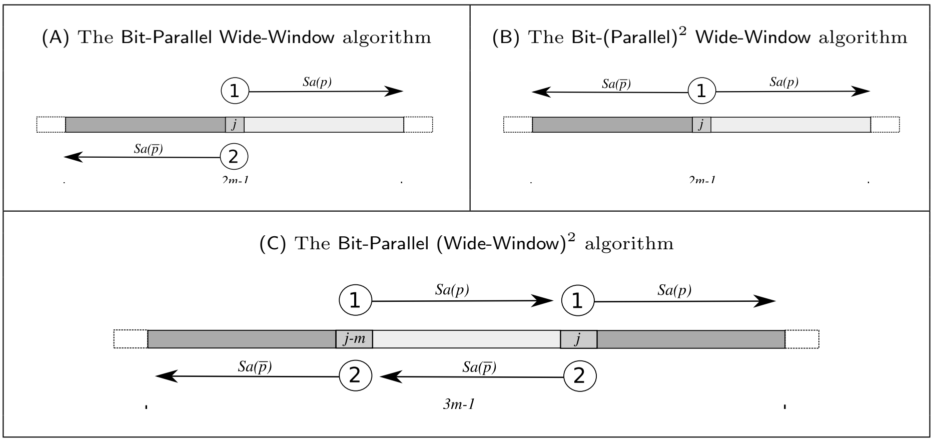

(A) The Bit-Parallel Wide-Window algorithm
1
Sa(p)
j
Sa($\overline{p}$)
2
2m-1
(B) The Bit-(Parallel)^2 Wide-Window algorithm
Sa($\overline{p}$)
1
Sa(p)
j
2m-1
(C) The Bit-Parallel (Wide-Window)^2 algorithm
1
Sa(p)
1
Sa(p)
j-m
j
Sa($\overline{p}$)
2
Sa($\overline{p}$)
2
3m-1

Figure 3.7: Structure of a searching iteration at a given position j in the text t of (A) the $\mathsf{B_pW_w}$ algorithm, (B) the $\mathsf{B_p^2W_w}$ algorithm, and (C) the $\mathsf{B_pW_w^2}$ algorithm.

Both variants use the SIMD (Single Instruction Multiple Data) paradigm. This approach, on which vectorial instructions sets like MMX and SSE are based, consists in executing the same instructions on multiple data in a parallel way. Tipically, a register of size w is logically divided into i blocks of k bits which are then updated simultaneously.

In both variants of the $\mathsf{B_pW_w}$ algorithm, we divide a word of w bits into *two* blocks, each being used to encode a suffix automaton. Thus, the maximum length of the pattern gets restricted to $\lfloor w/2 \rfloor$. We denote with B the array of bit masks encoding the suffix automaton $\mathscr{S}(p)$ and with C the array of bit masks encoding the suffix automaton $\mathscr{S}(p^r)$.

3.4.1 The Wide-Window Algorithm

The Wide-Window algorithm, (WW, for short) [37], is a recent algorithm for the string matching problem based on the suffix automaton. In the following description we slightly depart from its original version so as to make the algorithm parallelizable in ways that will be explained later. Let p be a pattern of length m and let t be a text of length n. The WW algorithm locates $\lfloor n/m \rfloor$ *attempt positions* in t, namely positions $j = km - 1$, for $k = 1, \ldots, \lfloor n/m \rfloor$. For each such position j, the pattern p is searched for in the *attempt window* of size $2m - 1$ centered at j, i.e., in the substring $t[j - m + 1 \mathinner{..} j + m - 1]$. Each of such search phases is divided into two steps.

In the first step, the right side of the attempt window, consisting of the last m characters, is scanned from left to right with the automaton $\mathscr{S}(p)$. In this step, the starting positions (in p) of the suffixes of p aligned with position j in t are collected in a set

$$\mathcal{S}_j = \{0 \leq i < m \mid p[i \mathinner{..} m-1] = t[j \mathinner{..} j+m-1-i]\}\,.$$

In the second step, the left half of the attempt window, consisting of the first m symbols, is scanned from right to left with the automaton $\mathscr{S}(p^r)$. During this step, the end positions (in p) of the prefixes of p aligned with position j in t are collected in a set

$$\mathcal{P}_j = \{0 \leq i < m \mid p[0 \mathinner{..} i] = t[j-i \mathinner{..} j]\}\,.$$

Taking advantage of the fact that an occurrence of p is located at position $(j-k)$ of t if and only if $k \in \mathcal{S}_j \cap \mathcal{P}_j$, for $k = 0, \ldots, m-1$, the number of all the occurrences of p in the attempt window centered at j is readily given by the cardinality $|\mathcal{S}_j \cap \mathcal{P}_j|$.

Figure 2(A) shows a simple schematization of the structure of an iteration of the WW algorithm at a given position j in t. The two sequential phases are represented by the arrows labeled 1 and 2, respectively.

It is straightforward to devise a bit-parallel implementation of the WW algorithm. The sets $\mathcal{P}$ and $\mathcal{S}$ can be encoded by two bit masks P and S, respectively. The nondeterministic automata $\mathscr{S}(p)$ and $\mathscr{S}(p^r)$ are then used for searching the suffixes and prefixes of p on the right and on the left parts of the window, respectively. Both automata state configurations and the final state configuration can be encoded by the bit masks D and $\mathsf{M} = (1 \ll (m-1))$, so that $(\mathsf{D}\ \&\ \mathsf{M}) \neq 0$ will mean that a suffix or a prefix of the search pattern p has been found, depending on whether D is encoding a state configuration of the automaton $\mathscr{S}(p)$ or of the automaton $\mathscr{S}(p^r)$. Whenever a suffix (resp., a prefix) of length $(\ell+1)$ is found (with $\ell = 0, 1, \ldots, m-1$), the bit $\mathsf{S}[m-1-\ell]$ (resp., the bit $\mathsf{P}[\ell]$) is set by one of the following bitwise operations:

$$\begin{array}{lcll} \mathsf{S} & \leftarrow & \mathsf{S} \mid ((\mathsf{D}\ \&\ \mathsf{M}) \gg \ell) & \text{(in the suffix case)} \\ \mathsf{P} & \leftarrow & \mathsf{P} \mid ((\mathsf{D}\ \&\ \mathsf{M}) \gg (m-1-\ell)) & \text{(in the prefix case)}\,. \end{array}$$

If we are only interested in counting the number of occurrences of p in t, we can just count the number of bits set in $(\mathsf{S}\ \&\ \mathsf{P})$. This can be done in $\log_2(w)$ operations by using a *population count* function, where w is the size of the computer word in bits (see [7]). Otherwise, if we also want to retrieve the matching positions of p in t, we can iterate over the bits set in $(\mathsf{S}\ \&\ \mathsf{P})$ by repeatedly computing the index of the highest bit set and then masking it.

The resulting algorithm based on bit-parallelism is named Bit-Parallel Wide-Window algorithm ($\mathsf{B_pW_w}$, for short). It needs $\lceil m/w \rceil$ words to represent the bit masks D, S, P, and $\mathsf{B}[c]$, for $c \in \Sigma$. The worst-case time complexity of the $\mathsf{B_pW_w}$ algorithm is $\mathcal{O}(n\lceil m/w \rceil + \lfloor n/m \rfloor \log_2(w))$.

Additionally, we observe that the $\mathsf{B_pW_w}$ algorithm can be easily modified so as to work on windows of size $2m$. For the sake of clarity, we have just discussed a simpler but slightly less efficient variant.

3.4.2 The Bit-Parallel (Wide-Window)^2 Algorithm

In the first variant, named Bit-Parallel (Wide-Window)^2 ($\mathsf{B_pW_w^2}$, for short), two partially overlapping windows in t, each of size $2m-1$, centered at consecutive attempt positions $j-m$ and j, are processed simultaneously. For the parallel simulation two automata are represented in a single word and updated in parallel.

Specifically, each search phase is divided again into two steps. During the first step, two copies of $\mathscr{S}(p)$ are operated in parallel to compute simultaneously the sets $\mathcal{S}_{j-m}$ and $\mathcal{S}_j$ (lines 13-18). Likewise, in the second step, two copies of $\mathscr{S}(p^r)$ are operated in parallel to compute the sets $\mathcal{P}_{j-m}$ and $\mathcal{P}_j$ (lines 20-25). To represent the automata with a single word, the bit masks D, M, S, and P are logically divided into two blocks, each of $k = w/2$ bits.

During the first step, the most significant k bits of D encode the state of the suffix automaton $\mathscr{S}(p)$ that scan the attempt window centered at $j-m$. Similarly, the least significant k bits of D encode the state of the suffix automaton $\mathscr{S}(p)$ that scans the attempt window centered at j. An analogous encoding is used in the second step, but with the automaton $\mathscr{S}(p^r)$ in place of $\mathscr{S}(p)$. Figure 2(C) schematizes the structure of a search iteration of the $\mathsf{B_pW_w^2}$ algorithm, at given attempt positions $j-m$ and j of t.

The most significant k bits of the bit mask S (resp., P) encode the set $\mathcal{S}_{j-m}$ (resp., $\mathcal{P}_{j-m}$), while the least significant k bits encode the set $\mathcal{S}_j$ (resp., $\mathcal{P}_j$). Thus, to properly detect suffixes in both windows, the bit mask M is initialized (lines 8-9) with the value

$$\mathsf{M} \leftarrow (1 \ll (m+k-1)) \mid (1 \ll (m-1))$$

and transitions of the automata are performed in parallel with the following bitwise operations (lines 14-15 and lines 21-22)

$$\begin{aligned} &\mathsf{D} \leftarrow (\mathsf{D} \ll 1) \,\&\, ((\mathsf{B}[t[j-m+\ell]] \ll k) \mid \mathsf{B}[t[j+\ell]]) && \text{(in the first phase)} \\ &\mathsf{D} \leftarrow (\mathsf{D} \ll 1) \,\&\, ((\mathsf{C}[t[j-m-\ell]] \ll k) \mid \mathsf{C}[t[j-\ell]]) && \text{(in the second phase)}, \end{aligned}$$

for $\ell = 1, \ldots, m-1$ (when $\ell = 0$, the left shift of D does not take place).

The remaining bitwise operations are left unchanged, as the automata configurations are updated using the same instructions. Since two windows are simultaneously scanned at each search iteration, the shift becomes $2m$, therefore doubling the length of the shift with respect to the WW algorithm. The pseudocode of the algorithm $\mathsf{B_pW_w^2}$ is reported in Fig. 3 (on the left).

3.4.3 The Bit-(Parallel)^2 Wide-Window Algorithm

The second variant of the WW algorithm which we present next is called Bit-(Parallel)^2 Wide-Window algorithm ($\mathsf{B_p^2W_w}$, for short). The idea behind it consists in processing a single window at each attempt (as in the original WW algorithm) but, in this case, by scanning its left and right sides simultaneously. Figure 2(B) schematizes the structure of a searching iteration of the $\mathsf{B_p^2W_w}$ algorithm, while Fig. 3 (on the right) shows the pseudocode of the $\mathsf{B_p^2W_w}$ algorithm.

As above, let p be a pattern of length m, and t be a text of length n. The bit masks B and C which are used to perform the transitions on both automata $\mathcal{S}(p)$ and $\mathcal{S}(p^r)$ are computed as in the $\mathsf{B_pW_w}$ algorithm (lines 3-7).

Automata state configurations are again encoded simultaneously in a bit mask D. Specifically, the most significant k bits of D encode the state of the suffix automaton $\mathcal{S}(p)$, while the least significant k bits of D encode the state of the suffix automaton $\mathcal{S}(p^r)$. The $\mathsf{B_p^2W_w}$ algorithm uses the following bitwise operations to perform transitions[2] of both automata in parallel (lines 14-15,17):

$$\mathsf{D} \leftarrow (\mathsf{D} \ll 1) \,\&\, ((\mathsf{B}[t[j+\ell]] \ll k) \mid \mathsf{C}[t[j-\ell]])\,,$$

for $\ell = 1, \ldots, m-1$. Note that in this case the left shift of k positions can be precomputed in B by setting $\mathsf{B}[c] \leftarrow \mathsf{B}[c] \ll k$, for each $c \in \Sigma$.

[2] For $\ell = 0$, D is simply updated by $\mathsf{D} \leftarrow \mathsf{D} \,\&\, ((\mathsf{B}[t[j+l]] \ll k) \mid \mathsf{C}[t[j-l]])$.

Using the same representation, the final-states bit mask M is initialized as

$$\mathsf{M} \leftarrow (1 \ll (m+k-1)) \mid (1 \ll (m-1)) \qquad \text{(lines 8-9)}\,.$$

At each iteration around an attempt position j of t, the sets $\mathcal{S}_j$ and $\mathcal{P}^*_j$ are computed, where $\mathcal{S}_j$ is defined as in the case of the $\mathsf{B_pW_w}$ algorithm, and $\mathcal{P}^*_j$ is defined as $\mathcal{P}^*_j = \{0 \leq i < m \mid p[0\,..\,m-1-i] = t[j-(m-1-i)\,..\,j]\}$, so that $\mathcal{P}_j = \{0 \leq i < m \mid (m-1-i) \in \mathcal{P}^*_j\}$.

The sets $\mathcal{S}_j$ and $\mathcal{P}^*_j$ can be encoded with a single bit mask PS, in the rightmost and leftmost k bits, respectively. Positions in $\mathcal{S}_j$ and $\mathcal{P}^*_j$ are then updated simultaneusly in PS by executing the following operation (line 16):

$$\mathsf{PS} \leftarrow \mathsf{PS} \mid ((\mathsf{D} \,\&\, \mathsf{M}) \gg l)\,.$$

At the end of each iteration, the bit masks S and P are retrieved from PS with the following bitwise operations (lines 19-20):

$$\mathsf{P} \leftarrow \mathsf{reverse}(\mathsf{PS}) \gg (w-m)\,, \qquad\qquad \mathsf{S} \leftarrow \mathsf{PS} \gg k\,,$$

where $\mathsf{reverse}$ denotes the bit-reversal function, which satisfies $\mathsf{reverse}(x)[i] = x[w-1-i]$, for $i = 0, \ldots, w-1$ and any bit mask x. In fact, to obtain the correct value of P we used bit-reversal modulo m, which has been easily achieved by right shifting $\mathsf{reverse}(\mathsf{PS})$ by $(w-m)$ positions. We recall that the $\mathsf{reverse}$ function can be implemented efficiently with $\mathcal{O}(\log_2(w))$ operations (see [7]).

3.4.4 Experimental evaluation

We present the results of an extensive experimental comparison of our proposed variants $\mathsf{B^2_pW_w}$ and $\mathsf{B_pW^2_w}$ with the $\mathsf{B_pW_w}$ and BNDM algorithms. In particular, we have tested two different implementations of the $\mathsf{B^2_pW_w}$ and $\mathsf{B_pW^2_w}$ algorithms, characterized by a different implementation of the *population-count* function. One implementation uses the built-in version of the $\mathsf{GNU\ C}$ compiler (algorithms $\mathsf{B^2_pW_w}$ and $\mathsf{B_pW^2_w}$), while the second implementation uses the population-count function described in [7] (algorithms $\mathsf{B^2_pW^{\underline{bc}}_w}$ and $\mathsf{B_pW^{2\ \underline{bc}}_w}$). We compared the following string matching algorithms, in terms of running times:

- the $\mathsf{Bit\text{-}Parallel\ Wide\text{-}Window}$ algorithm ($\mathsf{B_pW_w}$)
- the $\mathsf{Bit\text{-}(Parallel)^2\ Wide\text{-}Window}$ algorithm ($\mathsf{B^2_pW_w}$)
- the $\mathsf{Bit\text{-}(Parallel)^2\ Wide\text{-}Window}$ algorithm with $\mathsf{bit\text{-}count}$ ($\mathsf{B^2_pW^{\underline{bc}}_w}$)

```
Bit-Parallel (Wide-Window)$^2$ $(p, m, t, n)$
$count \leftarrow 0$
$k \leftarrow w/2$
for $c \in \Sigma$ do $\mathsf{B}[c] \leftarrow 0^m$
for $c \in \Sigma$ do $\mathsf{C}[c] \leftarrow 0^m$
for $i \leftarrow 0$ to $m-1$ do
    $\mathsf{B}[p[i]] \leftarrow \mathsf{B}[p[i]] \mid (0^{m-1}1 \ll i)$
    $c \leftarrow p[m-1-i]$
    $\mathsf{C}[c] \leftarrow \mathsf{C}[c] \mid (0^{m-1}1 \ll i)$
$\mathsf{H} \leftarrow 10^{m-1}$
$\mathsf{M} \leftarrow (\mathsf{H} \ll k) \mid \mathsf{H}$
$j \leftarrow 2m-1$
while $j < n-m$ do
    $\mathsf{D} \leftarrow 1^m, l \leftarrow 0, \mathsf{S} \leftarrow 0^m$
    while $\mathsf{D} \neq 0^m$ do
        $\mathsf{H} \leftarrow (\mathsf{B}[t[j-m+l]] \ll k) \mid \mathsf{B}[t[j+l]]$
        $\mathsf{D} \leftarrow \mathsf{D} \,\&\, \mathsf{H}$
        $\mathsf{S} \leftarrow \mathsf{S} \mid ((\mathsf{D} \,\&\, \mathsf{M}) \gg l)$
        $\mathsf{D} \leftarrow \mathsf{D} \ll 1$
        $l \leftarrow l+1$
    $\mathsf{D} \leftarrow 1^m, l \leftarrow 0, \mathsf{P} \leftarrow 0^m$
    while $D \neq 0^m$ do
        $\mathsf{H} \leftarrow (\mathsf{C}[t[j-m-l]] \ll k) \mid \mathsf{C}[t[j-l]]$
        $\mathsf{D} \leftarrow \mathsf{D} \,\&\, \mathsf{H}$
        $\mathsf{P} \leftarrow \mathsf{P} \mid ((\mathsf{D} \,\&\, \mathsf{M}) \gg (m-1-l))$
        $\mathsf{D} \leftarrow \mathsf{D} \ll 1$
        $l \leftarrow l+1$
    $count \leftarrow count + \mathsf{popcount}(\mathsf{P} \,\&\, \mathsf{S})$
    $j \leftarrow j+2m$
```

```
Bit-(Parallel)$^2$ Wide-Window $(p, m, t, n)$
$count \leftarrow 0$
$k \leftarrow w/2$
for $c \in \Sigma$ do $\mathsf{B}[c] \leftarrow 0^m$
for $c \in \Sigma$ do $\mathsf{C}[c] \leftarrow 0^m$
for $i \leftarrow 0$ to $m-1$ do
    $\mathsf{B}[p[i]] \leftarrow \mathsf{B}[p[i]] \mid (0^{m-1}1 \ll (k+i))$
    $c \leftarrow p[m-1-i]$
    $\mathsf{C}[c] \leftarrow \mathsf{C}[c] \mid (0^{m-1}1 \ll i)$
$\mathsf{H} \leftarrow 10^{m-1}$
$\mathsf{M} \leftarrow (H \ll k) \mid \mathsf{H}$
$j \leftarrow m-1$
while $j < n-m$ do
    $\mathsf{D} \leftarrow 1^m, l \leftarrow 0, \mathsf{PS} \leftarrow 0^m$
    while $\mathsf{D} \neq 0^m$ do
        $\mathsf{H} \leftarrow \mathsf{C}[t[j-l]] \mid \mathsf{B}[t[j+l]]$
        $\mathsf{D} \leftarrow \mathsf{D} \,\&\, \mathsf{H}$
        $\mathsf{PS} \leftarrow \mathsf{PS} \mid ((\mathsf{D} \,\&\, \mathsf{M}) \gg l)$
        $\mathsf{D} \leftarrow \mathsf{D} \ll 1$
        $l \leftarrow l+1$
    $\mathsf{P} \leftarrow \mathsf{reverse}(\mathsf{PS}) \gg (w-m)$
    $\mathsf{S} \leftarrow \mathsf{PS} \gg k$
    $count \leftarrow count + \mathsf{popcount}(\mathsf{P} \,\&\, \mathsf{S})$
    $j \leftarrow j+m$
```

Figure 3.8: The Bit-Parallel (Wide-Window)^2 algorithm (**left**) and the Bit-(Parallel)^2 Wide-Window algorithm (**right**) for the string matching problem.

- the Bit-Parallel (Wide-Window)^2 algorithm ($\mathsf{B_pW_w^2}$)
- the Bit-Parallel (Wide-Window)^2 algorithm with bit-count ($\mathsf{B_pW_w^{2\ bc}}$)
- the Backward-Nondeterministic-DAWG-Matching algorithm (BNDM).

The tests have been performed on a 1.66 GHz Intel Core 2 Duo. In particular, all algorithms have been tested on seven Randσ problems, for $\sigma = 2, 4, 8, 16, 32, 64, 128$, where a Rand$\sigma$ problem consists of searching a set of 400 random patterns of a given length in a 5Mb random text over a common alphabet of size σ, with a uniform character distribution.

Only short patterns of length $m = 2, 4, 6, 8, 10, 12, 14, 16$ have been considered in our tests, since the bit size of a word was, in our case, 32. However, the same approach could be applied with 64-bit processors or using Intel Processors with SSE instructions on 128 bit registers, to process patterns up to lengths of 32 and of 64, respectively. Moreover, we observe that $\lceil 2m/w \rceil$ different words could be used for representing our suffix automata for longer patterns, overcoming the

Algorithm	2	4	6	8	10	12	14	16
B_pW_w	81.65	63.40	52.12	44.25	38.15	33.42	30.07	27.10
$B_p^2W_w$	77.50	48.15	**33.22**	**25.55**	**21.17**	**18.30**	**16.22**	**14.60**
$B_p^2W_w^{\underline{bc}}$	**64.22**	**45.55**	35.35	28.75	24.30	21.07	18.65	16.72
$B_pW_w^2$	90.57	70.07	55.45	45.52	38.30	33.15	29.17	26.85
$B_pW_w^{2\,\underline{bc}}$	65.97	56.67	47.85	40.45	34.92	30.67	27.45	25.65
BNDM	69.07	54.57	41.05	30.80	24.45	20.15	17.20	15.00

Table 3.5: Running times for a Rand2 problem.

Algorithm	2	4	6	8	10	12	14	16
B_pW_w	61.57	38.12	27.80	22.15	18.55	15.92	14.02	12.55
$B_p^2W_w$	52.70	**31.17**	**23.27**	**18.67**	**15.57**	**13.40**	**11.82**	**10.62**
$B_p^2W_w^{\underline{bc}}$	**50.35**	34.27	26.00	20.75	17.27	14.82	13.05	11.65
$B_pW_w^2$	64.50	38.62	27.92	21.95	17.97	15.22	13.27	11.92
$B_pW_w^{2\,\underline{bc}}$	55.77	37.55	27.62	21.72	18.00	15.40	13.55	12.22
BNDM	55.55	31.20	**21.57**	**16.62**	**13.70**	**11.70**	**10.20**	**9.05**

Table 3.6: Running times for a Rand4 problem.

bound on the value of m though at the price of an increased running time.

In the following tables, running times are expressed in milliseconds. The best results among all bit-parallel WW variants have been boldfaced and underlined. Additionally, running times relative to the BNDM algorithm have been boldfaced and underlined when the BNDM algorithm outperforms the other algorithms.

The experimental results show that the algorithms obtained by applying a second level of parallelism perform always better then the original B_pW_w algorithm. The gap is more evident in the case of short patterns or small alphabets.

Algorithm	2	4	6	8	10	12	14	16
B_pW_w	42.27	27.35	19.97	15.37	12.37	10.35	8.92	7.90
$B_p^2W_w$	**37.57**	**23.45**	17.32	13.75	11.52	10.05	8.95	8.07
$B_p^2W_w^{\underline{bc}}$	37.82	25.22	18.60	14.75	12.32	10.72	9.52	8.60
$B_pW_w^2$	40.77	23.90	**16.47**	**12.62**	**10.32**	**8.90**	**7.82**	**7.07**
$B_pW_w^{2\,\underline{bc}}$	39.85	24.92	17.22	13.10	10.75	9.20	8.15	7.32
BNDM	**36.92**	23.65	16.77	**12.45**	**9.92**	**8.12**	**6.97**	**6.10**

Table 3.7: Running times for a Rand8 problem.

Algorithm	2	4	6	8	10	12	14	16
$\mathsf{B_pW_w}$	35.37	20.40	15.47	12.75	10.82	9.27	8.07	6.95
$\mathsf{B_p^2\,W_w}$	25.82	18.50	14.50	11.42	9.40	7.87	6.77	5.90
$\mathsf{B_p^2\,W_w^{\underline{bc}}}$	**26.50**	19.35	15.15	11.85	9.82	8.20	7.10	6.20
$\mathsf{B_pW_w^2}$	27.85	**17.35**	**12.90**	**9.92**	**7.97**	**6.57**	**5.62**	**4.85**
$\mathsf{B_pW_w^{2\,\underline{bc}}}$	28.27	18.12	13.42	10.32	8.27	6.82	5.80	5.02
BNDM	**26.50**	**169.0**	13.20	10.85	9.10	7.70	6.60	5.72

Table 3.8: Running times for a Rand16 problem.

Algorithm	2	4	6	8	10	12	14	16
$\mathsf{B_pW_w}$	27.37	16.07	11.92	9.82	8.50	7.60	6.85	6.27
$\mathsf{B_p^2\,W_w}$	**19.15**	13.65	11.47	9.57	8.15	7.15	6.30	5.60
$\mathsf{B_p^2\,W_w^{\underline{bc}}}$	19.70	14.05	11.90	9.80	8.45	7.35	6.42	5.67
$\mathsf{B_pW_w^2}$	21.50	**12.90**	**9.87**	**8.20**	**7.05**	**6.10**	**5.35**	**4.62**
$\mathsf{B_pW_w^{2\,\underline{bc}}}$	21.95	13.25	10.12	8.32	7.20	6.25	5.47	4.75
BNDM	21.87	**12.82**	**9.80**	**8.20**	7.20	6.45	5.82	5.27

Table 3.9: Running times for a Rand32 problem.

Algorithm	2	4	6	8	10	12	14	16
$\mathsf{B_pW_w}$	25.00	13.67	9.85	7.95	6.75	5.95	5.37	4.92
$\mathsf{B_p^2\,W_w}$	**16.00**	10.70	8.90	7.50	6.52	5.97	5.40	5.02
$\mathsf{B_p^2\,W_w^{\underline{bc}}}$	16.27	11.15	9.15	7.50	6.75	6.05	5.42	5.00
$\mathsf{B_pW_w^2}$	18.42	**10.42**	**7.70**	**6.32**	**5.52**	**4.95**	**4.50**	**4.12**
$\mathsf{B_pW_w^{2\,\underline{bc}}}$	18.60	10.57	7.82	6.42	5.57	4.97	4.55	4.17
BNDM	19.75	10.90	7.92	6.42	**5.52**	**4.95**	**4.50**	4.17

Table 3.10: Running times for a Rand64 problem.

Algorithm	2	4	6	8	10	12	14	16
$\mathsf{B_pW_w}$	23.85	12.62	8.82	6.92	5.80	5.00	4.45	4.02
$\mathsf{B_p^2\,W_w}$	**14.57**	9.27	7.35	6.15	5.30	4.90	4.35	4.15
$\mathsf{B_p^2\,W_w^{\underline{bc}}}$	14.80	9.60	7.60	6.15	5.47	4.90	4.30	4.10
$\mathsf{B_pW_w^2}$	16.82	**9.12**	**6.52**	**5.25**	**4.42**	**3.92**	**3.60**	**3.30**
$\mathsf{B_pW_w^{2\,\underline{bc}}}$	16.90	9.20	6.57	**5.25**	4.50	3.97	**3.60**	**3.30**
BNDM	18.72	9.95	7.02	5.55	4.67	4.07	3.65	3.35

Table 3.11: Running times for a Rand128 problem.

In particular the $\mathsf{B}_{\mathsf{p}}^2\mathsf{W}_{\mathsf{w}}$ algorithm achieves its best performances with small alphabets, while the $\mathsf{B}_{\mathsf{p}}\mathsf{W}_{\mathsf{w}}^2$ algorithm turns out to be the best choice for patterns with a length greater than 4. The BNDM algorithm obtains the best results in some cases and performs always better than the $\mathsf{B}_{\mathsf{p}}\mathsf{W}_{\mathsf{w}}$ algorithm. It is interesting to observe that the BNDM algorithm is slower than the $\mathsf{B}_{\mathsf{p}}^2\mathsf{W}_{\mathsf{w}}$ algorithm when the alphabet is small and by the $\mathsf{B}_{\mathsf{p}}\mathsf{W}_{\mathsf{w}}^2$ algorithm in the case of large alphabets.

Chapter 4

The multiple string matching problem

Given a set $\mathcal{P}$ of r patterns and a text T of length n, all strings over a common finite alphabet Σ of size σ, the *multiple string matching* problem consists in finding all the occurrences in T of the patterns in $\mathcal{P}$. The optimal average complexity of the problem is $\mathcal{O}(n \log_\sigma(rl_{min})/l_{min})$ [54], where l_{min} is the length of the shortest pattern in the set $\mathcal{P}$.

The first linear solution of the multiple string matching problem based on finite automata is due to Aho and Corasick [1]. The Aho-Corasick algorithm uses a deterministic incomplete finite automaton based on the *trie* for the input patterns and on the *failure function*, a generalization of the border function of the Knuth-Morris-Pratt algorithm [47]. The lower bound on average has been achieved by algorithms based on the suffix automaton induced from the DAWG data structure, namely the MultiBDM [29] and the SBDM algorithms [57], which are generalizations of the BDM [30] algorithm to the multiple pattern case.

In this chapter we focus on automata based solutions of this problem and, in particular, on the efficient simulation of the nondeterministic automaton for the language $\bigcup_{P\in\mathcal{P}} \Sigma^* P$ induced from the *trie* data structure for $\mathcal{P}$ (Aho-Corasick NFA, for short) and for the nondeterministic automaton relative to the language $\bigcup_{P\in\mathcal{P}} \mathit{Suff}(P)$ of all the suffixes of the strings in $\mathcal{P}$ induced from the DAWG data structure for $\mathcal{P}$ (suffix NFA, for short).

In the previous chapter we have introduced the bit-parallelism technique [8] to simulate efficiently simple nondeterministic finite automata for the single pattern case. In particular, we have seen that, to represent an automaton configuration as a bit-vector, the states of the automaton must be mapped onto the positions of a bit-vector by a suitable topological ordering of the NFA.[1] In the case of a single pattern, the construction of the topological ordering is quite simple since it is unique.

Appropriate topological orderings can be obtained also for the maximal trie of a set of patterns, by interleaving the tries of the single patterns in either a parallel fashion, under the constraint that all the patterns have the same length [68], or in a sequential fashion [56]. The Shift-And and BNDM algorithms can be easily extended to the multiple patterns case by deriving the corresponding automaton from the maximal trie of the set of patterns. The resulting algorithms have a $\mathcal{O}(\sigma\lceil size(\mathcal{P})/w\rceil)$-space complexity and work in $\mathcal{O}(n\lceil size(\mathcal{P})/w\rceil)$ and $\mathcal{O}(n\lceil size(\mathcal{P})/w\rceil l_{min})$ worst-case searching time complexity, respectively, where $size(\mathcal{P}) = \sum_{P\in\mathcal{P}} |P|$ is the sum of the lengths of the strings in $\mathcal{P}$ and w is the size of a computer word.

In both cases the bit-parallel simulation is based on the following property of the topological ordering π associated to the trie which allows to encode the transitions using a shift of k bits and a bitwise and: for each edge (p, q) the distance $\pi(q) - \pi(p)$ is equal to a constant k. For an in-depth survey on this topic the reader is referred to [19].

The problem which arises when trying to bit-parallel simulate the Aho-Corasick NFA and the suffix NFA is that, in general, there might be no topological ordering π such that, for each edge (p, q), the distance $\pi(q) - \pi(p)$ is fixed. Cantone and Faro [19] presented a bit-parallel simulation of the Aho-Corasick NFA that encodes variable length shifts using the carry property of addition and based on a particular topological ordering; however, such topological orderings do not always exist. Their algorithm has a $\mathcal{O}(\sigma\lceil m/w\rceil)$-space and $\mathcal{O}(n\lceil m/w\rceil)$-searching time complexity, where m is the number of nodes in the trie.

In this chapter we present an efficient bit-parallel simulation of the Aho-Corasick NFAs and suffix NFAs. Our construction is based on Lemma 3.1 presented in Section 3.2. We show that, by exploiting the relation between active states of the NFA and its associated failure function, it is possible to represent the $\Phi(\cdot)$ map in polynomial space.

[1] We recall that a *topological ordering* of an NFA is any total ordering $<$ of the set of its states such that $p < q$, for each edge (p, q) of the NFA.

```
Multiple-Shift-And ($T$, $\mathcal{P} = \{P_1, \dots, P_r\}$)
$n \leftarrow |T|$
$L \leftarrow size(\mathcal{P})$
for $c \in \Sigma$ do $B[c] \leftarrow 0^L$
$j \leftarrow 0$
$I \leftarrow M \leftarrow 0^L$
for $k \leftarrow 1$ to $r$ do
    for $i \leftarrow 0$ to $|P_k| - 1$ do
        $B[P_k[i]] \leftarrow B[P_k[i]] \mid (0^{L-1}1 \ll (j + i))$
    $I \leftarrow I \mid (0^{L-1}1 \ll j)$
    $M \leftarrow M \mid (0^{L-1}1 \ll (j + |P_k| - 1))$
    $j \leftarrow j + |P_k|$
$D \leftarrow 0^L$
for $j \leftarrow 0$ to $n - 1$ do
    $D \leftarrow ((D \ll 1) \mid I)$ & $B[T[j]]$
    if $D$ & $M \neq 0^L$ then Output($j$)
```

Figure 4.1: The Multiple-Shift-And algorithm.

4.1 Bit-parallelism for multiple string matching

The existing variants of the Shift-And and BNDM algorithms that search for a set $\mathcal{P} = \{P_1, \dots, P_r\}$ of patterns, using bit-parallelism, are based on the *maximal trie* of $\mathcal{P}$. The number of states of $\mathcal{T}^{max}_{\mathcal{P}}$ is given by $|Q_{\mathcal{T}^{max}_{\mathcal{P}}}| = \sum_{i=1}^{r} |Q_{\mathcal{T}_{\mathcal{P}_i}}| - r + 1 = size(\mathcal{P}) + 1$, so that it can be represented by a bit-vector of $L = size(\mathcal{P})$ bits. The states of $\mathcal{T}^{max}_{\mathcal{P}}$ are mapped onto positions in the bit-vectors through a bijection $\pi : Q_{\mathcal{T}^{max}_{\mathcal{P}}} \rightarrow \{0, 1, \dots, |Q_{\mathcal{T}^{max}_{\mathcal{P}}}| - 1\}$. The arrangement consists in concatenating the different branches of the maximal trie of $\mathcal{P}$ in a sequential fashion. More precisely, for each pattern P_i, the positions in π of the states $\{q^i_1, q^i_2, \dots, q^i_{|P_i|}\}$ of the trie $\mathcal{T}_{\mathcal{P}_i}$ are defined as

$$\pi(q^i_k) = \sum_{j=0}^{i-1} |P_j| + k$$

for $k = 1, \dots, |P_i|$. As we have r final states, the final-state bit mask is defined as

$$M \quad = \quad (10^{|P_1|-1})(10^{|P_2|-1}) \cdots (10^{|P_r|-1}) .$$

The automaton for the language $\Sigma^*\mathcal{P}$ can be easily obtained from the maximal trie of $\mathcal{P}$ by adding a self-loop on Σ on the initial state. The variant of Shift-And for the *multiple string matching* problem, named Multiple-Shift-And, has $\mathcal{O}(n\lceil size(\mathcal{P})/w\rceil)$ worst-case searching time complexity and $\mathcal{O}(\sigma\lceil size(\mathcal{P})/w\rceil)$ space complexity. In this case the initial state of the automaton can activate the r states corresponding to the first symbol of each pattern; correspondingly, the bitwise or with $0^{L-1}1$ must be changed to a bitwise or with the following bit mask

$$I \quad = \quad (0^{|P_1|-1}1)(0^{|P_2|-1}1)\cdots(0^{|P_r|-1}1)\,.$$

The pseudocode of the Multiple-Shift-And algorithm is reported in Fig. 4.1.

Likewise, the suffix automaton for the language $\mathit{Suff}(\mathcal{P})$ can be obtained from the maximal trie by adding an ϵ-transition from the initial state to every other state. The variant of BNDM for the *multiple string matching* problem, named Multiple-BNDM, has $\mathcal{O}(n\lceil size(\mathcal{P})/w\rceil l_{min})$ worst-case searching time complexity and $\mathcal{O}(\sigma\lceil size(\mathcal{P})/w\rceil)$-space complexity. It uses the maximal trie $\mathcal{T}_{\mathcal{P}^r_{l_{min}}}$ built on the prefixes of the reversed patterns of length l_{min} to maximize the shift, where l_{min} is the length of the shortest pattern. The pseudocode of the Multiple-BNDM algorithm is reported in Fig. 4.2; for the sake simplicity in the pseudocode it is assumed that all the patterns have the same length l.

4.2 The **Aho-Corasick** NFA

The Aho-Corasick NFA for a set $\mathcal{P}$ of patterns over an alphabet Σ is induced directly by the trie $\mathcal{T}_{\mathcal{P}}$ for $\mathcal{P}$. More precisely, it is the NFA $A_{\mathcal{P}} = (Q, \Sigma, \delta_A, q_0, F)$, where:

- Q is the set of nodes of $\mathcal{T}_{\mathcal{P}}$ (*the set of states*);
- $q_0 \in Q$ is the root of $\mathcal{T}_{\mathcal{P}}$ (*the initial state*);
- $\delta_A : Q \times \Sigma \rightarrow \mathscr{P}(Q)$ is the *transition function*, with

$$\delta_A(q, c) =_{Def} \begin{cases} \{p \in Q \mid lbl(p) = c\} \cup \{q_0\} & \text{if } q = q_0 \\ \{p \in Q \mid lbl(p) = lbl(q).c\} & \text{if } q \neq q_0\,, \end{cases}$$

  for $q \in Q$, $c \in \Sigma$, and where we recall that $\mathscr{P}(\cdot)$ denotes the powerset operator;

- $F =_{\text{Def}} \{q \in Q \mid lbl(q) \in \mathcal{P}\}$ is the set of *final* states.

Plainly, we have $|Q| \leq \sum_{P\in\mathcal{P}} |P|$.

```
Multiple-BNDM ($T$, $\mathcal{P} = \{P_1, \dots, P_r\}$)
$n \leftarrow |T|$
$L \leftarrow size(\mathcal{P})$
for $c \in \Sigma$ do $B[c] \leftarrow 0^L$
$j \leftarrow 0$
$C \leftarrow M \leftarrow 0^L$
for $k \leftarrow 1$ to $r$ do
    for $i \leftarrow 0$ to $l - 1$ do
        $c \leftarrow P_k[l - 1 - i]$
        $B[c] \leftarrow B[c] \mid (0^{L-1}1 \ll (j + i))$
    $C \leftarrow C \mid (0^{L-l}1^{l-1}0 \ll j)$
    $M \leftarrow M \mid (0^{L-1}1 \ll (j + l - 1))$
    $j \leftarrow j + l$
$j \leftarrow l - 1$
while $j < n$ do
    $k \leftarrow 0, last \leftarrow 0$
    $D \leftarrow 1^L$
    while $D \neq 0^L$ do
        $D \leftarrow D \,\&\, B[T[j - k]]$
        $k \leftarrow k + 1$
        if $D \,\&\, M \neq 0^L$ then
            if $k < l$ then
                $last \leftarrow k$
            else Output($j$)
        $D \leftarrow (D \ll 1) \,\&\, C$
    $j \leftarrow j + l - last$
```

Figure 4.2: The Multiple-BNDM algorithm.

We also associate with the NFA $A_{\mathcal{P}}$ a failure function $fail : Q \setminus \{q_0\} \rightarrow Q$ such that

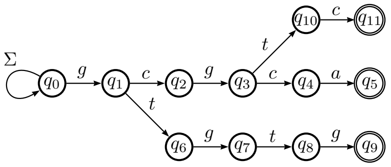
- $lbl(fail(q)) \sqsupset lbl(q)$, and
- $len(fail(q)) \geq len(p)$, for each $p \in Q$ such that $lbl(p) \sqsupset lbl(q)$

In other words, $lbl(fail(q))$ is the longest proper suffix of $lbl(q)$ which is also a prefix of a string in $\mathcal{P}$.

The automaton $A_{\mathcal{P}}$ can be seen as the nondeterministic version of the Aho-Corasick automaton: this is a trie $\mathcal{T}_{\mathcal{P}}$ for a set of patterns $\mathcal{P}$ augmented with *failure* links, which are followed when no transition is possible on a text character (cf. [1]).

An immediate, yet useful, property of the Aho-Corasick NFA, which can be readily proved by induction, is the following

$$q_0 \in \delta_A^*(q_0, u), \text{ for every } u \in \Sigma^*. \tag{4.1}$$



q	*fail*
q_0	
q_1	q_0
q_2	q_0
q_3	q_1
q_4	q_2
q_5	q_0
q_6	q_0
q_7	q_1
q_8	q_6
q_9	q_7
q_{10}	q_6
q_{11}	q_0

Figure 4.3: Aho-Corasick NFA for the set of strings {**gcgca**,**gtgtg**,**gcgtc**}.

The Aho-Corasick NFA $A_{\mathcal{P}} = (Q, \Sigma, \delta_A, q_0, F)$ relative to a given set $\mathcal{P}$ of patterns can be used to find the occurrences of the patterns of $\mathcal{P}$ in a given text T, by observing that a pattern $P \in \mathcal{P}$ has an occurrence in T ending at position i, i.e., $P \sqsupseteq T_i$, if and only if $\delta_A^*(q_0, T[0 \,..\, i])$ contains a final state $q \in F$ such that $lbl(q) = P$. Thus, to find all the occurrences in T of the patterns of $\mathcal{P}$, it suffices to compute the set $\delta_A^*(q_0, T_i) \cap F$, for $i = 0, 1, \ldots, |T| - 1$. As an immediate consequence of (2.1) and the definitions of δ_A and δ_A^* on $\mathscr{P}(Q)$, we have $\delta_A^*(q_0, T_i) = \delta_A(\delta_A^*(q_0, T_{i-1}), T[i])$, for $i = 1, 2, \ldots, |T| - 1$. Hence, the problem of computing efficiently the sets $\delta_A^*(q_0, T_i)$ can be reduced to the problem of evaluating efficiently transition actions of the form $\delta_A(D, c)$, for any $c \in \Sigma$ and any *reachable configuration* $D \subseteq Q$ of $A_{\mathcal{P}}$.

The following property is an immediate consequence of the definition of the failure function.

Lemma 4.1. *Given the Aho-Corasick NFA $A_{\mathcal{P}} = (Q, \Sigma, \delta_A, q_0, F)$ for a set $\mathcal{P}$ of patterns and its associated failure function $fail : Q \setminus \{q_0\} \to Q$, we have*

$$lbl(p) \sqsupset lbl(q) \to lbl(p) \sqsupseteq lbl(fail(q)),$$

for all $p \in Q$ and $q \in Q \setminus \{q_0\}$. □

4.3 The suffix NFA

The suffix NFA for a finite set $\mathcal{P}$ of patterns over an alphabet Σ is the NFA with ε-transitions $S_{\mathcal{P}} = (Q, \Sigma, \delta_S, q_0, F)$ induced by the DAWG for $\mathcal{P}$, where

- $Q =_{\text{Def}} \{R_{\mathcal{P}}(u) \mid u \in \mathit{Fact}(\mathcal{P})\}$ is the set of states;[2]
- $q_0 = R_{\mathcal{P}}(\varepsilon)$ is the initial state;
- $\delta_S : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathscr{P}(Q)$ is the transition function defined by:

$$\delta_S(R_{\mathcal{P}}(u), a) =_{\text{Def}} \begin{cases} Q & \text{if } ua = \varepsilon \\ \{R_{\mathcal{P}}(ua)\} & \text{if } ua \in \mathit{Fact}(\mathcal{P}) \setminus \{\varepsilon\} \\ \emptyset & \text{otherwise;} \end{cases}$$

- $F = \{q \in Q \mid \mathit{val}(q) \in \mathit{Suff}(\mathcal{P})\}$ is the set of final states.

It is well-known that the language recognized by the NFA $S_{\mathcal{P}}$ is $\mathit{Suff}(\mathcal{P})$. Additionally, the NFA $S'_{\mathcal{P}} = (Q, \Sigma, \delta_S, q_0, Q)$, obtained from $S_{\mathcal{P}}$ by considering all states in Q as final, recognizes the language $\mathit{Fact}(\mathcal{P})$ of all factors of strings in $\mathcal{P}$. In other words, for $u \in \Sigma^*$, we have

$$\delta_S(q_0, u) \neq \emptyset \quad \text{if and only if} \quad u \in \mathit{Fact}(\mathcal{P}) . \tag{4.2}$$

We also observe that if $\mathit{size}(\mathcal{P}) > 1$, then $|Q| \leq 2\sum_{P \in \mathcal{P}} |P| - 1$ (cf. [12]).

A useful property of the function $\mathit{suf}(\cdot)$ is proved in the following lemma.

Lemma 4.2. *Given a nondeterministic suffix automaton $S_{\mathcal{P}} = (Q, \Sigma, \delta_S, q_0, F)$ for a set of patterns $\mathcal{P}$, for all $p, q \in Q$ we have*

(a) if $\mathit{val}(p) \sqsupset \mathit{val}(q)$ then $\mathit{val}(p) \sqsupseteq \mathit{val}(\mathit{suf}(q)) \sqsupset \mathit{val}(q)$;

(b) if $\mathit{val}(p) \sqsupset \mathit{val}(q)$ then $\mathit{suf}^{(k)}(q) = p$, for some $k \geq 1$ (where $\mathit{suf}^{(0)}(q) =_{\mathit{Def}} q$ and, recursively, $\mathit{suf}^{(h+1)}(q) =_{\mathit{Def}} \mathit{suf}(\mathit{suf}^{(h)}(q))$, for $h \geq 0$, provided that $\mathit{suf}^{(h)}(q) \neq q_0$).

Proof. Let $\mathit{val}(p) \sqsupset \mathit{val}(q)$. From the definition (2.4) of the function $\mathit{suf}(\cdot)$, we have $\mathit{val}(p) \sqsupseteq \mathit{suf}(\mathit{val}(q)) \sqsupset \mathit{val}(q)$, so that $\mathit{val}(p) \sqsupseteq \mathit{val}(R_{\mathcal{P}}(\mathit{suf}(\mathit{val}(q)))) \sqsupset \mathit{val}(q)$. Thus, (a) follows by observing that $\mathit{val}(\mathit{suf}(q)) = \mathit{val}(R_{\mathcal{P}}(\mathit{suf}(\mathit{val}(q))))$.

Concerning (b), we argue as follows. From (a) we have $\mathit{val}(p) \sqsupseteq \mathit{val}(\mathit{suf}(q))$. If $\mathit{val}(p) = \mathit{val}(\mathit{suf}(q))$, then $\mathit{suf}^{(1)}(q) = \mathit{suf}(q) = p$, and (b) holds. Otherwise, $\mathit{val}(p) \sqsupset \mathit{val}(\mathit{suf}(q))$. By applying (a) repeatedly, we eventually obtain a sequence

$$\mathit{val}(p) = \mathit{val}(\mathit{suf}^{(k)}(q)) \sqsupset \mathit{val}(\mathit{suf}^{(k-1)}(q)) \sqsupset \cdots \sqsupset \mathit{val}(\mathit{suf}(q)) \sqsupset \mathit{val}(q) ,$$

[2] $R_{\mathcal{P}}$ is the equivalence relation defined by (2.2).

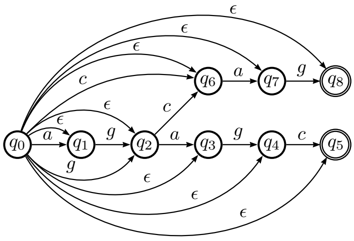


Figure 4.4: Suffix NFA for the set of strings {**agagc**,**agcag**}.

for some $k \geq 1$, which implies $suf^{(k)}(q) = p$, thus proving (b). □

Given a set of patterns $\mathcal{P}$ over Σ, the suffix NFA $S^{\mathsf{r}}_{\mathcal{P}} = (Q, \Sigma, \delta_{S^{\mathsf{r}}}, q_0, F)$ for $(\mathcal{P}_{l_{min}})^{\mathsf{r}}$ can be used to find the occurrences of the patterns of $\mathcal{P}$ in a text T of length n by observing that a pattern $P \in \mathcal{P}$ of length m has an occurrence in T ending at position $i + m - 1$, i.e., $T[i \mathinner{..} i + m - 1] = P$, if and only if $\delta^*_{S^{\mathsf{r}}}(q_0, (T[i \mathinner{..} i + l_{min} - 1])^{\mathsf{r}})$ contains a final state $q \in F$ such that $val(q) \sqsupseteq P^{\mathsf{r}}$ and $T[i + l_{min} \mathinner{..} i + m - 1] \sqsupseteq P$. Hence, to find all the occurrences of the patterns in $\mathcal{P}$ in T, one can compute $\delta^*_{S^{\mathsf{r}}}(q_0, (T[i \mathinner{..} i + l_{min} - 1])^{\mathsf{r}}) \cap F$, for $i = 0, 1, \ldots, n - l_{min}$ and then make the appropriate checks for the candidate matches. With this approach it is possible to skip windows: in fact, for a window of T of size l_{min} beginning at position i, let l be the length of the longest proper suffix of $T[i \mathinner{..} i + l_{min} - 1]$ such that $\delta^*_{S^{\mathsf{r}}}(q_0, (T[i + l_{min} - l \mathinner{..} i + l_{min} - 1])^{\mathsf{r}}) \cap F \neq \emptyset$. Then, the windows at positions $i, i + 1, \ldots i + l_{min} - l - 1$ can be safely skipped.

4.4 Bit-parallel simulation of NFAs for the multiple string matching problem

In Section 3.2 we have analyzed a technique to represent a certain class of NFAs which allows to improve the space complexity as compared to the naive technique but which still requires exponential space in the number of states of the automaton. We recall that the immediate solution of storing the maps $\Phi(\cdot)$ and $B(\cdot)$ as tables of bit words, respectively indexed by sets of states and by characters in Σ, requires $(2^m + \sigma) \cdot m$ bits for an automaton with m states. In particular, the exponential term in the space complexity is due to the $\Phi(\cdot)$ map. Thus we have to find a better way to store the map $\Phi(\cdot)$, exploiting the fact that $\Phi(D)$ needs to be evaluated over reachable configurations D of $A_{\mathcal{P}}$ or $S_{\mathcal{P}}$ only.

In Sections 4.5 and 4.6 we will show that the map $\Phi(\cdot)$ can be conveniently stored in $\mathcal{O}(m^2)$-space, for both the Aho-Corasick NFA and the suffix NFA. More specifically, we will show that, in both cases, each nonempty reachable configuration D can be represented in terms of a unique state, which will be referred to as $lead(D)$. This will allow us to represent $\Phi(D)$ as $\dot{\Phi}(lead(D))$, where $\dot{\Phi} : Q \to \mathscr{P}(Q)$ is the map such that the q-th bit of $\dot{\Phi}(p)$ is set if and only if there is a transition to state q originating from p or from any other state belonging to the reachable configuration uniquely identified by p. Plainly, the map $\dot{\Phi}$ can be stored in $\mathcal{O}(m^2)$-space and allows to rewrite equation (b) of Lemma 3.1 as

$$\delta(D, c) = \dot{\Phi}(lead(D)) \cap B(c)\,,$$

which in turn translates readily into the bit-parallel assignment

$$D \leftarrow \dot{\Phi}[lead(D)] \;\&\; B[c]\,.$$

4.5 Bit-parallel simulation of the Aho-Corasick NFA for a set of patterns

In this section we present a bit-parallel encoding of the Aho-Corasick NFA; specifically, following the idea explained in the previous section, we first show that each reachable configuration of $A_{\mathcal{P}}$ is uniquely identified by a single state. Then, we devise the map $\dot{\Phi}(\cdot)$ by using the relation between reachable configurations of the automaton and the associated failure function, and prove its correctness. Finally, we show that the map $lead(\cdot)$ admits an efficient implementation.

A key result is contained in the following elementary lemma.

Lemma 4.3. *Let $A_{\mathcal{P}} = (Q, \Sigma, \delta, q_0, F)$ be the Aho-Corasick NFA for a finite set $\mathcal{P}$ of patterns over the alphabet Σ, and let $u \in \Sigma^*$. Then $\delta^*(q_0, u) = \{q \in Q \mid lbl(q) \sqsupseteq u\}$.*

Proof. For $u = \varepsilon$, the lemma holds plainly. Thus, let $u = u'.c$, with $u' \in \Sigma^*$ and $c \in \Sigma$. We first show by induction on u that $\delta^*(q_0, u) \subseteq \{q \in Q \mid lbl(q) \sqsupseteq u\}$. Let $p \in \delta^*(q_0, u)$. Since, by (2.1),

$$\delta^*(q_0, u'.c) = \delta^*(\delta^*(q_0, u'), c) = \delta(\delta^*(q_0, u'), c) = \bigcup_{q \in \delta^*(q_0, u')} \delta(q, c)\,,$$

we have $p \in \delta(\overline{q}, c)$, for some $\overline{q} \in \delta^*(q_0, u')$, so that, by inductive hypothesis, $lbl(\overline{q}) \sqsupseteq u'$, and therefore $lbl(p) = lbl(\overline{q}).c \sqsupseteq u'.c = u$.

To show the converse inclusion relationship, let $p \in Q$ be such that $lbl(p) \sqsupseteq u$. We prove by induction on $lbl(p)$ that $p \in \delta^*(q_0, u)$. In view of (4.1), we may dismiss at once the case in which $lbl(p) = \varepsilon$, i.e., $p = q_0$, and therefore assume that $lbl(p) = lbl(p').c$, for some $p' \in Q$ and $c \in \Sigma$. Hence $u = u'.c$, for some $u' \in \Sigma^*$ such that $lbl(p') \sqsupseteq u'$, so that, by inductive hypothesis, we have $p' \in \delta^*(q_0, u')$. Thus, by (2.1), $p \in \delta(p', c) \subseteq \delta(\delta^*(q_0, u'), c) = \delta^*(\delta^*(q_0, u'), c) = \delta^*(q_0, u'.c) = \delta^*(q_0, u)$. □

Given a reachable configuration D, the previous lemma implies that for any two distinct states $p, p' \in D$ we have $|lbl(p)| \neq |lbl(p')|$, since either $lbl(p) \sqsupset lbl(p')$ or $lbl(p') \sqsupset lbl(p)$. Thus there must exist a unique state $\overline{q} \in D$ such that $|lbl(p)| \leq |lbl(\overline{q})|$, for every $p \in D$. Let us denote such a state by $lead(D)$. Then we have:

Corollary 4.1. *Let $A_{\mathcal{P}} = (Q, \Sigma, \delta, q_0, F)$ be the Aho-Corasick NFA for a finite set $\mathcal{P}$ of patterns over Σ, and let D be a reachable configuration of $A_{\mathcal{P}}$. Then $D = \{q \in Q \mid lbl(q) \sqsupseteq lbl(lead(D))\}$.*

Proof. Let $u \in \Sigma^*$ be such that $D = \delta^*(q_0, u)$. In view of Lemma 4.3, it suffices to observe that $lbl(q) \sqsupseteq u$ if and only if $lbl(q) \sqsupseteq lbl(lead(D))$, for every $q \in Q$. □

From the preceding corollary, it readily follows that the reachable configurations of the Aho-Corasick NFA $A_{\mathcal{P}} = (Q, \Sigma, \delta, q_0, F)$, for a set $\mathcal{P}$ of patterns, are in 1-1 correspondence with its states, and therefore their number is $|Q|$.

A convenient way to represent Φ uses the map $\dot{\Phi}_A : Q \to \mathscr{P}(Q)$, recursively defined by

$$\dot{\Phi}_A(q) =_{Def} \begin{cases} Follow(q_0), & \text{if } q = q_0 \\ Follow(q) \cup \dot{\Phi}_A(fail(q)), & \text{if } q \neq q_0\,, \end{cases} \tag{4.3}$$

as shown in the following lemma.

Lemma 4.4. *For any reachable configuration D of the Aho-Corasick NFA $A_{\mathcal{P}}$, we have $\Phi(D) = \dot{\Phi}_A(lead(D))$.*

Proof. We proceed by induction on $|lbl(lead(D))|$. If $|lbl(lead(D))| = 0$, then $lead(D) = q_0$ and $D = \{q_0\}$, so that $\Phi(D) = Follow(q_0) = \dot{\Phi}_A(q_0) = \dot{\Phi}_A(lead(D))$.

For the inductive step, we have

$$\begin{aligned}
\Phi(D) &= \bigcup_{q \in D} \mathit{Follow}(q) = \bigcup_{\substack{q \in Q \\ \mathit{lbl}(q) \sqsupseteq \mathit{lbl}(\mathit{lead}(D))}} \mathit{Follow}(q) \\
&= \mathit{Follow}(\mathit{lead}(D)) \cup \bigcup_{\substack{q \in Q \\ \mathit{lbl}(q) \sqsupset \mathit{lbl}(\mathit{lead}(D))}} \mathit{Follow}(q) \\
&= \mathit{Follow}(\mathit{lead}(D)) \cup \bigcup_{\substack{q \in Q \\ \mathit{lbl}(q) \sqsupseteq \mathit{lbl}(\mathit{fail}(\mathit{lead}(D)))}} \mathit{Follow}(q) \\
&= \mathit{Follow}(\mathit{lead}(D)) \cup \Phi(\{q \in Q \mid \mathit{lbl}(q) \sqsupseteq \mathit{lbl}(\mathit{fail}(\mathit{lead}(D)))\}) \\
&= \mathit{Follow}(\mathit{lead}(D)) \cup \dot{\Phi}_A(\mathit{fail}(\mathit{lead}(D)) = \dot{\Phi}_A(\mathit{lead}(D))). \quad \square
\end{aligned}$$

Plainly, the map $\dot{\Phi}_A(\cdot)$ requires only $|Q|^2$ bits. Additionally, the map $\mathit{lead}(\cdot)$ can be computed very efficiently at run-time, provided that the states of $A_{\mathcal{P}}$ are ordered in such a way that a state p precedes a state q whenever $|\mathit{lbl}(p)| < |\mathit{lbl}(q)|$ (say, by a breadth-first visit of $A_{\mathcal{P}}$ from q_0). Indeed, in such a case, if we assume that D is encoded as a bit mask, then $\mathit{lead}(D)$ is the index of the highest bit of D set to 1.

4.5.1 The Log-And algorithm

Based on the previous considerations, we present an efficient bit-parallel algorithm, which we call Log-And, for solving the multiple string matching problem.

In the Log-And algorithm, reported in Fig. 4.5, the sets D, B and the map $\dot{\Phi}_A$ are encoded as bit tables.

As opposed to the Shift-And algorithm, bit 0 is reserved for the initial state, so that $\mathit{lead}(D)$ is never computed for an empty set (0 value) as the initial state is always active.

In the preprocessing phase, the Log-And algorithm iterates over the nodes of $A_{\mathcal{P}}$, which are assumed to be sorted by a breadth-first search; for each node, the corresponding Φ mask is computed using (4.3) and the B masks associated to the labels of its outgoing edges are augmented accordingly. The algorithm precomputes also a *final state* bit mask, L, where a bit is set to 1 if and only if it corresponds to a final state of the automaton. Then, during the searching phase, the Log-And algorithm scans the text T, character by character, using the

```
Log-And ($T$; $P_1, P_2, \dots, P_r$)
  /* Preprocessing */
  Let $A_{\mathcal{P}} = (Q, \Sigma, \delta, q_0, F)$ be the Aho-Corasick NFA relative to the set of
     patterns $\mathcal{P} = \{P_1, P_2, \dots, P_r\}$ and let the maps $Follow()$, $lbl()$, and $fail()$ be
     defined as before, relative to $A_{\mathcal{P}}$. We also assume that $Q = \{0, 1, \dots, \ell - 1\}$,
     where $\ell = |Q|$, and that if $|lbl(p)| < |lbl(q)|$ then $p < q$, for any $p, q \in Q$.
  $L \leftarrow 0^\ell$
  for $c \in \Sigma$ do $B[c] \leftarrow 0^{\ell-1}1$
  for $p \leftarrow 0$ to $\ell - 1$ do
     $\dot{\Phi}_A[p] \leftarrow 0^{\ell-1}1$
     for $q \in Follow(p) \setminus \{0\}$ do
        $H \leftarrow 0^{\ell-1}1 \ll q$
        $c \leftarrow lbl(p, q)$
        $B[c] \leftarrow B[c] \mid H$
        if $q \in F$ then $L \leftarrow L \mid H$
        $\dot{\Phi}_A[p] \leftarrow \dot{\Phi}_A[p] \mid H$
     if $p \neq 0$ then
        $\dot{\Phi}_A[p] \leftarrow \dot{\Phi}_A[p] \mid \dot{\Phi}_A[fail(p)]$
  /* Searching */
  $D \leftarrow 0^{\ell-1}1$
  for $j \leftarrow 0$ to $|T| - 1$ do
     $lead \leftarrow \lfloor \log_2(D) \rfloor$
     $D \leftarrow \dot{\Phi}_A[lead]$ & $B[T[j]]$
     if $D$ & $L \neq 0^\ell$ then Output($j$)
```

Figure 4.5: The Log-And algorithm for the multiple string matching problem.

following basic transition, based on Lemma 3.1(b),

$$D \leftarrow \dot{\Phi}_A[\lfloor \log_2(D) \rfloor] \mathbin{\&} B[c]\,.$$

The resulting algorithm has $\mathcal{O}((m + \sigma)\lceil m/w \rceil)$-space and $\mathcal{O}(n\lceil m/w \rceil)$-searching time complexity, where $n = |T|$, m is the number of nodes of $A_{\mathcal{P}}$, σ is the alphabet size, and w is the word size in bits. When $m \in \mathcal{O}(w)$, the Log-And algorithm turns out to have a $\mathcal{O}(m + \sigma)$-space and $\mathcal{O}(n)$-searching time complexity.

If one is also interested in retrieving the patterns (if any) that match at each text position, it is convenient to precompute a table which maps each final state of $A_{\mathcal{P}}$ onto the corresponding pattern index. Then, in the searching phase, for each position j, the algorithm iterates over the bits of (D & L) by computing the index of the highest bit set and querying the corresponding pattern number. The whole sequence is repeated, after having cleared the highest bit, until there are no more bits set.

4.6 Bit-parallel simulation of the suffix NFA for a set of patterns

In this section we illustrate a bit-parallel encoding of the suffix automaton induced by the DAWG data structure for a set of patterns. We observe that the maximal trie of a set $\mathcal{P}$ of patterns can also be turned into an automaton that recognizes the language $\mathit{Suff}(\mathcal{P})$, by adding an ε-transition from the initial state to all remaining states. The size of the DAWG data structure can vary between the number $|Q_{A_\mathcal{P}}|$ of states of the Aho-Corasick automaton for $\mathcal{P}$ and $2 \cdot \mathit{size}(\mathcal{P}) - 1$ (cf. [13]). Thus, although the DAWG allows to factor prefix redundancy in the patterns, it is not always preferable to the maximal trie, whose size is $\mathit{size}(\mathcal{P})$. However, it turns out that the average size of the DAWG is close to $|Q_{A_\mathcal{P}}|$ which, depending on the degree of prefix redundancy in $\mathcal{P}$ may be much smaller than $\mathit{size}(\mathcal{P})$.

Let $S_\mathcal{P}$ be the suffix NFA for a set $\mathcal{P}$ of patterns over an alphabet Σ. We devise a bit-parallel encoding of this automaton much along the lines of what has been done for the $A_\mathcal{P}$ automaton.

The following lemma is analogous to Lemma 4.3 in the present context of suffix NFAs. For the sake of completeness, we include its proof, though, aside from a few adaptations, it follows closely the proof of Lemma 4.3.

Lemma 4.5. *Let $S_\mathcal{P} = (Q, \Sigma, \delta_S, q_0, F)$ be the suffix NFA for a finite set $\mathcal{P}$ of patterns, and let $u \in \Sigma^*$. Then $\delta_S^*(q_0, u) = \{q \in Q \mid u \sqsupseteq \mathit{val}(q)\}$.*

Proof. For $u = \varepsilon$, the lemma holds plainly. Thus, let $u = u'.c$, with $u' \in \Sigma^*$ and $c \in \Sigma$. We first show by induction on u that $\delta_S^*(q_0, u) \subseteq \{q \in Q \mid u \sqsupseteq \mathit{val}(q)\}$. Thus, let $p \in \delta_S^*(q_0, u)$. By (2.1), we have $\delta_S^*(q_0, u'.c) = \delta_S^*(\delta_S^*(q_0, u'), c) = \delta_S(\delta_S^*(q_0, u'), c) = \bigcup_{q \in \delta_S^*(q_0, u')} \delta_S(q, c)$. Hence, $p \in \delta_S(\overline{q}, c)$, for some $\overline{q} \in \delta_S^*(q_0, u')$, so that, by inductive hypothesis, $u' \sqsupseteq \mathit{val}(\overline{q})$, and therefore $u = u'.c \sqsupseteq \mathit{val}(\overline{q}).c = \mathit{val}(p)$.

To show the converse inclusion relationship, let $p \in Q$ be such that $u \sqsupseteq \mathit{val}(p)$. We prove by induction on $\mathit{val}(p)$ that $p \in \delta_S^*(q_0, u)$. If $\mathit{val}(q) = \varepsilon$, then $p = q_0$ and $u = \varepsilon$, so that $p \in \delta_S^*(q_0, u)$ holds trivially. Let us then assume that $\mathit{val}(p) = \mathit{val}(p').c$, for some $p' \in Q$ and $c \in \Sigma$. Hence $u = u'.c$, for some $u' \in \Sigma^*$ such that $u' \sqsupseteq \mathit{val}(p')$, so that, by inductive hypothesis, we have $p' \in \delta_S^*(q_0, u')$. Thus, by (2.1), $p \in \delta_S(p', c) \subseteq \delta_S(\delta_S^*(q_0, u'), c) = \delta_S^*(\delta_S^*(q_0, u'), c) = \delta_S^*(q_0, u'.c) = \delta_S^*(q_0, u)$. □

The following lemma illustrates some useful properties concerning a nonempty reachable configuration $D = \delta_S^*(q_0, u)$ of $S_\mathcal{P}$ for a string u and relative equivalence class $R_\mathcal{P}(u)$ defined by (2.2).

Lemma 4.6. *Let $S_\mathcal{P} = (Q, \Sigma, \delta_S, q_0, F)$ be the suffix NFA for a set of patterns $\mathcal{P}$. Then, for any string $u \in \Sigma^*$, the following implications hold:*

(a) if $q \in \delta_S^*(q_0, u)$, then $val(R_\mathcal{P}(u)) \sqsupseteq val(q)$;

(b) if $\delta_S^*(q_0, u) \neq \emptyset$, then $R_\mathcal{P}(u) \in \delta_S^*(q_0, u)$;

(c) if $\delta_S^*(q_0, u) = \delta_S^*(q_0, v) \neq \emptyset$, then $u \ R_\mathcal{P} \ v$, for $v \in \Sigma^*$.

Proof. Concerning (a), let $q \in \delta_S^*(q_0, u)$. From (4.2), it follows that $u \in Fact(\mathcal{P})$, so that $val(R_\mathcal{P}(u))$ is defined. Then by lemma 4.5 we have that $u \sqsupseteq val(q)$, which in turn implies that $end\text{-}pos(val(q)) \subseteq end\text{-}pos(u) = end\text{-}pos(val(R_\mathcal{P}(u)))$. Hence, $val(R_\mathcal{P}(u)) \sqsupseteq val(q)$.

Concerning (b), from the very definitions of $R_\mathcal{P}$ and $val(\cdot)$ (see (2.2) and (2.3)), we have that $u \sqsupseteq val(R_\mathcal{P}(u))$ which, by Lemma 4.5, implies that $R_\mathcal{P}(u) \in \delta_S^*(q_0, u)$.

Finally, concerning (c), let $\delta_S^*(q_0, u) = \delta_S^*(q_0, v) \neq \emptyset$. Then (b) yields $R_\mathcal{P}(u) \in \delta_S^*(q_0, v)$ and $R_\mathcal{P}(v) \in \delta_S^*(q_0, u)$ which, again by Lemma 4.5, imply $val(R_\mathcal{P}(v)) \sqsupseteq val(R_\mathcal{P}(u))$ and $val(R_\mathcal{P}(u)) \sqsupseteq val(R_\mathcal{P}(v))$, respectively. Hence, $val(R_\mathcal{P}(u)) = val(R_\mathcal{P}(v))$ so that $u \ R_\mathcal{P} \ v$. □

Given a nonempty reachable configuration D for a string u, the previous lemma implies that the set

$$\{R_\mathcal{P}(u) \mid \delta_S^*(q_0, u) = D, \text{ for } u \in Fact(\mathcal{P})\}$$

has exactly one element. Therefore the following definition is well founded

$$lead(D) =_{\text{Def}} R_\mathcal{P}(u),$$

for any $u \in Fact(\mathcal{P})$ such that $\delta_S^*(q_0, u) = D$.

Corollary 4.2. *Let $S_\mathcal{P} = (Q, \Sigma, \delta, q_0, F)$ be the suffix NFA for a set of patterns $\mathcal{P}$, and let D be a nonempty reachable configuration of $S_\mathcal{P}$. Then $D = \{q \in Q \mid val(lead(D)) \sqsupseteq val(q)\}$.*

Proof. Let $u \in \mathit{Fact}(\mathcal{P})$ such that $\delta_S^*(q_0, u) = D$ and let $q \in D$. From Lemma 4.6(a) we have that $\mathit{val}(\mathit{lead}(D)) = \mathit{val}(R_{\mathcal{P}}(u)) \sqsupseteq \mathit{val}(q)$. Conversely, if $\mathit{val}(R_{\mathcal{P}}(u)) \sqsupseteq \mathit{val}(q)$, then $u \sqsupseteq \mathit{val}(q)$, so that, by Lemma 4.5, $q \in D$. □

From the preceding corollary, it follows at once that the nonempty reachable configurations of a suffix NFA $S_{\mathcal{P}} = (Q, \Sigma, \delta_S, q_0, F)$ for a set $\mathcal{P}$ of patterns are in 1-1 correspondence with its states, and therefore their number is $|Q|$.

For $q \in Q$, let

$$\mathit{rsuf}(q) \ =_{\mathrm{Def}} \mathit{suf}^{-1}[\{q\}] = \{p \in Q \mid \mathit{suf}(p) = q\}$$

be the set of states whose suffix link is q, where $\mathit{suf}(\cdot)$ is the map defined in (2.4).

We will show that a reachable configuration of $S_{\mathcal{P}}$ can be represented in terms of the maps $\mathit{lead}(\cdot)$ and $\mathit{rsuf}(\cdot)$.

Lemma 4.7. *Let D be a nonempty reachable configuration of the suffix NFA $S_{\mathcal{P}} = (Q, \Sigma, \delta_S, q_0, F)$ for a set $\mathcal{P}$ of patterns. Then*

$$D = \{\mathit{lead}(D)\} \cup \bigcup_{p \in \mathit{rsuf}(\mathit{lead}(D))} \{q \in Q \mid \mathit{val}(p) \sqsupseteq \mathit{val}(q)\}\,.$$

Proof. From Corollary 4.2 we have

$$\begin{aligned} D &= \{q \in Q \mid \mathit{val}(\mathit{lead}(D)) \sqsupseteq \mathit{val}(q)\} \\ &= \{\mathit{lead}(D)\} \cup \{q \in Q \mid \mathit{val}(\mathit{lead}(D)) \sqsupset \mathit{val}(q)\}\,. \end{aligned}$$

Then to prove the lemma it is enough to show that

$$\{q \in Q \mid \mathit{val}(\mathit{lead}(D)) \sqsupset \mathit{val}(q)\} = \bigcup_{p \in \mathit{rsuf}(\mathit{lead}(D))} \{q \in Q \mid \mathit{val}(p) \sqsupseteq \mathit{val}(q)\}\,.$$

Let $q' \in Q$ be such that $\mathit{val}(\mathit{lead}(D)) \sqsupset \mathit{val}(q')$. By Lemma 4.2(b), there exists $k \geq 1$ such that $\mathit{suf}^{(k)}(q') = \mathit{lead}(D)$. Let $p' = \mathit{suf}^{(k-1)}(q')$. Plainly, $\mathit{suf}(p') = \mathit{lead}(D)$, so that $p' \in \mathit{rsuf}(\mathit{lead}(D))$. Additionally, $\mathit{val}(p') = \mathit{val}(\mathit{suf}^{(k-1)}(q')) \sqsupseteq \mathit{val}(q')$.

Hence,

$$q' \in \{q \in Q \mid \mathit{val}(p') \sqsupseteq \mathit{val}(q)\} \subseteq \bigcup_{p \in \mathit{rsuf}(\mathit{lead}(D))} \{q \in Q \mid \mathit{val}(p) \sqsupseteq \mathit{val}(q)\}$$

so that

$$\{q \in Q \mid val(lead(D)) \sqsupset val(q)\} \subseteq \bigcup_{p \in rsuf(lead(D))} \{q \in Q \mid val(p) \sqsupseteq val(q)\}\,.$$

To prove the converse relationship, let

$$q' \in \bigcup_{p \in rsuf(lead(D))} \{q \in Q \mid val(p) \sqsupseteq val(q)\}$$

and let $p' \in rsuf(lead(D))$ such that $val(p') \sqsupseteq val(q')$. Then $val(lead(D)) = val(suf(p')) \sqsupset val(p) \sqsupseteq val(q')$, since $suf(p') = lead(D)$. Hence $q' \in \{q \in Q \mid val(lead(D)) \sqsupset val(q)\}$ proving

$$\bigcup_{p \in rsuf(lead(D))} \{q \in Q \mid val(p) \sqsupseteq val(q)\} \subseteq \{q \in Q \mid val(lead(D)) \sqsupset val(q)\}$$

and in turn completing the proof of the lemma. □

A convenient way to represent the map $\Phi(\cdot)$ makes use of the following map $\dot{\Phi}_S : Q \to \mathscr{P}(Q)$, defined by

$$\dot{\Phi}_S(q) =_{Def} \begin{cases} Follow(q), & \text{if } rsuf(q) = \emptyset \\ Follow(q) \cup \bigcup_{p \in rsuf(q)} \dot{\Phi}_S(p), & \text{if } rsuf(q) \neq \emptyset\,, \end{cases} \tag{4.4}$$

as proved in the following lemma.

Lemma 4.8. *For any nonempty reachable configuration D of the suffix NFA $S_{\mathcal{P}} = (Q, \Sigma, \delta_S, q_0, F)$ for a set $\mathcal{P}$ of patterns, we have*

$$\Phi(D) = \dot{\Phi}_S(lead(D))\,.$$

Proof. To begin with, let us put $D_p =_{\text{Def}} \{q \in Q \mid val(p) \sqsupseteq val(q)\}$, for $p \in rsuf(lead(D))$, so that the decomposition of D provided by the preceding lemma can be rewritten in a more compact way as

$$D = \{lead(D)\} \cup \bigcup_{p \in rsuf(lead(D))} D_p\,. \tag{4.5}$$

Additionally, we observe that

$$lead(D_p) = p\,, \tag{4.6}$$

for each $p \in rsuf(lead(D))$. Indeed, by Lemma 4.5,

$$\delta^*_S(q_0, val(p)) = \{q \in Q \mid val(p) \sqsupseteq val(p)\} = D_p\,,$$

so that $lead(D_p) = R_{\mathcal{P}}(val(p)) = p$. We are now ready to prove the lemma. We proceed by induction on $height(lead(D))$, where

$$height(q) \ =_{\text{Def}} \ \text{length of the longest chain of suffix link ending at } q\,.$$

If $height(lead(D)) = 0$ then $D = \{lead(D)\}$ and $rsuf(lead(D)) = \emptyset$. For the inductive step, in view of (4.5) and (4.6) above and of the fact that $height(lead(D_p)) < height(lead(D))$ for $p \in rsuf(lead(D))$, we have

$$\begin{aligned}
\Phi(D) &= \Phi(\{lead(D)\}) \cup \bigcup_{p \in rsuf(lead(D))} \Phi(D_p) \\
&= Follow(lead(D)) \cup \bigcup_{p \in rsuf(lead(D))} \dot{\Phi}_S(lead(D_p)) \\
&= Follow(lead(D)) \cup \bigcup_{p \in rsuf(lead(D))} \dot{\Phi}_S(p) \\
&= \dot{\Phi}_S(lead(D))\,,
\end{aligned}$$

completing the proof of the lemma. □

As for the Aho-Corasick NFA, the map $\dot{\Phi}_S$ requires $|Q|^2$ bits only and the map $lead(\cdot)$ can be computed very efficiently at run-time, provided that the states of $S_{\mathcal{P}}$ are ordered in such a way that a state p precedes a state q whenever $|val(p)| < |val(q)|$ (say, by a breadth-first search from q_0). Indeed, in such a case, if we assume that D is encoded as a bit mask, then $lead(D)$ is the index of the lowest bit of D set to 1.

4.6.1 The **Backward-Log-And** algorithm

In this section we present the Backward-Log-And algorithm, a BNDM-like bit-parallel algorithm based on the suffix NFA, for the multiple string matching

```
Backward-Log-And ($T$; $P_1, P_2, \ldots, P_r$)
  /* Preprocessing */
  Let $S_{\mathcal{P}^r_{l_{min}}} = (Q, \Sigma, \delta, q_0, F)$ be the suffix NFA relative to the set of patterns
  $\mathcal{P}^r = \{P^r_1, P^r_2, \ldots, P^r_r\}$ and let the maps $Follow()$, $val()$, and $suf()$ be defined
  as before, relative to $S_{\mathcal{P}^r_{l_{min}}}$. We also assume that $Q = \{0, 1, \ldots, \ell - 1\}$, where
  $\ell = |Q|$, and that if $|val(p)| < |val(q)|$ then $p < q$, for any $p, q \in Q$.
  $L \leftarrow 0^\ell$
  for $c \in \Sigma$ do $B[c] \leftarrow 0^\ell$
  for $p \leftarrow \ell - 1$ to $0$ do
     for $q \in Follow(p)$ do
        $H \leftarrow 0^{\ell-1}1 \ll q$
        $c \leftarrow lbl(p, q)$
        $B[c] \leftarrow B[c] \mid H$
        if $q \in F$ then $L \leftarrow L \mid H$
        $\dot{\Phi}_S[p] \leftarrow \dot{\Phi}_S[p] \mid H$
     if $p \neq 0$ then
        $\dot{\Phi}_S[suf(p)] \leftarrow \dot{\Phi}_S[suf(p)] \mid \dot{\Phi}_S[p]$
  /* Searching */
  $j \leftarrow l - 1$
  while $j < n$ do
     $k \leftarrow 0, last \leftarrow 0$
     $D \leftarrow 1^\ell$
     while $D \neq 0^\ell$ do
        $lead \leftarrow \lfloor \log_2(D \,\&\, (\sim D + 1)) \rfloor$
        $D \leftarrow \dot{\Phi}_S[lead] \,\&\, B[T[j - k]]$
        if $D \,\&\, L \neq 0^\ell$ then
           if $k < l$ then
              $last \leftarrow k$
           else Output($j$)
        $k \leftarrow k + 1$
     $j \leftarrow j + l - last$
```

Figure 4.6: The Backward-Log-And algorithm for the multiple string matching problem.

problem. In the Backward-Log-And algorithm, whose pseudocode is reported in Fig. 4.6, the sets D, B and the map $\dot{\Phi}_S(\cdot)$ are encoded as bit tables. There is no need to reserve bit 0 for the initial state, as the simulation stops when there are no longer active states. For simplicity, in the pseudocode it is assumed that all patterns have the same length l.

During the preprocessing phase, the Backward-Log-And algorithm iterates over the states of the suffix NFA $S_{\mathcal{P}^r_{l_{min}}}$, which are assumed to be sorted by a breadth-first search; for each state, the corresponding masks B and L are computed as in the Log-And algorithm, while the mask Φ is computed using (4.4). Then, during the searching phase, the Backward-Log-And algorithm scans the text T, character by character, using the following transition based on Lemma 3.1(b),

$$D \leftarrow \dot{\Phi}_A[\lfloor \log_2(D \,\&\, (\sim D + 1)) \rfloor] \,\&\, B[c]\,.$$

The resulting algorithm has $\mathcal{O}(n\lceil m/w \rceil l_{min})$-searching time and $\mathcal{O}((m+\sigma)\lceil m/w \rceil)$-space complexity, where $n = |T|$, l_{min} is the length of the shortest pattern, m is the number of nodes of $S_{\mathcal{P}^r_{l_{min}}}$, σ is the alphabet size and w is the word size in bits. When $m \in \mathcal{O}(w)$, the Backward-Log-And algorithm turns out to have a $\mathcal{O}(m + \sigma)$-space and $\mathcal{O}(nl_{min})$-searching time complexity.

Chapter 5

The approximate string matching problem

The *approximate string matching* problem consists in finding all the occurrences of a pattern in a text allowing for a finite number of errors. Errors are formalized by means of a distance function on strings which maps two strings into the minimal cost of a sequence of edit operations that are needed to convert the first string into the second string. Well known distance functions for this problem are the *edit distance* [49] (also called the *Levenshtein distance*) or the *Damerau edit distance* [31]. The edit operations in the former edit distance are *insertion*, *deletion*, and *substitution* of characters; instead, in the second case, one allows for *swaps* of characters, i.e., transpositions of two adjacent characters [1]. *Approximate string matching* under the *Damerau* distance is also known as *string matching with swaps*. In Section 5.1 we illustrate an algorithm for this problem that is sublinear on average and is also able to report, for each occurrence of the pattern, the corresponding number of swaps, without any time or space overhead. The distances described above assume that changes between strings occur locally, i.e., only a small portion of the string is involved in the mutation event. In contrast, evidence shows that large scale changes are possible. For example, large pieces of DNA can be moved from one location to another (*translocations*) or replaced by their reversed complements (*inversions*). In Section 5.2 we introduce a distance function modelled on these kinds of transformations, based on edit operations

[1] For an in-depth survey on approximate string matching see [53].

that involve substrings rather than just single characters. We then present an algorithm, based on dynamic programming and on finite automata, to solve the *approximate string matching* problem under this distance.

5.1 String matching with swaps

The *pattern matching with swaps* problem (swap matching problem, for short) is a well-studied variant of the classic pattern matching problem. It consists in finding all occurrences, allowing for swaps of characters , of a pattern P of length m in a text T of length n, with P and T sequences of characters over a common 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 ending (or, equivalently, starting) at location j. All swaps are constrained to be disjoint, i.e., each character can be involved in at most one swap. Moreover, we assume that identical adjacent characters cannot be swapped.

Amir *et al.* [3] provided a $\mathcal{O}(nm^{1/3}\log m)$-time algorithm in the case of alphabets of size 2 and showed that the case of alphabets of size greater than 2 can be reduced to that of size 2 with a $\mathcal{O}(\log\sigma)$-time overhead. In [5] Amir *et al.* studied some rather restrictive cases in which a $\mathcal{O}(n\log^2 m)$-time algorithm can be obtained. More recently, Amir *et al.* solved the swap matching problem in $\mathcal{O}(n\log m\log\sigma)$-time [4]. We observe that the above solutions are all based on the fast Fourier transform (FFT) technique.

Iliopoulos and Rahman provided the first solution to the swap matching problem that does not make use of the FFT technique [41]. They modelled the problem using a graph-theoretic approach and devised an algorithm, based on the bit-parallelism technique [8], which runs in $\mathcal{O}((n+m)\log m)$-time if the pattern length is $\mathcal{O}(w)$, where w is the size in bits of a computer word.

More recently, Cantone and Faro [20] presented an algorithm, named Cross-Sampling, that has $\mathcal{O}(nm)$ worst-case time complexity and admits a bit-parallel implementation, named BP-Cross-Sampling, which achieves $\mathcal{O}(n)$ worst-case time complexity if the pattern length is $\mathcal{O}(w)$. In [16] the same authors presented another algorithm, named Backward-Cross-Sampling, that has $\mathcal{O}(nm^2)$ worst-case time complexity but shows a sublinear behaviour on average. They also devised a bit-parallel implementation, named BP-Backward-Cross-Sampling, which has $\mathcal{O}(nm)$-time complexity if the pattern length is $\mathcal{O}(w)$.

In this section we investigate the approximate variant of the swap matching problem. The *approximate pattern matching with swaps* problem is tantamount to computing, for each text location j, the number of swaps necessary to convert the pattern into the substring of length m ending at j.

A straightforward solution to the approximate swap matching problem consists in searching for all the swapped occurrences of the pattern, using any algorithm for the standard swap matching problem. For a given occurrence, to compute the associated number of swaps it suffices to count the number of mismatches relative to the original pattern and then divide it by 2.

Amir *et al.* presented a theoretical algorithm that solves the approximate swap matching problem in time $\mathcal{O}(n \log m \log \sigma)$ [6].

Cantone and Faro presented also an extension of the CROSS-SAMPLING algorithm, named APPROXIMATE-CROSS-SAMPLING, for the approximate swap matching problem. However, its bit-parallel implementation has a notably high space overhead since it requires $(m \log(\lfloor m/2 \rfloor + 1) + m)$ bits [20].

In this section we present a variant of the BACKWARD-CROSS-SAMPLING algorithm for the approximate swap matching problem, which works in $\mathcal{O}(nm^2)$-time and requires $\mathcal{O}(m)$-space. Its bit-parallel implementation, as opposed to the BP-APPROXIMATE-CROSS-SAMPLING algorithm, does not add any space overhead and maintains a worst-case $\mathcal{O}(nm)$-time and $\mathcal{O}(\sigma)$-space complexity, when the pattern length is $\mathcal{O}(w)$.

5.1.1 Preliminary definitions

Definition 5.1. *A* swap permutation *for a string P of length m is a permutation $\pi : \{0, ..., m-1\} \rightarrow \{0, ..., m-1\}$ 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 are swapped);

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

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

Definition 5.2. *Given a text T of length n and a pattern P of length m, P is said to* swap-match *(or to have a* swapped occurrence*) at location $j \geq m-1$ of T if there exists a swap permutation π of P such that $\pi(P)$ matches T at location j, i.e., $\pi(P) = T[j-m+1\,..\,j]$. In such a case we write $P \propto T_j$.*

As already observed, if a pattern P of length m has a swap match ending at location j of a text T, then the number k of swaps needed to transform P into its swapped version $\pi(P) = T[j - m + 1 \mathinner{..} j]$ is equal to half the number of mismatches of P at location j. Thus the value of k lies between 0 and $\lfloor m/2 \rfloor$.

Definition 5.3. *Given a text T of length n and a pattern P of length m, P is said to* swap-match *(or to have a* swapped occurrence*) at location j of T with* k swaps *if there exists a swap permutation π of P such that $\pi(P)$ matches T at location j and $k = |\{i : P[i] \neq P[\pi(i)]\}|/2$. In such a case we write $P \propto_k T_j$.*

Definition 5.4 (Pattern matching with swaps problem)**.** *Given a text T of length n and a pattern P of length m, find all locations $j \in \{m - 1, ..., n - 1\}$ such that P swap-matches with T at location j, i.e., $P \propto T_j$.*

Definition 5.5 (Approximate pattern Matching with swaps problem)**.** *Given a text T of length n and a pattern P of length m, find all pairs (j, k), with $j \in \{m - 1 ... n - 1\}$ and $0 \leq k \leq \lfloor m/2 \rfloor$, such that P has a swapped occurrence in T at location j with k swaps, i.e., $P \propto_k T_j$.*

The following elementary result will be used later.

Lemma 5.1 ([20])**.** *Let P and R be strings of length m over an alphabet Σ and suppose that there exists a swap permutation π such that $\pi(P) = R$. Then π is unique.*

Proof. Suppose, by way of contradiction, that there exist two different swap permutations π and π' such that $\pi(P) = \pi'(P) = R$. Then there must exist an index i such that $\pi(i) \neq \pi'(i)$. Without loss of generality, let us assume that $\pi(i) < \pi'(i)$ and suppose that i be the smallest index such that $\pi(i) \neq \pi'(i)$. Since $\pi(i), \pi'(i) \in \{i - 1, i, i + 1\}$, by Definition 5.1(b), it is enough to consider the following three cases:

Case 1: $\pi(i) = i - 1$ and $\pi'(i) = i$.

Then, by Definition 5.1(a), we have $\pi(i - 1) = i$, so that $P[\pi(i - 1)] = P[i] = P[\pi'(i)] = P[\pi(i)]$, thus violating Definition 5.1(c).

Case 2: $\pi(i) = i$ and $\pi'(i) = i + 1$.

Since by Definition 5.1(a) we have $\pi'(i + 1) = i$, then $P[\pi'(i + 1)] = P[i] = P[\pi(i)] = P[\pi'(i)]$, thus violating again Definition 5.1(c).

Case 3: $\pi(i) = i - 1$ and $\pi'(i) = i + 1$.

By Definition 5.1(c) we have $\pi(i-1) = \pi'(i+1) = i$. Thus $\pi'(i-1) \neq i = \pi(i-1)$, contradicting the minimality of i. □

Corollary 5.1. *Given a text T of length n and a pattern P of length m, if $P \propto T_j$, for a given position $j \in \{m-1, \ldots, n-1\}$, then there exists a unique swapped occurrence of P in T ending at position j.* □

5.1.2 The Approximate-Cross-Sampling algorithm

The Approximate-Cross-Sampling algorithm [20] computes the swap occurrences of all prefixes of a pattern P (of length m) in continuously increasing prefixes of a text T (of length n), using a dynamic programming approach. Additionally, for each occurrence of P in T, the algorithm computes also the number of swaps necessary to convert the pattern into its swapped occurrence.

In particular, during its $(j+1)$-th iteration, for $j = 0, 1, \ldots, n-1$, it is established whether $P_i \propto_k T_j$, for each $i = 0, 1, \ldots, m-1$, by exploiting information gathered during previous iterations as is described below.

Let us put

$$\begin{aligned} \bar{\mathcal{S}}_j &=_{\text{Def}} \{(i,k) \mid 0 \leq i \leq m-1 \text{ and } P_i \propto_k T_j\} \\ \bar{\lambda}_{j,i} &=_{\text{Def}} \begin{cases} \{(0,0)\} & \text{if } P[i] = T[j] \\ \emptyset & \text{otherwise}, \end{cases} \end{aligned}$$

for $0 \leq j \leq n-1$, and

$$\bar{\mathcal{S}}'_j =_{\text{Def}} \{(i,k) \mid 0 \leq i < m-1 \text{ and } (P_{i-1} \propto_k T_{j-1} \vee i = 0) \text{ and } P[i+1] = T[j]\},$$

for $1 \leq j < n-1$. Then the following recurrences hold:

$$\begin{aligned} \bar{\mathcal{S}}_{j+1} = {} & \{(i,k) \mid i \leq m-1 \text{ and } ((i-1,k) \in \bar{\mathcal{S}}_j \text{ and } P[i] = T[j+1]) \text{ or} \\ & ((i-1,k-1) \in \bar{\mathcal{S}}'_j \text{ and } P[i-1] = T[j+1]) \} \cup \bar{\lambda}_{j+1,0} \\ \bar{\mathcal{S}}'_{j+1} = {} & \{(i,k) \mid i < m-1 \text{ and } (i-1,k) \in \bar{\mathcal{S}}_j \text{ and } P[i+1] = T[j+1]\} \cup \bar{\lambda}_{j+1,1}\,. \end{aligned} \tag{5.1}$$

where the base cases are given by $\mathcal{S}_0 = \bar{\lambda}_{0,0}$ and $\mathcal{S}'_{0,0} = \bar{\lambda}_{0,1}$.

Such relations allow one to compute the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ in an iterative fashion, as shown in Fig. 5.1. Observe that $\bar{\mathcal{S}}_{j+1}$ is computed in terms of both $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$, whereas $\bar{\mathcal{S}}'_{j+1}$ needs only $\bar{\mathcal{S}}_j$ for its computation. The code of the Approximate-

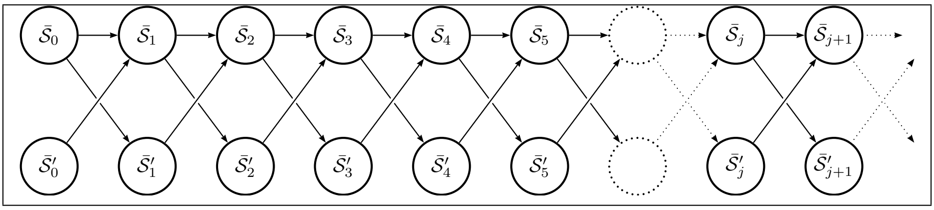


Figure 5.1: A graphic representation of the iterative fashion for computing sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ for increasing values of j.

Cross-Sampling algorithm is shown in Fig. 5.2 (left). The time complexity of the Approximate-Cross-Sampling algorithm is $\mathcal{O}(nm)$.

In [20] a bit-parallel implementation of the Approximate-Cross-Sampling algorithm, called BP-Approximate-Cross-Sampling, has been presented.

The BP-Approximate-Cross-Sampling algorithm uses a representation of the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ as bit-vectors of qm bits, $\bar{D}_j$ and $\bar{D}'_j$ respectively, where m is the length of the pattern and $q = \log(\lfloor m/2 \rfloor + 1) + 1$. If $(i, k) \in \bar{\mathcal{S}}_j$, where $0 \leq i < m$ and $0 \leq k \leq \lfloor m/2 \rfloor$, then the rightmost bit of the i-th block of $\bar{D}_j$ is set to 1 and the leftmost $q - 1$ bits of the i-th block correspond to the value k (we need exactly q bits to represent a value between 0 and $\lfloor m/2 \rfloor$). The same considerations hold for the sets $\bar{\mathcal{S}}'_j$.

For each character c of the alphabet Σ, the algorithm maintains a bit mask $M[c]$, where the rightmost bit of the i-th block is set to 1 if $P[i] = c$, and a bit mask $B[c]$, whose i-th block have all its bits set to 1 if $P[i] = c$.

The algorithm also maintains two bit vectors, $\bar{D}$ and $\bar{D}'$, whose configurations during the computation are denoted $\bar{D}_j$ and $\bar{D}'_j$ respectively, as the location j advances over the input text. For convenience, we introduce also the bit vectors $\bar{D}_{-1}$ and $\bar{D}'_{-1}$, which are both set to 0^{qm}.

While scanning the text from left to right, the algorithm computes for each position $j \geq 0$ the bit vector $\bar{D}_j$ in terms of $\bar{D}_{j-1}$ and $\bar{D}'_{j-1}$, by performing the following bitwise operations:

$$\begin{array}{ll}
(a) & \bar{D}_j \leftarrow (\bar{D}_{j-1} \ll q) \mid 1 \\
(b) & \bar{D}_j \leftarrow \bar{D}_j \ \& \ B[T[j]] \\
(c) & \bar{D}_j \leftarrow \bar{D}_j \mid ((\bar{D}'_{j-1} \ \& \ B[T[j]]) \ll q) \\
(d) & \bar{D}_j \leftarrow \bar{D}_j \ + \ (((\bar{D}'_{j-1} \ \& \ M[T[j]]) \ll q) \ll 1)
\end{array}$$

corresponding respectively to the relations:

$$\begin{array}{rl}(a) & \bar{\mathcal{S}}_j = \{(i,k) : (i-1,k) \in \bar{\mathcal{S}}_{j-1}\} \cup \{(0,0)\} \\ (b) & \bar{\mathcal{S}}_j \leftarrow \bar{\mathcal{S}}_j \setminus \{(i,k) : P[i] \neq T[j]\} \\ (c-d) & \bar{\mathcal{S}}_j \leftarrow \bar{\mathcal{S}}_j \cup \{(i,k+1) : (i-1,k) \in \bar{\mathcal{S}}'_{j-1} \wedge P[i-1] = T[j]\}\end{array}$$

Similarly, the bit vector $\bar{D}'_j$ is computed in the j-th iteration of the algorithm in terms of $\bar{D}_{j-1}$, by performing the following bitwise operations:

$$\begin{array}{rl}(a) & \bar{D}'_j \leftarrow (\bar{D}_{j-1} \ll q) \mid 1 \\ (b) & \bar{D}'_j \leftarrow \bar{D}'_j \ \& \ (B[T[j]] \gg q) \\ (c) & \bar{D}'_j \leftarrow \bar{D}'_j \ \& \ \sim \bar{D}_j\end{array}$$

corresponding respectively to the relations:

$$\begin{array}{rl}(a) & \bar{\mathcal{S}}'_j \leftarrow \{(i,k) : (i-1,k) \in \bar{\mathcal{S}}_{j-1}\} \cup \{(0,0)\} \\ (b) & \bar{\mathcal{S}}'_j \leftarrow \bar{\mathcal{S}}'_j \setminus \{(i,k) : P[i+1] \neq T[j]\} \\ (c) & \bar{\mathcal{S}}'_j \leftarrow \bar{\mathcal{S}}'_j \setminus \{(i,k) : (i,k) \in \bar{\mathcal{S}}_j\}\end{array}$$

During the j-th iteration, if the rightmost bit of the $(m-1)$-th block of $\bar{D}_j$ is set to 1, i.e., if $(\bar{D}_j \ \& \ \mathsf{1}\mathsf{0}^{q(m-1)}) \neq \mathsf{0}^m$, a swap match is reported at position j. The total number of swaps is contained in the $q-1$ leftmost bits of the $(m-1)$-th block of $\bar{D}_j$, which can be retrieved by performing a bitwise shift on $\bar{D}_j$ of $(q(m-1)+1)$ positions to the right.

The code of the BP-Approximate-Cross-Sampling algorithm is shown in Fig. 5.2 (right). It achieves a $\mathcal{O}(n\lceil m \log m/w\rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma\lceil m \log m/w\rceil)$ extra space, where σ is the size of the alphabet. If $m(\log(\lfloor m/2\rfloor+1)+1) = \mathcal{O}(w)$ then the algorithm requires $\mathcal{O}(n)$-time and $\mathcal{O}(\sigma)$ extra space.

5.1.3 New algorithms for the approximate swap matching problem

In this section we present a new practical algorithm for solving the swap matching problem, called Approximate-BCS (Approximate Backward Cross Sampling), which is characterized by a $\mathcal{O}(nm^2)$-time and $\mathcal{O}(m)$-space complexity, where m and n are the lengths of the pattern and text, respectively.

Our algorithm is an extension of the Backward-Cross-Sampling algo-

```
Approximate-Cross-Sampling $(P, m, T, n)$
$\bar{S}_0 \leftarrow \bar{S}'_0 \leftarrow \emptyset$
if $P[0] = T[0]$ then $\bar{S}_0 \leftarrow \{(0,0)\}$
if $1 < m$ and $P[1] = T[0]$
   then $\bar{S}'_0 \leftarrow \{(0,0)\}$
for $j \leftarrow 1$ to $n-1$ do
   $\bar{S}_j \leftarrow \bar{S}'_j \leftarrow \emptyset$
   for $(i,k) \in \bar{S}_{j-1}$ do
      if $i < m-1$ and $P[i+1] = T[j]$
         then $\bar{S}_j \leftarrow \bar{S}_j \cup \{(i+1,k)\}$
      if $i+1 < m-1$ and $P[i+2] = T[j]$
         then $\bar{S}'_j \leftarrow \bar{S}'_j \cup \{(i+1,k)\}$
   for $(i,k) \in \bar{S}'_{j-1}$ do
      if $i < m-1$ and $P[i] = T[j]$
         then $\bar{S}_j \leftarrow \bar{S}_j \cup \{(i+1,k+1)\}$
   if $P[0] = T[j]$ then $\bar{S}_j \leftarrow \bar{S}_j \cup \{(0,0)\}$
   if $1 < m$ and $P[1] = T[j]$
      then $\bar{S}'_j \leftarrow \bar{S}'_j \cup \{(0,0)\}$
   for $(i,k) \in \bar{S}_j$ do
      if $i = m-1$ then Output$(j,k)$
```

```
BP-Approximate-Cross-Sampling $(P, m, T, n)$
$q \leftarrow \log(\lfloor m/2 \rfloor + 1) + 1$
for $c \in \Sigma$ do $M[c] \leftarrow 0^{qm}$
for $c \in \Sigma$ do $B[c] \leftarrow 0^{qm}$
$j \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
   $M[P[i]] \leftarrow M[P[i]] \mid (0^{qm-1}1 \ll j)$
   $B[P[i]] \leftarrow B[P[i]] \mid (0^{q(m-1)}1^q \ll j)$
   $j \leftarrow j + q$
$\bar{D} \leftarrow \bar{D}' \leftarrow 0^{qm}$
for $j \leftarrow 0$ to $n-1$ do
   $H^0 \leftarrow (\bar{D} \ll q) \mid 1$
   $H^1 \leftarrow (\bar{D}' \,\&\, B[T[j]]) \ll q$
   $H^2 \leftarrow (\bar{D}' \,\&\, M[T[j]]) \ll q$
   $\bar{D} \leftarrow H^0 \,\&\, B[T[j]]$
   $\bar{D} \leftarrow (\bar{D} \mid H^1) + (H^2 \ll 1)$
   $\bar{D}' \leftarrow (H^0 \,\&\, (B[T[j]] \gg q)) \,\&\, \sim \bar{D}$
   if $\bar{D} \,\&\, 0^{q-1}10^{q(m-1)} \neq 0^{qm}$ then
      $k \leftarrow (\bar{D} \gg (q(m-1)+1))$
      Output$(j,k)$
```

Figure 5.2: The Approximate-Cross-Sampling algorithm (**left**) for the approximate swap matching problem and its bit-parallel variant BP-Approximate-Cross-Sampling (**right**).

rithm [16], for the standard swap matching problem. It inherits from the Approximate-Cross-Sampling algorithm the same doubly crossed structure in its iterative computation, but searches for all occurrences of the pattern in the text by scanning characters backwards, from right to left.

Later, in Section 5.1.3, we present an efficient implementation based on bit-parallelism of the Approximate-BCS algorithm, which achieves a $\mathcal{O}(nm)$-time and $\mathcal{O}(\sigma)$-space complexity, when $m = \mathcal{O}(w)$.

The Approximate-BCS Algorithm

The Approximate-BCS algorithm searches for all the swap occurrences of a pattern P (of length m) in a text T (of length n) using right-to-left scans in windows of size m, as in the Backward-DAWG-Matching (BDM) algorithm for the exact single pattern matching problem [30]. In addition, for each occurrence of P in T, the algorithm counts the number of swaps necessary to convert the pattern in its swapped occurrence.

As in the BDM algorithm, the Approximate-BCS algorithm processes the text in windows of size m. Each attempt is identified by the last position, j, of

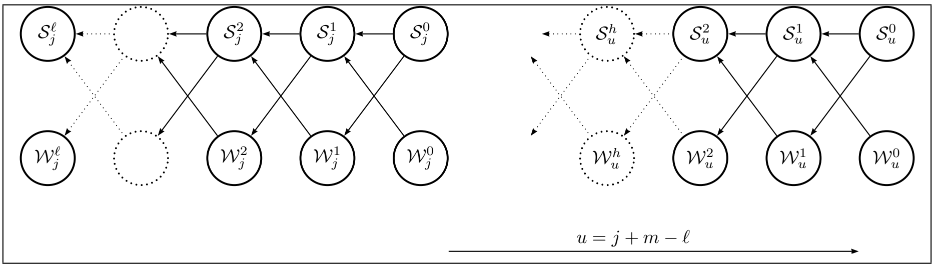


Figure 5.3: A graphic representation of the iterative fashion for computing the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ for increasing values of h. A first attempt starts at position j of the text and stops with $h = \ell$. The subsequent attempt starts at position $u = j + m - l$.

the current window of the text. The window is searched for the longest prefix of the pattern which has a swapped occurrence ending at position j of the text. At the end of each attempt, a new value of j is computed by performing a safe shift to the right of the current window in such a way to left-align it with the longest prefix matched in the previous attempt.

To this end, if we put

$$\begin{aligned}
\mathcal{S}_j^h &=_{\text{Def}} \{h-1 \leq i \leq m-1 \mid \; P[i-h+1 \mathinner{..} i] \propto T_j\}\,, \\
\mathcal{W}_j^h &=_{\text{Def}} \{h \leq i < m-1 \mid P[i-h+2 \mathinner{..} i] \propto T_j \;\text{ and }\; P[i-h] = T[j-h+1]\}\,,
\end{aligned}$$

for $0 \leq j < n$ and $0 \leq h \leq m$, then the following recurrences hold:

$$\begin{aligned}
\mathcal{S}_j^{h+1} &= \{h-1 \leq i \leq m-1 \mid (i \in \mathcal{S}_j^h \;\text{ and }\; P[i-h] = T[j-h]) \text{ or} \\
&\qquad\qquad\qquad\qquad (i \in \mathcal{W}_j^h \;\text{ and }\; P[i-h+1] = T[j-h])\,\} \\
\mathcal{W}_j^{h+1} &= \{h \leq i \leq m-1 \mid i \in \mathcal{S}_j^h \;\text{ and }\; P[i-h-1] = T[j-h]\}\,.
\end{aligned} \tag{5.2}$$

where the base cases are given by

$$\mathcal{S}_j^0 = \{i \mid 0 \leq i < m\} \quad \text{and} \quad \mathcal{W}_j^0 = \{0 \leq i < m \mid P[i] = T[j+1]\}\,.$$

Such relations allow one to compute the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ in an iterative fashion, where $\mathcal{S}_j^{h+1}$ is computed in terms of both $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$, whereas $\mathcal{W}_j^{h+1}$ needs only $\mathcal{S}_j^h$ for its computation. The structure is similar to the one of the Approximate-Cross-Sampling algorithm, as shown in Fig. 5.3.

Plainly the set $\mathcal{S}_j^h$ includes all the values i such that the h-substring of P

ending at position i has a swapped occurrence ending at position j in T. Thus, if $(h-1) \in \mathcal{S}_j^h$, then there is a swapped occurrence of the prefix of length h of P. Hence, it follows that P has a swapped occurrence ending at position j if and only if $(m-1) \in \mathcal{S}_j^m$.

Observe however that the only prefix of length m is the pattern P itself. Thus $(m-1) \in \mathcal{S}_j^m$ if and only if $\mathcal{S}_j^m \neq \emptyset$.

The following result follows immediately from (5.2).

Lemma 5.2. *Let P and T be a pattern of length m and a text of length n, respectively. Moreover let $m-1 \leq j \leq n-1$ and $0 \leq i < m$. If $i \in \mathcal{S}_j^\gamma$, then it follows that $i \in (\mathcal{S}_j^h \cup \mathcal{W}_j^h)$, for $1 \leq h \leq \gamma$.* □

Lemma 5.3. *Let P and T be a pattern of length m and a text of length n, respectively. Then, for every $m-1 \leq j \leq n-1$ and $0 \leq i < m$ such that $i \in (\mathcal{S}_j^\gamma \cap \mathcal{W}_j^{\gamma-1} \cap \mathcal{S}_j^{\gamma-1})$, we have $P[i-\gamma+1] = P[i-\gamma+2]$.*

Proof. From $i \in (\mathcal{S}_j^\gamma \cap \mathcal{S}_j^{\gamma-1})$ it follows that $P[i-\gamma+1] = T[j-\gamma+1]$. Also, from $i \in \mathcal{W}_j^{\gamma-1}$ it follows that $P[i-\gamma+2] = T[j-\gamma+1]$. Thus $P[i-\gamma+1] = P[i-\gamma+2]$. □

The following lemma will be used.

Lemma 5.4. *Let P and T be a pattern of length m and a text of length n, respectively. Moreover let $m-1 \leq j \leq n-1$ and $0 \leq i < m$. Then, if $i \in \mathcal{S}_j^\gamma$, there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$ if and only if $i \in (\mathcal{S}_j^\gamma \setminus \mathcal{S}_j^{\gamma-1})$.*

Proof. Before entering into details we remember that, by Definition 5.1, a swap can take place between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$ if and only if $P[i-\gamma+1] = T[j-\gamma+2]$, $P[i-\gamma+2] = T[j-\gamma+1]$ and $P[i-\gamma+1] \neq P[i-\gamma+2]$.

Now, suppose that $i \in \mathcal{S}_j^\gamma$ and that there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$. We proceed by contradiction to prove that $i \notin \mathcal{S}_j^{\gamma-1}$. Thus, we have

(i) $i \in \mathcal{S}_j^\gamma$ (by hypothesis)

(ii) $P[i-\gamma+2] = T[j-\gamma+1] \neq P[i-\gamma+1]$ (by hypothesis)

(iii) $i \in \mathcal{S}_j^{\gamma-1}$ (by contradiction)

(iv) $i \notin \mathcal{W}_j^{\gamma-1}$ (by (ii), (iii), and Lemma 5.3)

(v) $P[i-\gamma+1] = T[j-\gamma+1]$ (by (i) and (iv)) obtaining a contradiction between (ii) and (v).

Next, suppose that $i \in (\mathcal{S}_j^\gamma \setminus \mathcal{S}_j^{\gamma-1})$. We prove that there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$. We have

(i)	$i \in \mathcal{S}_j^\gamma$ and $i \notin \mathcal{S}_j^{\gamma-1}$	(by hypothesis)
(ii)	$i \in \mathcal{W}_j^{\gamma-1}$	(by (i) and Lemma 5.2)
(iii)	$i \in \mathcal{S}_j^{\gamma-2}$	(by (ii) and (5.2))
(iv)	$P[i-\gamma+1] = T[j-\gamma+2]$	(by (i) and (ii))
(v)	$P[i-\gamma+2] = T[j-\gamma+1]$	(by (ii))
(vi)	$P[i-\gamma+2] \neq T[j-\gamma+2] = P[i-\gamma+1]$	(by (i) and (iii)).

□

The following corollary is an immediate consequence of Lemmas 5.4 and 5.2.

Corollary 5.2. *Let P and T be strings of length m and n, respectively, over a common alphabet Σ. Then, for $m-1 \leq j \leq n-1$, P has a swapped occurrence in T at location j with k swaps, i.e., $P \propto_k T_j$, if and only if*

$$(m-1) \in \mathcal{S}_j^m \quad \textit{and} \quad |\Delta_j| = k\,,$$

where $\Delta_j = \{1 \leq h < m \;:\; (m-1) \;\in\; (\mathcal{S}_j^{h+1} \setminus \mathcal{S}_j^h)\}$. □

In consideration of the preceding corollary, the Approximate-BCS algorithm maintains a counter which is incremented every time $(m-1) \in (\mathcal{S}_j^{h+1} \setminus \mathcal{S}_j^h)$, for any $1 < h \leq m$, in order to count the swaps for an occurrence ending at a given position j of the text.

For any attempt at position j of the text, let us denote by ℓ the length of the longest prefix matched in the current attempt. Then the algorithm starts its computation with $j = m-1$ and $\ell = 0$. During each attempt, the window of the text is scanned from right to left, for $h = 1, \ldots, m$. If, for a given value of h, the algorithm discovers that $(h-1) \in \mathcal{S}_j^h$, then ℓ is set to the value h.

The algorithm is not able to remember the characters read in previous iterations. Thus, an attempt ends successfully when h reaches the value m (a match is found), or unsuccessfully when both sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ are empty. In any case, at the end of each attempt, the starting position of the window, i.e., position $j-m+1$ in the text, can be shifted to the starting position of the longest proper prefix detected during the backward scan. Thus the window is advanced by $m-\ell$ positions to the right. Observe that since $\ell < m$, we have $m - \ell > 0$.

The code of the Approximate-BCS algorithm is shown in Fig. 5.4 (left). Its time complexity is $\mathcal{O}(nm^2)$ in the worst case and requires $\mathcal{O}(m)$ extra space to represent the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$.

The Approximate-BPBCS Algorithm

In [16] an efficient bit-parallel implementation of the Backward-Cross-Sampling algorithm, called BP-Backward-Cross-Sampling, has also been presented. Here we illustrate a practical bit-parallel implementation of the Approximate-BCS algorithm, named Approximate-BPBCS, along the same lines of the BP-Backward-Cross-Sampling algorithm.

In the Approximate-BPBCS algorithm, the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ are represented as bit-vectors of m bits, D_j^h and C_j^h respectively, where m is the length of the pattern.

The $(i-h+1)$-th bit of D_j^h is set to 1 if $i \in \mathcal{S}_j$, i.e., if $P[i-h+1\,..\,i] \propto T_j$, whereas the $(i-h+1)$-th bit of C_j^h is set to 1 if $i \in \mathcal{W}_j^h$, i.e., if $P[i-h+2\,..\,i] \propto T_j$ and $P[i-h] = T[j-h+1]$. All remaining bits are set to 0.

For each character c of the alphabet Σ, the algorithm maintains a bit mask $M[c]$ whose i-th bit is set to 1 if $P[i] = c$.

As in the Approximate-BCS algorithm, the text is processed in windows of size m, identified by their last position j, and the first attempt starts at position $j = m-1$. For any searching attempt at location j of the text, the bit vectors D_j^1 and C_j^1 are initialized to $M[T[j]] \mid (M[T[j+1]] \;\&\; (M[T[j]] \ll 1))$ and $M[T[j]] \gg 1$, respectively, according to the recurrences (5.2) and relative base cases. Then the current window of the text, i.e., $T[j-m+1\,..\,j]$, is scanned from right to left, by reading character $T[j-h+1]$, for increasing values of h. Namely, for each value of $h > 1$, the bit vector D_j^{h+1} is computed in terms of D_j^h and C_j^h, by performing the following bitwise operations:

$$\begin{array}{ll} (a) & D_j^{h+1} \leftarrow (D_j^h \ll 1) \;\&\; M[T[j-h]] \\ (b) & D_j^{h+1} \leftarrow D_j^{h+1} \mid ((C_j^h \;\&\; M[T[j-h]]) \ll 1)\,. \end{array}$$

Concerning (a), by a left shift of D_j^h, all the elements of $\mathcal{S}_j^h$ are added to the set $\mathcal{S}_j^{h+1}$. Then, by performing a bitwise and with the mask $M[T[j-h]]$, all elements i such that $P[i-h] \neq T[j-h]$ are removed from $\mathcal{S}_j^{h+1}$. Similarly, the bit operations in (b) have the effect of adding to $\mathcal{S}_j^{h+1}$ all the elements i in $\mathcal{W}_j^h$

such that $P[i-h+1] = T[j-h]$. Formally, we have:

$$\begin{array}{ll} (a') & \mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^h \setminus \{i \in \mathcal{S}_j^h \ : \ P[i-h] \neq T[j-h]\} \\ (b') & \mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \mathcal{W}_j^h \setminus \{i \in \mathcal{W}_j^h \ : \ P[i-h+1] \neq T[j-h]\}\,. \end{array}$$

Similarly, the bit vector C_j^{h+1} is computed in terms of D_j^h, by performing the following bitwise operations

$$(c) \quad C_j^{h+1} \leftarrow (D_j^h \ll 1) \ \& \ (M[T[j-h]] \gg 1)$$

which have the effect of adding to the set $\mathcal{W}_j^{h+1}$ all the elements of the set $\mathcal{S}_j^h$ (by shifting D_j^h to the left by one position) and of removing all the elements i such that $P[i-h-1] \neq T[j-h]$ holds (by a bitwise and with the mask $M[T[j-h]] \gg 1$), or, more formally:

$$(c') \quad \mathcal{W}_j^{h+1} \leftarrow \mathcal{S}_j^h \setminus \{i \in \mathcal{S}_j^h \ : \ P[i-h-1] \neq T[j-h]\}\,.$$

In order to count the number of swaps, observe that the $(i-h+1)$-th bit of D_j^h is set to 1 if $i \in \mathcal{S}_j^h$. Thus, the condition $(m-1) \in (\mathcal{S}_j^{h+1} \setminus \mathcal{S}_j^h)$ can be implemented by the following bitwise condition:

$$(d) \quad ((D^{h+1} \ \& \ \sim (D^h \ll 1)) \ \& \ (1 \ll h)) \neq 0\,.$$

As in the Approximate-BCS algorithm, an attempt ends when $h = m$ or $(D_j^h | C_j^h) = 0$. If $h = m$ and $D_j^h \neq 0$, a swap match at position j of the text is reported. In any case, if $h < m$ is the largest value such that $D_j^h \neq 0$, then a prefix of the pattern of length $\ell = h$, which has a swapped occurrence ending at position j of the text, has been found. Thus, a safe shift of $m - \ell$ positions to the right can take place.

In practice, two vectors only are enough to implement the sets D_j^h and C_j^h, for $h = 0, 1, \ldots, m$, as one can transform the vector D_j^h into the vector D_j^{h+1} and the vector C_j^h into the vector C_j^{h+1}, during the h-th iteration of the algorithm at a given location j of the text.

The counter for taking note of the number of swaps requires $\log(\lfloor m/2 \rfloor + 1)$ bits to be implemented. This compares favorably with the BP-Approximate-Cross-Sampling algorithm which uses instead m counters of $\log(\lfloor m/2 \rfloor + 1)$ bits, one for each prefix of the pattern.

The resulting Approximate-BPBCS algorithm is shown in Fig. 5.4 (right). It achieves a $\mathcal{O}(\lceil nm^2/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma \lceil m/w \rceil +$

```
Approximate-BCS $(P, m, T, n)$
j ← m − 1
while $j < n$ do
    $h \leftarrow 0, l \leftarrow 0$
    $\mathcal{S}_j^0 \leftarrow \{i \mid 0 \leq i < m\}$
    $\mathcal{W}_j^0 \leftarrow \{0 \leq i < m \mid P[i] = T[j+1]\}$
    $c \leftarrow 0$
    while $h < m$ and $\mathcal{S}_j^h \cup \mathcal{W}_j^h \neq \emptyset$ do
        $\mathcal{S}_j^{h+1} \leftarrow \mathcal{W}_j^{h+1} \leftarrow \emptyset$
        if $(h-1) \in \mathcal{S}_j^h$ then $\ell \leftarrow h$
        for $i \in \mathcal{S}_j^h$ do
            if $i \geq h$ and $P[i-h] = T[j-h]$
                then $\mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \{i\}$
            if $i > h$ and $P[i-h-1] = T[j-h]$
                then $\mathcal{W}_j^{h+1} \leftarrow \mathcal{W}_j^{h+1} \cup \{i\}$
        for $i \in \mathcal{W}_j^h$ do
            if $i \geq h$ and $P[i-h+1] = T[j-h]$
                then $\mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \{i\}$
        if $m-1 \in \mathcal{S}_j^{h+1}$ and $m-1 \notin \mathcal{S}_j^h$
            then $c \leftarrow c+1$
        $h \leftarrow h+1$
    if $(h-1) \in \mathcal{S}_j^h$ then Output($j$,$c$)
    $j \leftarrow j+m-\ell$
```

```
Approximate-BPBCS $(P, m, T, n)$
for $c \in \Sigma$ do $M[c] \leftarrow 0^m$
for $i \leftarrow 0$ to $m-1$ do
    $c \leftarrow P[m-1-i]$
    $M[c] \leftarrow M[c] \mid (0^{m-1}1 \ll i)$
$j \leftarrow m-1$
while $j < n$ do
    $h \leftarrow 1, \ell \leftarrow 0$
    $H \leftarrow M[T[j+1]]$ & $(M[T[j]] \ll 1)$
    $D \leftarrow M[T[j]] \mid H$
    $C \leftarrow M[T[j]] \gg 1$
    $c \leftarrow 0$
    while $h < m$ and $(D \mid C) \neq 0^m$ do
        if $D$ & $10^{m-1} \neq 0^m$ then $\ell \leftarrow h$
        $D' \leftarrow D \ll 1$
        $H \leftarrow (C$ & $M[T[j-h]]) \ll 1$
        $D \leftarrow D'$ & $M[T[j-h]]$
        $D \leftarrow D \mid H$
        $C \leftarrow D'$ & $(M[T[j-h]] \gg 1)$
        if $(D$ & $\sim D')$ & $(0^{m-1}1 \ll h) \neq 0$
            then $c \leftarrow c+1$
        $h \leftarrow h+1$
    if $D \neq 0^m$ then Output($j$,$c$)
    $j \leftarrow j+m-\ell$
```

Figure 5.4: The Approximate-BCS algorithm (**left**) for the approximate swap matching problem and its bit-parallel variant Approximate-BPBCS (**right**).

$\log(\lfloor m/2 \rfloor + 1))$ extra space, where σ is the alphabet size. If $m = \mathcal{O}(w)$, then the algorithm finds all the swapped occurrences of the pattern and their corresponding number of swaps in $\mathcal{O}(nm)$ time and $\mathcal{O}(\sigma)$ extra space.

5.1.4 Experimental evaluation

Next we report and comment on the experimental results relative to an extensive comparison, under various conditions, of the following approximate swap matching algorithms:

- Approximate-Cross-Sampling (ACS)
- BP-Approximate-Cross-Sampling (BPACS)
- Approximate-BCS (ABCS)
- Approximate-BPBCS (BPABCS)
- Iliopoulos-Rahman algorithm with a naive check of the swaps (IR*)

Algorithm	4	8	12	16	20	24	28	32
ACS	385.66	387.14	383.95	392.69	383.95	387.09	385.80	384.28
ABCS	420.39	297.81	248.44	229.37	212.75	204.41	199.82	190.85
BPACS	44.72	41.37	41.41	41.95	42.00	42.02	41.99	43.13
BPABCS	27.37	14.93	**10.24**	**8.15**	**6.75**	**5.75**	**5.07**	**4.52**
IR*	**13.77**	**12.29**	12.21	12.22	12.21	12.21	12.21	12.21
BPBCS*	30.34	15.54	10.57	8.32	6.88	5.86	5.18	4.62

Table 5.1: Running times (ms) for a Rand4 problem.

Algorithm	4	8	12	16	20	24	28	32
ACS	338.09	337.75	337.33	344.72	342.72	342.82	343.48	338.19
ABCS	280.04	206.02	176.68	157.63	147.02	138.32	131.98	127.70
BPACS	41.71	41.35	41.39	41.41	41.38	41.40	41.35	41.48
BPABCS	19.96	**11.36**	**7.85**	**6.08**	**4.94**	**4.18**	**3.66**	**3.24**
IR*	**12.32**	12.22	12.22	12.22	12.20	12.20	12.21	12.20
BPBCS*	20.40	11.63	8.04	6.19	5.01	4.23	3.68	3.24

Table 5.2: Running times (ms) for a Rand8 problem.

- BP-Backward-Cross-Sampling algorithm with a naive check of the swaps (BPBCS*)

We have also included in our comparison the algorithms IR* and BPBCS*, since the algorithms IR and BPBCS turned out to be the most efficient solutions for the swap matching problem [16]. Instead, the Naive algorithm and algorithms based on the FFT technique have not been taken into consideration as their overhead is quite high.

The tests have been performed on a 2.33 GHz Intel Core 2 Duo. In particular, all the algorithms have been tested on six Randσ problems, for $\sigma = 4, 8, 16, 32, 64$, and 128, and on files (i), (iii) and (iv) (see Section 2.6) with patterns of length m = 4, 8, 12, 16, 20, 24, 28, 32. Each Randσ problem consists in searching a set of 100 random patterns for a given length value in a 4Mb random text over a common alphabet of size σ, with a uniform character distribution. In the following tables, the running times are expressed in milliseconds and the best results have been bold-faced.

The experimental results show that the BPABCS algorithm obtains the best performance in most cases. The only exception is found in the case of very short patterns and small alphabets, where the IR* algorithm is faster. For long patterns the difference between the BPABCS algorithm and the BPBCS* algorithm is small,

Algorithm	4	8	12	16	20	24	28	32
ACS	317.06	317.28	317.29	324.95	322.67	325.98	324.82	317.22
ABCS	222.69	167.56	142.81	128.57	119.03	112.56	106.97	102.99
BPACS	41.36	41.32	41.37	41.35	41.34	41.37	41.32	41.38
BPABCS	13.34	**9.09**	**6.68**	**5.15**	**4.11**	**3.47**	**2.96**	**2.60**
IR*	**12.22**	12.21	12.20	12.20	12.20	12.20	12.20	12.21
BPBCS*	13.60	9.33	6.85	5.29	4.22	3.57	3.05	2.68

Table 5.3: Running times (ms) for a Rand16 problem.

Algorithm	4	8	12	16	20	24	28	32
ACS	340.75	325.60	335.11	322.18	315.97	348.98	332.09	336.83
ABCS	201.47	154.68	123.63	112.35	104.85	102.78	95.25	91.88
BPACS	41.36	41.36	41.33	41.34	41.32	41.36	41.36	41.32
BPABCS	**10.22**	**6.62**	**5.23**	**4.35**	**3.73**	**3.28**	**2.89**	**2.53**
IR*	13.10	12.45	12.46	12.81	13.38	12.85	12.45	12.98
BPBCS*	11.46	7.51	5.59	4.45	3.85	3.54	2.99	2.75

Table 5.4: Running times (ms) for a Rand32 problem.

Algorithm	4	8	12	16	20	24	28	32
ACS	304.73	308.63	304.70	314.56	320.50	326.09	331.66	329.80
ABCS	185.15	135.45	118.52	101.56	97.27	90.71	87.66	84.88
BPACS	41.36	41.36	41.33	41.31	41.32	41.36	41.36	41.34
BPABCS	**8.51**	**5.03**	**3.88**	**3.29**	**2.90**	**2.60**	**2.36**	**2.16**
IR*	14.59	14.58	14.61	14.60	14.59	14.59	14.60	14.60
BPBCS*	8.76	5.15	3.96	3.37	2.97	2.66	2.41	2.21

Table 5.5: Running times (ms) for a Rand64 problem.

Algorithm	4	8	12	16	20	24	28	32
ACS	303.10	302.91	302.77	356.32	356.05	356.27	356.02	302.91
ABCS	176.61	121.39	104.44	94.66	88.59	84.77	81.38	79.36
BPACS	41.32	41.33	41.32	41.32	41.37	41.33	41.39	41.32
BPABCS	**7.72**	**4.27**	**3.12**	**2.55**	**2.20**	**1.99**	**1.79**	**1.67**
IR*	15.50	15.51	15.48	15.48	15.49	15.49	15.51	15.51
BPBCS*	7.98	4.39	3.19	2.60	2.24	2.01	1.82	1.69

Table 5.6: Running times (ms) for a Rand128 problem.

Algorithm	4	8	12	16	20	24	28	32
ACS	441.41	447.77	453.67	446.16	452.23	449.74	442.72	452.46
ABCS	466.88	340.48	288.85	266.94	247.83	239.04	226.08	223.28
BPACS	52.29	48.00	48.09	48.73	48.72	48.79	48.69	50.12
BPABCS	32.25	17.07	**11.79**	**9.40**	**7.83**	**6.67**	**5.84**	**5.26**
IR*	**16.16**	**14.27**	14.18	14.19	14.19	14.18	14.19	14.19
BPBCS*	35.85	17.89	12.20	9.63	7.99	6.81	5.95	5.37

Table 5.7: Running times (ms) for a DNA sequence ($\sigma = 4$).

Algorithm	4	8	12	16	20	24	28	32
ACS	228.95	228.78	227.05	229.41	229.31	230.46	230.82	228.85
ABCS	161.95	115.47	101.37	92.35	85.70	80.10	76.97	73.70
BPACS	29.97	29.94	29.97	29.98	29.94	29.94	29.97	29.98
BPABCS	9.74	**6.28**	**4.68**	**3.66**	**3.00**	**2.51**	**2.16**	**1.91**
IR*	**8.83**	8.83	8.83	8.83	8.83	8.82	8.83	8.83
BPBCS*	9.94	6.43	4.79	3.76	3.09	2.57	2.22	1.96

Table 5.8: Running times (ms) for a protein sequence ($\sigma = 20$).

Algorithm	4	8	12	16	20	24	28	32
ACS	190.74	192.93	196.09	201.45	200.99	197.57	201.64	192.25
ABCS	124.98	95.45	85.95	75.93	69.92	65.31	64.56	61.19
BPACS	25.62	25.54	25.58	25.54	25.55	25.57	25.58	25.64
BPABCS	**7.63**	**4.97**	**4.04**	**3.08**	**2.65**	**2.29**	**2.00**	**1.79**
IR*	7.84	7.80	7.79	7.79	7.79	7.80	7.79	7.79
BPBCS*	7.77	5.07	4.14	3.14	2.70	2.35	2.04	1.83

Table 5.9: Running times (ms) for the CIA World Fact Book ($\sigma = 94$).

because the number of occurrences of the pattern and, consequently, the number of verifications performed by the BPBCS* algorithm decrease significantly as m grows. Observe also that the algorithms IR*, ACS, and BPACS maintain a linear behaviour whereas the algorithms ABCS and BPABCS show a sublinear trend.

5.2 Approximate string matching with inversions and translocations

In this section we investigate the approximate string matching problem under a string distance whose edit operations are translocations of equal length adjacent factors and inversions of factors. In particular, we present a $\mathcal{O}(nm \max(\alpha, \beta))$-time and $\mathcal{O}(m^2)$-space algorithm, where α and β are the maximum length of the factors involved in a translocation and in an inversion, respectively. Our algorithm is based on a dynamic-programming approach and makes use of the Directed Acyclic Word Graph of the pattern. The DAWG data structure has already been used in algorithms for the approximate string matching problem [66, 65] to keep track of the substrings of the pattern that match the text at every location. We show that, under the assumption of equiprobability and independence of characters in the alphabet, our algorithm has, on the average, a $\mathcal{O}(n \log_\sigma m)$-time complexity. Finally, we also present an efficient implementation of our algorithm, based on bit-parallelism, which has $\mathcal{O}(n \max(\alpha, \beta))$-time and $\mathcal{O}(\sigma + m)$-space complexity when the pattern length is comparable with the size of the computer word. To our knowledge there is no report in the literature of a similar formalization of the above problem.

5.2.1 Preliminary definitions

A *distance* $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ is a function which associates to any pair of strings X and Y the minimal cost of any finite sequence of edit operations which transforms X into Y, if such a sequence exists, ∞ otherwise. Edit operations have the form $Z \rightarrow_t W$, where $Z, W \in \Sigma^*$ and t is a nonnegative real number representing the cost. Any factor of X can be involved in at most one edit operation. If, for every operation $Z \rightarrow_t W$, there is also the symmetric operation $W \rightarrow_t Z$ (with the same cost), then the distance d is symmetric, i.e., $d(X, Y) = d(Y, X)$, for all $X, Y \in \Sigma^*$.

Definition 5.6. *Given two strings X and Y, the* mutation distance $md(X,Y)$ *is based on the following edit operations:*

(1) ***Translocation****: a factor of the form ZW is transformed into WZ, provided that*
$|Z| = |W| > 0$.

(2) ***Inversion****: a factor Z is tranformed into Z^r.*

Both operations are assigned unit cost. □

Observe that, by definition, the maximum length of the factors involved in a translocation is $\lfloor |X|/2 \rfloor$, whereas the length of the factors involved in an inversion can be at most $|X|$. Note, moreover, that there are strings X, Y such that X can not be converted into Y by any sequence of translocations and inversions, in which case $md(X,Y) = \infty$. When $md(X,Y) < \infty$, we say that X and Y have an md-match. Additionally, if X has an md-match with a suffix of Y, we write $X \sqsupseteq_{md} Y$.

5.2.2 An automaton-based approach for the pattern matching problem with translocations and inversions

We present an efficient algorithm, called M-Sampling, which finds the md-matches of a given pattern P (of length m) in a text T (of length n). Our algorithm, based on the dynamic programming approach, has a $\mathcal{O}(nm \max(\alpha, \beta))$-time and $\mathcal{O}(m^2)$-space complexity, where $\alpha \leq \lfloor m/2 \rfloor$ is a bound on the length of the factors involved in any translocation and $\beta \leq m$ is a bound on the length of the factors involved in any inversion.

Given P, T, m, n, α, and β as above, the M-Sampling algorithm iteratively computes, for $j = m-1, m, \dots, n-1$, all the prefixes of P which have an md-match with a suffix of T_j, by exploiting information gathered at previous iterations. For this purpose a set $\mathcal{S}_j$ defined by

$$\mathcal{S}_j =_{\text{Def}} \{0 \leq i \leq m-1 \mid P_i \sqsupseteq_{md} T_j\}.$$

is maintained. Thus, the pattern P has an md-match ending at position j of the text T if and only if $(m-1) \in \mathcal{S}_j$.

Since the allowed edit operations involve substrings of the pattern P, it is useful to introduce the set $\mathcal{F}_j^k$ of all the positions in P where an occurrence of the suffix of T_j of length k ends. More precisely, for $1 \leq k \leq \alpha$ and $k-1 \leq j < n$, we put

$$\mathcal{F}_j^k =_{\text{Def}} \{k-1 \leq i \leq m-1 \mid T[j-k+1\,..\,j] \sqsupseteq P_i\}\,.$$

Observe that $\mathcal{F}_j^k \subseteq \mathcal{F}_j^h$, for $1 \leq h \leq k \leq m$.

Similarly, to handle inversions, it is convenient to define the set $\mathcal{I}_j^k$ of the positions in P where an occurrence of the reverse of the suffix of T_j of length k ends. More precisely, for $1 \leq k \leq \beta$ and $k-1 \leq j < n$, we put

$$\mathcal{I}_j^k =_{\text{Def}} \{k-1 \leq i \leq m-1 \mid (T[j-k+1\,..\,j])^{\mathsf{r}} \sqsupseteq P_i\}\,.$$

The sets $\mathcal{S}_j$ can then be computed based on the following elementary recursion.

Lemma 5.5. *Let T and P be a text of length n and a pattern of length m, respectively. Then $i \in \mathcal{S}_j$, for $0 \leq i < m$ and $i \leq j < n$, if and only if one of the following three facts holds*

(a) $P[i] = T[j]$ and $(i-1) \in \mathcal{S}_{j-1} \cup \{-1\}$ (standard match);

(b) $(i-k) \in \mathcal{F}_j^k$, $i \in \mathcal{F}_{j-k}^k$, and $(i-2k) \in \mathcal{S}_{j-2k} \cup \{-1\}$, for some $1 \leq k \leq \lfloor \frac{i+1}{2} \rfloor$ (translocation);

(c) $i \in \mathcal{I}_j^k$ and $(i-k) \in \mathcal{S}_{j-k} \cup \{-1\}$, for some $1 \leq k \leq i+1$ (inversion). □

Conditions (b) and (c) refer to a translocation of adjacent factors of length k and an inversion of a factor of length k, respectively.

Likewise, the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$ can be computed according to the following lemma:

Lemma 5.6. *Let T and P be a text of length n and a pattern of length m, respectively. Then $i \in \mathcal{F}_j^k$, for $1 \leq k \leq \alpha$, $k-1 \leq i < m$, and $k-1 \leq j < n$, if and only if the following condition holds*

$$(k = 1 \text{ or } (i-1) \in \mathcal{F}_{j-1}^{k-1}) \text{ and } P[i] = T[j]\,.$$

Similarly, $i \in \mathcal{I}_j^k$, for $1 \leq k \leq \beta$, $k-1 \leq i < m$, and $k-1 \leq j < n$, if and only if the following condition holds

$$(k = 1 \text{ or } i \in \mathcal{I}_{j-1}^{k-1}) \text{ and } P[i-k+1] = T[j]\,.$$ □

Based on Lemmas 5.5 and 5.6, a general dynamic programming algorithm can be readily constructed, characterized by an overall $\mathcal{O}(nm\max(\alpha,\beta))$-time and $\mathcal{O}(m^2)$-space complexity. However, the overhead due to the computation of the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$ turns out to be quite large. By suitably preprocessing the pattern with the DAWG data structure, as will be described in the next section, the M-Sampling algorithm succeeds in drastically reducing such a overhead (see Fig. 5.7). The code of the algorithm M-Sampling is shown in Fig. 5.6.

Efficient computation of the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$

An efficient method for computing the sets $\mathcal{F}_j^k$ defined above, for $1 \leq k \leq \alpha$ and $k-1 \leq j < n$, makes use of the DAWG of the pattern P and of the function *end-pos*. Later we also show how to compute efficiently the sets $\mathcal{I}_j^k$.

Let $\mathscr{F}(P) = (Q, \Sigma, \delta, root, F)$ be the DAWG of P. For each position j in T, let P' be the longest factor of P, of length at most α, which is a suffix of T_j, let q_j be the state of $\mathscr{F}(P)$ such that $R_{\mathcal{P}}(P') = q_j$, and let l_j be the length of P'. We call the pair (q_j, l_j) a T-configuration of $\mathscr{F}(P)$. The idea is then to compute the T-configuration (q_j, l_j) of $\mathscr{F}(P)$, for each position j of the text, while scanning the text. The set $\mathcal{F}_j^k$ computed at previous iterations are not maintained explicitly; rather, only T-configurations are maintained. These are then used to compute efficiently the set $\mathcal{F}_j^k$ only when needed.

The longest factor of P ending at position j of T is computed in the same way as in the Forward-Dawg-Matching algorithm for the exact pattern matching problem (cf. [30]). Specifically the algorithm makes use of an improved suffix link function $s\ell^* : Q \rightarrow Q$, defined as

$$s\ell^*(q) = \begin{cases} suf(q) & \text{if } context(q) \neq context(suf(q)) \\ s\ell^*(suf(q)) & \text{otherwise} \end{cases}$$

where $context(q) = \{c \in \Sigma \mid \delta(q,c) \neq \textsc{nil}\}$ is the set of all characters which label an outgoing transition from state q. Rougly speaking, given a node q of the automaton, the improved suffix link function jumps directly to the first node in the suffix path which has a context different from $context(q)$.

Since we are interested in factors of length at most α, we maintain the invariant that the current state of the automaton never corresponds to factors longer than α (we discovered that a similar idea was used in [65]).

Let (q_{j-1}, l_{j-1}) be the T-configuration of $\mathscr{F}(P)$ at step $(j-1)$. Two cases

```
DAWG-DELTA$(q, l, k, c, \mathcal{B})$
if $l = k$ then
   $l \leftarrow l - 1$
   if $length(suf_{\mathcal{B}}(q)) = l$
      then $q \leftarrow suf_{\mathcal{B}}(q)$
if $\delta_{\mathcal{B}}(q, c) =$ NIL then
   do
      $q \leftarrow s\ell^*_{\mathcal{B}}(q)$
   while $q \neq$ NIL and $\delta_{\mathcal{B}}(q, c) =$ NIL
   if $q =$ NIL then
      $l \leftarrow 0, q \leftarrow root_{\mathcal{B}}$
   else $l \leftarrow length(q) + 1$
        $q \leftarrow \delta_{\mathcal{B}}(q, c)$
else $l \leftarrow l + 1$
        $q \leftarrow \delta_{\mathcal{B}}(q, c)$
return $(q, l)$
```

Figure 5.5: the DAWG state update algorithm.

must be distinguished.

Case $l_{j-1} < \alpha$**:** The new T-configuration (q_j, l_j) is set to $(\delta(q, T[j]), \mathit{length}(q) + 1)$, where q is the first node in the suffix path

$$(q_{j-1}, \mathit{suf}(q_{j-1}), \mathit{suf}^{(2)}(q_{j-1}), \ldots)$$

of q_{j-1}, including q_{j-1}, having a transition on $T[j]$, if such a node exists; otherwise (q_j, l_j) is set to $(\mathit{root}, 0)$.[2]

Case $l_{j-1} = \alpha$**:** We first compute the T-configuration corresponding to the factor $T[j - \alpha + 1 \mathinner{..} j - 1]$ of P of length $(\alpha - 1)$ ending at position $j - 1$ in T, namely the T-configuration (q'_{j-1}, l'_{j-1}), where

$$(q'_{j-1}, l'_{j-1}) =_{\text{Def}} \begin{cases} (\mathit{suf}(q_{j-1}), l_{j-1} - 1) & \text{if } \mathit{length}(\mathit{suf}(q_{j-1})) = l_{j-1} - 1 \\ (q_{j-1}, l_{j-1} - 1) & \text{otherwise.} \end{cases}$$

Then we compute the new T-configuration (q_j, l_j) starting from (q'_{j-1}, l'_{j-1}) as in the previous case, observing that $l'_{j-1} = \alpha - 1$. The algorithm to update the T-configuration of the DAWG $\mathscr{F}(P)$ is given in Fig. 5.5, where $s\ell^*$ denotes the improved suffix link [30].

[2] We recall that $\mathit{suf}^{(0)}(q) =_{\text{Def}} q$ and, recursively, $\mathit{suf}^{(h+1)}(q) =_{\text{Def}} \mathit{suf}(\mathit{suf}^{(h)}(q))$, for $h \geq 0$, provided that $\mathit{suf}^{(h)}(q) \neq \mathit{root}$.

Before explaining how to compute the sets $\mathcal{F}_j^k$, it is convenient to introduce a partial function $\phi : Q \times \mathbb{N} \to Q$ which, given a node $q \in Q$ and a length $k \leq \mathit{length}(q)$, computes the node $\phi(q,k)$ whose corresponding set of factors contains the suffix of $\mathit{val}(q)$ of length k. This is the same as saying, more formally, that $\phi(q,k)$ is the node $\mathit{suf}^{(i)}(q)$ such that

$$\mathit{length}(\mathit{suf}^{(i+1)}(q)) < k \leq \mathit{length}(\mathit{suf}^{(i)}(q)) \,,$$

for each $q \in Q$ and each integer $k \leq \mathit{length}(q)$. Roughly speaking, $\phi(q,k)$ is the first node p in the suffix path of q such that $\mathit{length}(\mathit{suf}(p)) < k$.

In the preprocessing phase, the DAWG $\mathscr{F}(P) = (Q, \Sigma, \delta, \mathit{root}, F)$ is computed together with the associated *end-pos* function. Since for a pattern P of length m we have that $|Q| \leq 2m+1$ and $|\mathit{end\text{-}pos}(q)| \leq m$, for each $q \in Q$, we need only $\mathcal{O}(m^2)$ extra space (see [11, 28]).

To compute the set $\mathcal{F}_j^k$, for $1 \leq k \leq l_j$, one can take advantage of the following relation

$$\mathcal{F}_j^k = \mathit{end\text{-}pos}(\phi(q_j,k)) \,. \tag{5.3}$$

Notice, in particular, that we have $\mathcal{F}_j^{l_j} = \mathit{end\text{-}pos}(q_j)$.

The time complexity of the computation of $\phi(q,k)$ can be bounded by the length of the suffix path of node q. Specifically, since the sequence

$$(\mathit{length}(\mathit{suf}^{(0)}(q)), \mathit{length}(\mathit{suf}^{(1)}(q)), \ldots, 0)$$

of the lengths of the nodes in the suffix path from q is strictly deacreasing, we can do at most $\mathit{length}(q)$ iterations over the suffix link, obtaining a $\mathcal{O}(m)$-time complexity.

According to Lemma 5.5, a translocation of length $2k$ at position j of the text T is possible only if factors of P of length at least k have been recognized at both positions j and $j-k$, namely if $l_j \geq k$ and $l_{j-k} \geq k$.

Let $\langle k_1, k_2, \ldots, k_r \rangle$ be the increasing sequence of all values k such that $1 \leq k \leq \min(l_j, l_{j-k})$. For each $1 \leq i \leq r$, condition (b) of Lemma 5.5 requires member queries on the sets $\mathcal{F}_j^{k_i}$ and $\mathcal{F}_{j-k_i}^{k_i}$.

We notice that, if we proceed for decreasing values of k, the sets $\mathcal{F}_j^k$, for $1 \leq k \leq l_j$, can be computed in constant time. Specifically, the set $\mathcal{F}_j^k$ can be computed in constant time from $\mathcal{F}_j^{k+1}$, for $k = 1, \ldots, l_j - 1$, with at most one iteration over the suffix link of the state $\phi(q_j, k+1)$.

The computation of $\mathcal{F}^{k_r}_{j-k_r}$ has a $\mathcal{O}(\alpha)$-time complexity, since $length(q_{j-k_r}) \leq \alpha$. To compute $\mathcal{F}^{k_i}_{j-k_i}$, for $i = r-1, r-2, \ldots, 1$, we distinguish the following two cases:

Case $k_{i+1} = k_i + 1$**:** Let $q' = \phi(q_{j-k_{i+1}}, k_{i+1})$. Given the node q' computed in the previous iteration, the node $\phi(q_{j-k_i}, k_i)$ can be computed in two steps: first, we look up the node corresponding to the suffix of length $k_{i+1} - 2$ of the factor represented by q', with at most two iterations of the suffix link of q'; then, we perform a transition on $T[j-k_i]$ on the node so found. Formally:

$$\phi(q_{j-k_i}, k_i) = \delta(\phi(q', k_{i+1} - 2), T[j - k_i])\,.$$

Case $k_{i+1} > k_i + 1$**:** Observe that $l_{j-s} \leq s - 1$ must hold, for each $s = k_{i+1} - 1, \ldots, k_i + 1$. In particular, we have $l_{j-(k_i+1)} \leq k_i$ which implies that $l_{j-k_i} \leq k_i + 1$ since $l_j \leq l_{j-1} + 1$ always holds. Hence, the computation of $\phi(q_{j-k_i}, k_i)$ requires at most one iteration of the suffix link of q_{j-k_i}.

Thus, in both cases, $\mathcal{F}^{k_i}_{j-k_i}$ can be computed in constant time, for $1 \leq i < r$. Therefore, the total complexity for computing all the sets $\mathcal{F}^{k_i}_{j-k_i}$, for $i = 1, \ldots, r$, is $\mathcal{O}(\alpha)$.

Next, to compute the sets $\mathcal{I}^k_j$ we use the DAWG $\mathscr{F}(P^{\mathsf{r}})$ of P^{r}. Specifically, we compute the longest reversed factor ending at j and maintain the invariant that the current state of the automaton never corresponds to factors longer than β, using algorithm given in Fig. 5.5, as for the computation of the sets $\mathcal{F}^k_j$. Let $(q^{\mathsf{r}}_j, l^{\mathsf{r}}_j)$ denote the T-configuration of $\mathscr{F}(P^{\mathsf{r}})$ after having read the character of T at position j, where l^{r}_j is the length of the longest reversed factor of P recognized. Then the sets $\mathcal{I}^k_j$ can be computed, for $2 \leq k \leq l^{\mathsf{r}}_j$, by

$$\mathcal{I}^k_j = \{i \mid (m - i + k - 2) \in \textit{end-pos}(\phi(q^{\mathsf{r}}_j, k))\}\,. \tag{5.4}$$

Indeed, $i \in \mathcal{I}^k_j$ iff $P[i-k+1 \mathinner{..} i] = (T[j-k+1 \mathinner{..} j])^{\mathsf{r}}$ iff

$$P^{\mathsf{r}}[(m-1) - i \mathinner{..} (m-1) - (i-k+1)] = (P[i-k+1 \mathinner{..} i])^{\mathsf{r}} = T[j-k+1 \mathinner{..} j]\,.$$

Thus (5.4) follows, as the latter is equivalent to $(m-i+k-2) \in \textit{end-pos}(\phi(q^{\mathsf{r}}_j, k))$.

For each $k = 1, \ldots, l^{\mathsf{r}}_j$, condition (c) of Lemma 5.5 requires member queries on the sets $\mathcal{I}^k_j$. As in the case of the sets $\mathcal{F}^k_j$, the set $\textit{end-pos}(\phi(q^{\mathsf{r}}_j, k))$ can be

```
M-Sampling $(P, T, \alpha, \beta, \mathscr{F}, \mathscr{F}')$
    /* $\mathscr{F}$ is the DAWG of $P$ and $\mathscr{F}'$ is the DAWG of $P^r$ */
    $m \leftarrow |P|$,    $n \leftarrow |T|$
    $(q_0, l_0) \leftarrow$ Dawg-Delta$(root_{\mathscr{F}}, 0, \alpha, T[0], \mathscr{F})$
    $(q^{\mathsf{r}}_0, l^{\mathsf{r}}_0) \leftarrow$ Dawg-Delta$(root_{\mathscr{F}'}, 0, \beta, T[0], \mathscr{F}')$
    $\mathcal{S}_0 \leftarrow \emptyset$
    if $P[0] = T[0]$ then $\mathcal{S}_0 \leftarrow \{0\}$
    for $j \leftarrow 1$ to $n - 1$ do
        $(q_j, l_j) \leftarrow$ Dawg-Delta$(q_{j-1}, l_{j-1}, \alpha, T[j], \mathscr{F})$
        $(q^{\mathsf{r}}_j, l^{\mathsf{r}}) \leftarrow$ Dawg-Delta$(q^{\mathsf{r}}_j, l^{\mathsf{r}}, \beta, T[j], \mathscr{F}')$
        $\mathcal{S}_j \leftarrow \emptyset$
        /* Standard matches */
        if $P[0] = T[j]$ then $\mathcal{S}_j \leftarrow \{0\}$
        for $i \in \mathcal{S}_{j-1}$ do
            if $i < m - 1$ and $P[i+1] = T[j]$ then
                $\mathcal{S}_j \leftarrow \mathcal{S}_j \cup \{i+1\}$
        /* Inversions */
        $p \leftarrow q^{\mathsf{r}}_j$
        for $k \leftarrow l^{\mathsf{r}}_j$ downto 2 do
            for $i \in \mathcal{S}_{j-k} \cup \{-1\}$ do
                if $(m - 2 - i) \in$ $end\text{-}pos^{\mathsf{r}}(p)$ then
                    $\mathcal{S}_j \leftarrow \mathcal{S}_j \cup \{i+k\}$
            if $k = length(suf_{\mathscr{F}'}(p)) + 1$ then
                $p \leftarrow suf_{\mathscr{F}'}(p)$
        /* Translocations */
        $last \leftarrow 0$
        $p \leftarrow q_j$
        for $k \leftarrow l_j$ downto 1 do
            if $k \leq j$ and $k \leq l_{j-k}$ then
                if $last = k + 1$ then
                    while $p' \neq root_{\mathscr{F}}$
                        and $k - 1 \leq length(suf_{\mathscr{F}}(p'))$ do
                            $p' \leftarrow suf_{\mathscr{F}}(p')$
                    $p' \leftarrow \delta_{\mathscr{F}}(p', T[j-k])$
                else
                    $p' \leftarrow q_{j-k}$
                    while $k \leq length(suf_{\mathscr{F}}(p'))$ do
                        $p' \leftarrow suf_{\mathscr{F}}(p')$
                $last \leftarrow k$
                for $i \in \mathcal{S}_{j-2k} \cup \{-1\}$ do
                    if $(i + k) \in end\text{-}pos(p)$
                        and $(i + 2k) \in end\text{-}pos(p')$ then
                            $\mathcal{S}_j \leftarrow \mathcal{S}_j \cup \{i + 2k\}$
            if $k = length(suf_{\mathscr{F}}(p)) + 1$
                then $p \leftarrow suf_{\mathscr{F}}(p)$
        if $(m - 1) \in \mathcal{S}_j$ then
            Output$(j)$
```

Figure 5.6: The M-Sampling algorithm for solving the pattern matching problem with translocations and inversions.

computed in constant time, in decreasing order of k, by iterating the suffix link on q^{r}_j. Although $\mathcal{I}^k_j$ is not equal to *end-pos*$(\phi(q^{\mathsf{r}}_j, k))$, a member query on $\mathcal{I}^k_j$ can still be done in constant time using (5.4).

5.2.3 Complexity analysis

We first analyze the worst-case time complexity of the M-Sampling algorithm and then its average-case complexity. Our analysis assumes that sets are implemented as bit vectors so that any member query on a set takes constant time. We shall also evaluate the space complexity of the M-Sampling algorithm.

Worst-case analysis

First of all, observe that the main **for**-loop at line 6 is always executed n times. Moreover, observe that $|\mathcal{S}_j| \leq m$, $l_j \leq \alpha$, and $l^{\mathsf{r}}_j \leq \beta$, for all $0 \leq j < n$. For each iteration of the **for**-loop at line 23, the amortized cost of the two **while**-loops at lines 26 and 31 is $\mathcal{O}(1)$. Thus, at each iteration of the main **for**-loop, the **for**-loop at line 11 takes at most $\mathcal{O}(m)$-time while the **for**-loops at lines 15 and 23 take at most $\mathcal{O}(m\beta)$- and $\mathcal{O}(m\alpha)$-time respectively. Summing up, the algorithm has a $\mathcal{O}(nm\max(\alpha,\beta))$ worst-case time complexity, which becomes $\mathcal{O}(nm^2)$-time when $\max(\alpha,\beta) = \Theta(m)$.

Average-case analysis

Next, we evaluate the average time complexity of the algorithm M-Sampling assuming a uniform distribution and the independence of characters.

Given integers $1 \leq \alpha, \beta \leq m \leq n$ and an alphabet Σ of size $\sigma \geq 4$, for $j = 0, 1, \dots, n-1$, we consider the following nonnegative random variables over the sample space of the pairs of strings $P, T \in \Sigma^*$ of length m and n, respectively:

- $X(j) =_{\mathrm{Def}}$ the length $l_j \leq \alpha$ of the longest factor of P which is a suffix of T_j,

- $Y(j) =_{\mathrm{Def}}$ the length $l^{\mathsf{r}}_j \leq \beta$ of the longest factor of P^{r} which is a suffix of T_j,

- $Z(j) =_{\mathrm{Def}}$ $|\mathcal{S}_j|$, where we recall that $\mathcal{S}_j = \{0 \leq i \leq m-1 \mid P_i \sqsupseteq_{md} T_j\}$.

Then the run-time of a call to the M-Sampling algorithm with parameters

(P, T, α, β) is proportional to

$$\sum_{j=1}^{n-1}\left(Z(j-1)+\sum_{k=2}^{Y(j)}Z(j-k)+\left(\sum_{k=1}^{X(j)}Z(j-2k)+X(j)\right)\right)\,, \tag{5.5}$$

where the external summation refers to the main **for**-loop (at line 6), and the three terms within it take care of the internal **for**-loops at lines 11, 15, and 23, in that order.

The average-case complexity of the M-Sampling algorithm is thus the expectation of (5.5), which, in view of the linearity of expectation, is equal to

$$\sum_{j=1}^{n-1}\left(E(Z(j-1))+E\left(\sum_{k=2}^{Y(j)}Z(j-k)\right)+E\left(\sum_{k=1}^{X(j)}Z(j-2k)\right)+E(X(j))\right)\,. \tag{5.6}$$

Since

$$\begin{array}{rcl} E(X(j)) & \leq & E(X(n-1)) \\ E(Y(j)) & \leq & E(Y(n-1)) \\ E(Z(j)) & \leq & E(Z(n-1))\,, \end{array}$$

for $0 \leq j \leq n-1$,[3] and also

$$E(X(n-1)) = E(Y(n-1))\,,$$

by putting

$$X \;=_{\text{Def}} X(n-1) \qquad \text{and} \qquad Z \;=_{\text{Def}} Z(n-1)\,,$$

expression (5.6) gets bounded from above by

$$\sum_{j=1}^{n-1}\left(E(Z)+E\left(\sum_{k=2}^{X}Z\right)+E\left(\sum_{k=1}^{X}Z\right)+E(X)\right)\,. \tag{5.7}$$

For $i = 0, \ldots, m-1$, let Z_i be the indicator variable

$$Z_i \;=_{\text{Def}} \begin{cases} 1 & \text{if } i \in \mathcal{S}_{n-1} \\ 0 & \text{otherwise}\,, \end{cases}$$

[3] In fact, for $j = m, \ldots, n-1$ all inequalities hold as equalities.

so that

$$Z = \sum_{i=0}^{m-1} Z_i \qquad \text{and} \qquad E(Z_i^2) = E(Z_i) = Pr\{P_i \sqsupseteq_{md} T\}\,.$$

Likewise, for $k = 1, \ldots, m$, let X_k be the indicator variable

$$X_k =_{\text{Def}} \begin{cases} 1 & \text{if } X \geq k \\ 0 & \text{otherwise}\,, \end{cases}$$

so that

$$X = \sum_{k=1}^{m} X_k \qquad \text{and} \qquad E(X_k^2) = E(X_k) = Pr\{X \geq k\}\,.$$

The we have

$$\sum_{k=1}^{X} Z = XZ = \left(\sum_{k=1}^{m} X_k\right) \cdot \left(\sum_{i=0}^{m-1} Z_i\right) = \sum_{k=1}^{m}\sum_{i=0}^{m-1} X_k Z_i\,.$$

Therefore

$$E\left(\sum_{k=2}^{X} Z\right) \leq E\left(\sum_{k=1}^{X} Z\right) = \sum_{k=1}^{m}\sum_{i=0}^{m-1} E(X_k Z_i)\ ,$$

yielding the following upper bound for (5.7):

$$\sum_{j=1}^{n-1}\left(E(Z) + 2 \cdot \sum_{k=1}^{m}\sum_{i=0}^{m-1} E(X_k Z_i) + E(X)\right)\,. \tag{5.8}$$

To estimate each of the terms $E(X_k Z_i)$ in (5.8), we use the well-known Cauchy-Schwarz inequality which in the context of expectations assumes the form

$$|E(UV)| \leq \sqrt{E(U^2)E(V^2)}\,,$$

for any two random variables U and V such that $E(U^2)$, $E(V^2)$ and $E(UV)$ are all finite.

Then, for $1 \leq k \leq m$ and $0 \leq i \leq m-1$, we have

$$E(X_k Z_i) \leq \sqrt{E(X_k^2)E(Z_i^2)} = \sqrt{E(X_k)E(Z_i)}\,. \tag{5.9}$$

From (5.9), it then follows that (5.8) is bounded from above by

$$\begin{aligned}\sum_{j=1}^{n-1}&\left(E(Z)+2\cdot\sum_{k=1}^{m}\sum_{i=0}^{m-1}\sqrt{E(X_k)E(Z_i)}+E(X)\right)\\&=\sum_{j=1}^{n-1}\left(E(Z)+2\cdot\left(\sum_{k=1}^{m}\sqrt{E(X_k)}\right)\cdot\left(\sum_{i=0}^{m-1}\sqrt{E(Z_i)}\right)+E(X)\right)\,.\end{aligned}\tag{5.10}$$

To better understand (5.10), we evaluate the expectations $E(X)$ and $E(Z)$ and the sums $\sum_{k=1}^{m}\sqrt{E(X_k)}$ and $\sum_{i=0}^{m-1}\sqrt{E(Z_i)}$. To this purpose, it will be useful to estimate also the expectations

- $E(X_k) = Pr\{X \geq k\}$, for $1 \leq k \leq m$, and
- $E(Z_i) = Pr\{P_i \sqsupseteq_{md} T\}$, for $0 \leq i \leq m-1$.

Concerning $E(X_k) = Pr\{X \geq k\}$, we reason as follows. Since $T[n-k\,..\,n-1]$ ranges uniformly over a collection of σ^k strings and there can be at most $\min(\sigma^k, m-k+1)$ distinct factors of length k in P, the probability $Pr\{X \geq k\}$ that one of them matches $T[n-k\,..\,n-1]$ is at most $\min\left(1, \frac{m-k+1}{\sigma^k}\right)$, so that, for $k = 1, \ldots, m$, we have

$$E(X_k) \leq \min\left(1, \frac{m-k+1}{\sigma^k}\right)\,. \tag{5.11}$$

Then, in view of (5.11), we have:

$$E(X) = \sum_{i=0}^{m} i\cdot Pr\{X = i\} = \sum_{i=1}^{m} Pr\{X \geq i\} \leq \sum_{i=1}^{m}\min\left(1, \frac{m-i+1}{\sigma^i}\right)\,. \tag{5.12}$$

Let $\overline{k}$ be the smallest integer $1 \leq k < m$ such that $\frac{m-k+1}{\sigma^k} < 1$. Then from (5.12) we have

$$\begin{aligned}E(X) &\leq \sum_{i=1}^{\overline{k}-1}1+\sum_{i=\overline{k}}^{m}\frac{m-i+1}{\sigma^i} \leq \overline{k}-1+(m-\overline{k}+1)\sum_{i=\overline{k}}^{m}\frac{1}{\sigma^i}\\&< \overline{k}-1+\frac{\sigma}{\sigma-1}\cdot\frac{m-\overline{k}+1}{\sigma^{\overline{k}}} < \overline{k}-1+\frac{\sigma}{\sigma-1} < \overline{k}+1\,.\end{aligned}\tag{5.13}$$

Since $\frac{m-(\overline{k}+1)+1}{\sigma^{\overline{k}+1}} \geq 1$, then $\sigma^{\overline{k}+1} \leq m-(\overline{k}+1)+1 \leq m-1$, so that

$$\overline{k}+1 < \log_\sigma m\,. \tag{5.14}$$

From (5.13) and (5.14), we obtain

$$E(X) < \log_\sigma m\,. \tag{5.15}$$

Likewise, from (5.11) and (5.14) we have

$$\begin{aligned}\sum_{k=1}^{m}\sqrt{E(X_k)} &\le \sum_{k=1}^{m}\sqrt{\min\left(1,\frac{m-k+1}{\sigma^k}\right)} = \sum_{k=1}^{\overline{k}-1}1+\sum_{k=\overline{k}}^{m}\sqrt{\frac{m-k+1}{\sigma^k}}\\ &\le \overline{k}-1+\sqrt{m-\overline{k}+1}\cdot\sum_{k=\overline{k}}^{m}\frac{1}{\sqrt{\sigma^k}} < \overline{k}-1+\frac{\sqrt{\sigma}}{\sqrt{\sigma}-1}\cdot\sqrt{\frac{m-\overline{k}+1}{\sigma^{\overline{k}}}}\\ &< \overline{k}-1+\frac{\sqrt{\sigma}}{\sqrt{\sigma}-1}\le\overline{k}+1<\log_\sigma m\,,\end{aligned} \tag{5.16}$$

where $\overline{k}$ is defined as above.

Next we estimate $E(Z_i) = Pr\{P_i \sqsupseteq_{md} T\}$, for $0 \le i \le m-1$.

Let us denote by $\mu(i)$ the number of distinct strings which have an md-match with a given string of length i and whose characters are pairwise distinct. Then

$$Pr\{P_i \sqsupseteq_{md} T\} \le \frac{\mu(i+1)}{\sigma^{i+1}}\,.$$

From the recursion

$$\begin{cases}\mu(0) &= 1\\ \mu(k+1) &= \sum_{h=0}^{k}\mu(h)+\sum_{h=1}^{\lfloor\frac{k-1}{2}\rfloor}\mu(k-2h-1) \qquad (\text{for } k\ge 0)\,,\end{cases}$$

it is not hard to see that $\mu(i+1) \le 3^i$, for $i = 0, 1, \dots, m-1$, so that we have

$$E(Z_i) = Pr\{P_i \sqsupseteq_{md} T\} \le \frac{3^i}{\sigma^{i+1}}\,. \tag{5.17}$$

Then, concerning $E(Z)$, from (5.17) we have

$$E(Z) = E\left(\sum_{i=0}^{m-1} Z_i\right) = \sum_{i=0}^{m-1}E(Z_i) \le \sum_{i=0}^{m-1}\frac{3^i}{\sigma^{i+1}} < \frac{1}{\sigma}\cdot\frac{1}{1-\frac{3}{\sigma}} = \frac{1}{\sigma-3} \le 1 \tag{5.18}$$

(we recall that we have assumed $\sigma \ge 4$).

Likewise, from (5.17) we have

$$\sum_{i=0}^{m-1} \sqrt{E(Z_i)} \leq \sum_{i=0}^{m-1} \sqrt{\frac{3^i}{\sigma^{i+1}}} < \frac{1}{\sqrt{\sigma}} \cdot \frac{1}{1 - \sqrt{\frac{3}{\sigma}}} = \frac{1}{\sqrt{\sigma} - \sqrt{3}} < 4 \,. \tag{5.19}$$

From (5.18), (5.15), (5.16), and (5.19), it then follows that (5.10) is bounded from above by

$$(n-1) \cdot (9 \log_\sigma m + 1) \,,$$

yielding a $\mathcal{O}(n \log_\sigma m)$ average-time complexity for the M-SAMPLING algorithm.

Space complexity

In order to evaluate the space complexity of the M-SAMPLING algorithm, we observe that in the worst case, during the j-th iteration of its main **for**-loop, the sets $\mathcal{F}^k_{j-k}$ and $\mathcal{S}_{j-2k}$, for $1 \leq k \leq \alpha$, must be kept in memory to handle translocations, as well as the sets $\mathcal{S}_{j-k}$, for $2 \leq k \leq \beta$, to handle inversions. However, as explained before, we do not keep the values of $\mathcal{F}^k_{j-k}$ explicitly but we rather maintain only their corresponding T-configurations of the automaton $\mathscr{F}(P)$. Thus, we need $\mathcal{O}(\alpha)$-space for the last α configurations of the automaton and $\mathcal{O}(m \max(\alpha, \beta))$-space to keep the last $\max(2\alpha, \beta)$ values of the sets $\mathcal{S}_{j-k}$, considering the maximum cardinality of each set is m. Observe also that, although the size of the DAWG is linear in m, the *end–pos*$(\cdot)$ function can require $\mathcal{O}(m^2)$-space. Therefore, the total space complexity of the M-SAMPLING algorithm is $\mathcal{O}(m^2)$.

5.2.4 A bit-parallel implementation

In this section we present an efficient simulation of the M-SAMPLING algorithm based on bit-vectors.

We associate a bit vector pos to each node of the DAWG. For each node q of the DAWG of P, $\mathsf{pos}(q)$ encodes the *end-pos* function, while, for each node q of the DAWG of P^{r}, $\mathsf{pos}(q)$ encodes the starting positions in P of the reversed factors represented by the node, i.e., $\{(m-1-i) \mid i \in \textit{end-pos}(q)\}$. The bit-vectors F^k_j and I^k_j, corresponding to $\mathcal{F}^k_j$ and $\mathcal{I}^k_j$ respectively, can be computed by the following assignments:

$$\begin{aligned}\mathsf{F}_j^k &\leftarrow \ \mathsf{pos}(\phi(q_j,k))\\ \mathsf{I}_j^k &\leftarrow \ \mathsf{pos}(\phi(q_j^{\mathsf{r}},k)) \ll (k-1)\,.\end{aligned}$$

Each set $\mathcal{S}_j$ is mapped into a corresponding bit-vector S_j. Finally, for each character c of the alphabet Σ, a bit mask $\mathsf{B}[c]$, representing the positions of c in P, is maintained.

The algorithm scans T from left to right and, for each position $j \geq 0$, it computes the vector S_j in terms of S_{j-1}, of S_{j-2k}, F_{j-k}^k, and F_j^k, for $1 \leq k \leq l_j$, and of S_{j-k} and I_j^k for $1 \leq k \leq l_j^{\mathsf{r}}$, with the following bitwise operations:

$$\begin{aligned}\mathsf{S}_j &\leftarrow ((\mathsf{S}_{j-1} \ll 1) \mid 1) \ \& \ B[T[j]]\\ \mathsf{S}_j &\leftarrow \mathsf{S}_j \mid (((((\mathsf{S}_{j-2k} \ll k) \mid \mathsf{10}^{k-1}) \ \& \ \mathsf{F}_j^k) \ll k) \ \& \ \mathsf{F}_{j-k}^k)\\ \mathsf{S}_j &\leftarrow \mathsf{S}_j \mid (((\mathsf{S}_{j-k} \ll k) \mid \mathsf{10}^{k-1}) \ \& \ \mathsf{I}_j^k)\,,\end{aligned}$$

corresponding respectively to the relations:

$$\begin{aligned}\mathcal{S}_j &= \{i+1 : i \in \mathcal{S}_{j-1} \cup \{-1\} \wedge P[i] = T[j]\}\\ \mathcal{S}_j &= \mathcal{S}_j \cup \{i+2k : i \in \mathcal{S}_{j-2k} \cup \{-1\} \wedge (i+k) \in \mathcal{F}_j^k \wedge (i+2k) \in \mathcal{F}_{j-k}^k\}\\ \mathcal{S}_j &= \mathcal{S}_j \cup \{i+k : i \in \mathcal{S}_{j-k} \cup \{-1\} \wedge (i+k) \in \mathcal{I}_j^k\}\,.\end{aligned}$$

During the j-th iteration, if the m-th bit of S_j is set to 1, i.e., if $\mathsf{S}_j \ \& \ \mathsf{10}^{m-1} \neq \mathsf{0}^m$, a match at position j is reported.

The algorithm has a $\mathcal{O}(n \max(\alpha,\beta)\lceil m/w \rceil)$ worst-case time complexity and a $\mathcal{O}((m+\sigma)\lceil m/w \rceil)$-space complexity, where σ is the size of the alphabet. When the length of the pattern satisfies $m \leq w$, the worst-case time and space complexity become $\mathcal{O}(n \max(\alpha,\beta))$ and $\mathcal{O}(\sigma + m)$, respectively.

5.2.5 Computing the minimum cost

For some applications it is not enough to find all the approximate occurrences of the pattern; rather, it can also be important to compute, for each match, the associated minimum cost. In this section we show how to extend the dynamic programming algorithm to compute the minimum cost under the *md* distance, where $\delta(zw, wz)$ and $\delta(z, z^{\mathsf{r}})$ are the cost of a translocation and inversion operation, respectively. The equation and the recurrence to compute the sets $\mathcal{S}_j$ can be easily modified to compute, for each $i \in \mathcal{S}_j$, the distance $md(P_i, S[j-i+1\,..\,j])$.

To ease the formalization, we represent $\mathcal{S}$ as a matrix $n \times m$, where

$$\mathcal{S}_{j,i} = md(P_i, S[j-i+1\,..\,j])$$

for $0 \leq j < n$ and $0 \leq i < m$. In this scenario we must take care not to consider a translocation or an inversion when $z = w$ or $z = z^{\mathsf{r}}$ (i.e., z is a palindrome), respectively. To this end, we shall make use of the following simple result concerning the *end-pos*$(\cdot)$ function:

Lemma 5.7. *Given a string $P \in \Sigma^*$ and strings $u, v \in Fact(P)$ such that $|u| = |v|$, then the following implication holds*

$$(\textit{end-pos}(u) \cap \textit{end-pos}(v)) \neq \emptyset \rightarrow u = v. \quad \square$$

Based on Lemma 5.7, condition (b) of Lemma 5.5 can be modified as follows:

$$(i-k) \in \mathcal{F}_j^k \wedge i \in \mathcal{F}_{j-k}^k \wedge i \notin \mathcal{F}_j^k \wedge (i-2k) \in \mathcal{S}_{j-2k} \cup \{-1\}$$

i.e., we add the condition $i \notin \mathcal{F}_j^k$ to verify that $\mathcal{F}_j^k$ and $\mathcal{F}_{j-k}^k$ do not represent the same factor.

Similarly, condition (c) of Lemma 5.5 can be modified as follows:

$$i \in \mathcal{I}_j^k \wedge i \notin \mathcal{F}_j^k \wedge (i-k) \in \mathcal{S}_{j-k} \cup \{-1\}$$

i.e., we add the same condition $i \notin \mathcal{F}_j^k$ to verify that $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$ do not represent the same factor. Indeed, it is easy to see that if this is the case, then the implied factor is a palindrome. Within this new formulation we have to compute $\mathcal{F}_j^k$, for $i = 0, \ldots, l_j^r$. This can be easily achieved in two steps; we first compute the node q correspoding to the longest factor of P ending at j with length at most l_j^r:

$$q = \begin{cases} \phi(q_j, l_j^r) & \text{if } l_j > l_j^r \\ q_j & \text{otherwise.} \end{cases}$$

Then, in the loop which handles inversions, we compute $\mathcal{F}_j^k$, for $i = 0, \ldots, l_j^r$, in decreasing order of k, with at most one iteration over the suffix link of q. Note that, if $l_j < l_j^r$, the sets $\mathcal{F}_j^k$ are empty, for $k = l_j + 1, \ldots l_j^r$, and so we start iterating after $l_j^r - l_j$ steps . With this modification, the complexity of the loop to compute inversions becomes $\mathcal{O}(\max(\alpha, \beta))$.

We are now ready to describe how to compute the matrix $\mathcal{S}$. The element $\mathcal{S}_{j,i}$ can be recursively defined as follows:

$$\mathcal{S}_{j,i} = \min(g_1(i,j), g_2(i,j), g_3(i,j)),$$

where the functions g_1, g_2, and g_3 map a pair of indices (i,j) onto the cost of converting P_i into the substring of length i ending at j with a standard match, a translocation, and an inversion ending at j respectively. These functions are defined as follows:

$$g_1(i,j) \quad = \begin{cases} \mathcal{S}_{j-1,i-1} & \text{if } i > 0 \text{ and } \mathcal{S}_{j-1,i-1} < \infty \text{ and } P[i] = T[j] \\ 0 & \text{if } i = 0 \text{ and } P[i] = T[j] \\ \infty & \text{ otherwise} \end{cases}$$

$$g_2(i,j) \quad = \begin{cases} \mathcal{S}_{j-2k,i-2k} + \delta(zw,wz) & \text{if } i \geq 2k \text{ and } \mathcal{S}_{j-2k,i-2k} < \infty \text{ and} \\ & \quad (i-k) \in F^k_j \text{ and } i \in F^k_{j-k} \setminus F^k_j \\ \delta(zw,wz) & \text{if } k \leq i < 2k \text{ and } i-k \in F^k_j \\ & \text{and } i \in F^k_{j-k} \setminus F^k_j \\ \infty & \text{otherwise} \end{cases}$$

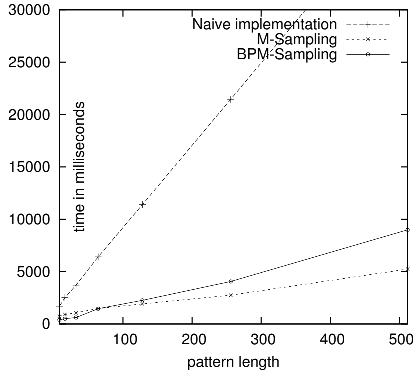
$$g_3(i,j) \quad = \begin{cases} \mathcal{S}_{j-k,i-k} + \delta(z,z^{\mathrm{r}}) & \text{if } i \geq k \text{ and } \mathcal{S}_{j-k,i-k} < \infty \text{ and } i \in I^k_j \setminus F^k_j \\ \delta(z,z^{\mathrm{r}}) & \text{if } i < k \text{ and } i \in I^k_j \setminus F^k_j \\ \infty & \text{otherwise.} \end{cases}$$

Observe that, by using a matrix representation, the average size of the row $\mathcal{S}_j$ is $\Theta(m)$; hence, the average time complexity of the algorithm becomes $\mathcal{O}(nm \log_\sigma m)$.

5.2.6 Experimental evaluation

Next we present some experimental results which allow to compare, in terms of running times, the M-Sampling algorithm, based on the DAWG approach, against its direct dynamic programming implementation. We have also included in our comparison the variant BPM-Sampling of the M-Sampling algorithm.

We remark that sets have been implemented as bit vectors also in the first two algorithms, so that *member* and *insert* operations can be performed in constant



m	M-S	BPM-S
8	740.82	402.26
16	911.21	503.96
32	1068.46	615.47
64	1473.12	1462.29

Figure 5.7: Experimental results relative to a DNA sequence of the *Escherichia coli* genome with $\sigma = 4$. Running times (ms) are also tabulated for small values of m to ease the comparison between the M-Sampling algorithm (M-S) and the BPM-Sampling algorithm (BPM-S).

time.

Iteration over the elements of a set represented as a bit vector can then be implemented efficiently in time proportional to its cardinality by repeatedly

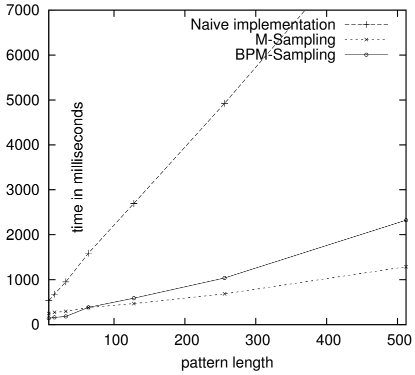
(a) extracting the lowest bit set,

(b) computing its index, and

(c) masking it, until there are no more bits set.

In the BPM-Sampling algorithm, bitwise operations have a $\Theta(\lceil m/w \rceil)$ complexity, since they have to update $\lceil m/w \rceil$ words. Instead, in the M-Sampling algorithm the corresponding operations have a $\Theta(\lceil m/w \rceil + |\mathcal{S}_j|)$ complexity, because, for each word of the bit vector that encodes $\mathcal{S}_j$, it iterates over all the bits set ($|\mathcal{S}_j|$ in total). Since, on average, the sets $\mathcal{S}_j$ contain only a few elements, the average complexity of iterating over all the elements of a set is $\mathcal{O}(\lceil m/w \rceil)$. The tests have been performed on a 1.5 GHz PowerPC G4. We used the input files (iii) and (iv) (see Section 2.6). For each input file, we have generated sets of 50 patterns of fixed length m, randomly extracted from the text, for m ranging in the set $\{8, 16, 32, 64, 128, 256, 512\}$. For each set of patterns, we have calculated the mean over the running times of the 50 runs.

As can be seen from the plots in Figs. 5.7 and 5.8, the M-Sampling algorithm is considerably faster than its naive implementation. Indeed, even if their



m	M-S	BPM-S
8	247.00	141.53
16	273.77	162.10
32	294.60	183.24
64	379.26	385.60

Figure 5.8: Experimental results relative to a protein sequence of the *Saccharomyces cerevisiae* genome with $\sigma = 20$. Running times (ms) are also tabulated for small values of m to ease the comparison between the M-Sampling algorithm (M-S) and the BPM-Sampling algorithm (BPM-S).

asymptotic time complexity is the same, the hidden constant in the naive implementation, due to the explicit computation of the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$, is quite large. In our experiment with a computer word of size 32, it turns out that the BPM-Sampling algorithm is faster than the M-Sampling algorithm only for $m \leq 32$, as can be observed by looking at the running times (ms) reported in the tables. As explained above, the complexity of the bitwise operations is the same, on average, for both algorithms. However, the M-Sampling algorithm scales better because it requires fewer bitwise operations. Finally, observe that the rate of growth of the M-Sampling and the BPM-Sampling algorithm matches the average $\mathcal{O}(n \log_\sigma m)$-time complexity estimated in Section 5.2.3 under the assumptions of equiprobability and independence of characters.

Chapter 6

The compressed string matching problem

The *compressed string matching* problem consists in finding all the occurrences of a pattern in a text stored in compressed form. A straightforward solution is the so-called *decompress-and-search* strategy, which consists in decompressing the text and then using any classical string matching algorithm for searching. However, recent results show that in many cases searching directly in compressed texts can be more efficient. In Section 6.1 we discuss this problem when the compression method employed is an optimal prefix code (we recall that a prefix code is a variable-length code with the property that no codeword is a prefix of any other codeword in the set). This compression method is also known as *Huffman coding* [40]. The limitation of using a variable-length code is that the decoding of the compressed text must be performed from left to right and no bit can be skipped. For this reason, it is natural to try to adapt the Knuth-Morris-Pratt algorithm rather than Boyer-Moore like algorithms to work directly on the compressed text. Daptardar and Shapira proposed [62] a modified Knuth-Morris-Pratt algorithm, while Takeda *et al.* proposed [64] a modification of the Aho-Corasick algorithm. We present a method to solve the *string matching* problem on Huffman encoded texts that makes it possible to skip bits when decoding, by making use of a fairly recent data structure, named *skeleton tree* [44], to represent a prefix code.

In Section 6.2.1 we investigate this problem when the compression method used is the *Burrows-Wheeler transform* [15]. The Burrows-Wheeler transform (BWT) is a reversible transformation which yields a permutation of the text that can be better compressed using the combination of a locally-adaptive encoding, such as move-to-front [10], and statistical methods [40, 67]. The main issue which arises in this case is that in the decoding phase the text must be preprocessed at least once and $\mathcal{O}(n)$-space is needed for the preprocessed data, where n is the text size. In particular, finding the set of positions of the pattern in the text requires three iterations over the text and $\Theta(n)$-space. Instead, counting the occurrences of the pattern in the text requires two iterations over the text and $\Theta(n)$-space. We present a new method to solve the problem of counting the occurrences of a pattern in a BWT encoded text that requires one iteration only over the text and $\mathcal{O}(n)$-space. Despite the space used is still bounded by n, for large alphabets the actual space compares favorably with the total size of the text.

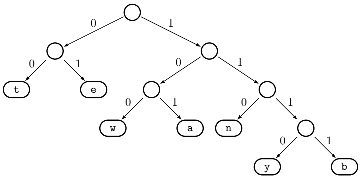
6.1 String matching on Huffman encoded texts

We investigate next the string matching problem on Huffman compressed texts. The Huffman data compression method [40] is an optimal statistical coding. More precisely, the Huffman algorithm computes an optimal *prefix code* relative to given frequencies of the alphabet characters. A prefix code is a set of (binary) words containing no word which is a prefix of another word in the set. Thanks to such a property, decoding is particularly simple. Indeed, a binary prefix code can be represented by an ordered binary tree, whose leaves are labeled with the alphabet characters and whose edges are labeled by 0 (left edges) and 1 (right edges) in such a way that the codeword of an alphabet character is the word labeling the branch from the root to the leaf labeled by the same character.

Prefix code trees, as computed by the Huffman algorithm, are called *Huffman trees*. These are not, by any means, unique. The usually preferred tree for a given set of frequencies, out of the various possible Huffman trees, is the one induced by *canonical Huffman codes* [61]. This tree has the property that, when scanning its leaves from left to right, the sequence of their depths is nondecreasing.

When performing a search on the bitstream of a Huffman encoded text by a classical string matching algorithm, one faces the problem of *false matches*, i.e., occurrences of the encoded pattern in the encoded text which do not correspond to occurrences of the pattern in the original text. Indeed, the only valid occurrences of the pattern are those correctly aligned with codeword boundaries, or, otherwise

t :	00
e :	01
w :	100
a :	101
n :	110
y :	1110
b :	1111



```
twenty   0̄01̄000̄11̄100̄01̄110
ten         0̄00̄11̄10
ten               0̄00̄11̄10
```

Figure 6.1: A Huffman code for the set of symbols {t, e, w, a, n, y, b}. The binary string 0̄01̄000̄11̄100̄01̄110 is the encoding of the string twenty, where a "bar" indicates the starting bit of each codeword. Two occurrences of the binary string ten start at the 4-th and 10-th bit of the encoded version of the string twenty. Both of them are false matches.

said, valid matches must start on the first bit of a codeword. Consider, for example, the Huffman code presented in Fig. 6.1. Note that there are two false occurrences of the string ten starting at the 4-th and at the 10-th bit, respectively, of the encoded string twenty. Thus a verification that the occurrences detected by the pattern matching algorithm are correctly aligned on codeword boundaries is in order.

False matches can be avoided by using codes in which no codeword is a prefix or a suffix of any other codeword. However, such codes, which are called *affix* or *fix-free*, are extremely infrequent [35].

Klein and Shapira [46] showed that, for long enough patterns, the probability of finding a false match is very low, independently of the algorithm. They then proposed a probabilistic algorithm which works on the assumption that Huffman codes tend to realign quickly after an error.

More recently, Shapira and Daptardar [62] proposed a modification of the Knuth-Morris-Pratt algorithm [47], here referred to as HUFFMAN-KMP, which makes use of a data structure, called *skeleton tree* [44], suitably designed for efficient decoding of Huffman encoded sequences. The resulting algorithm is characterized by fast search times, if compared with the *decompress-and-search* method.

Algorithms based on the Boyer-Moore algorithm [14] have been considered unsuitable for searching Huffman encoded texts because the right-to-left scan does not allow one to determine the codeword boundaries in the compressed text, unless the text is decoded from left to right. In addition, Boyer-Moore-like algorithms are generally considered unsuitable for binary alphabets.

We present a new way to exploit skeleton trees for adapting Boyer-Moore-like algorithms to the compressed string matching problem in Huffman encoded texts. Specifically, we use skeleton trees to verify codeword alignments rather than for decoding. This allows one to skip up to 70% of bits during the processing of the encoded text. Futhermore, we make use of algorithms based on the Boyer-Moore strategy, suitably adapted for searching on binary strings by regarding texts and patterns as sequences of q-grams rather than as sequences of bits.

6.1.1 Preliminary definitions

A *compression method* for a given text T over an alphabet Σ is characterized by a system $(\mathcal{E}, \mathcal{D})$ of two complementary functions:

- an *encoding function* $\mathcal{E} : \Sigma \rightarrow \{0,1\}^+$, and
- an inverse *decoding function* $\mathcal{D}$,

such that $\mathcal{D}(\mathcal{E}(c)) = c$, for each $c \in \Sigma$. The encoding function $\mathcal{E}$ is then recursively extended over strings of characters by putting

$$\begin{array}{rcl} \mathcal{E}(\varepsilon) & = & \varepsilon \\ \mathcal{E}(T[0\mathinner{..}\ell]) & = & \mathcal{E}(T[0\mathinner{..}\ell-1]).\mathcal{E}(T[\ell]), \quad \text{for } 0 \leq \ell < |T|, \end{array}$$

so that $\mathcal{E}(T) = \mathcal{E}(T[0\mathinner{..}|T|-1])$ is just a binary string, i.e., a string over the alphabet $\{0,1\}$.

For ease of notation, we usually write t in place of $\mathcal{E}(T)$ and, more generally, denote binary strings with lowercase letters.

Binary strings are conveniently stored in blocks of k bits, typically bytes ($k = 8$), half-words ($k = 16$), or words ($k = 32$), which can be processed at the cost of a single operation. If p is any binary string, we denote by B_p the vector of blocks whose concatenation gives p, for a given block size k, so that

$$p[i] = B_p[\lfloor i/k \rfloor][i \bmod k], \quad \text{for } i = 0, \ldots, |p|-1$$

(we assume that the last block, if not complete, is padded with 0's).

The sequence of k bits starting at position i in p, denoted by $B_{p,i}$, can be computed from B_p by the following bitwise operations:

$$B_{p,i} = (B_p[\lfloor i/k \rfloor] \gg (i \bmod k)) \mid (B_p[\lfloor i/k \rfloor + 1] \ll (k - (i \bmod k))),$$

(A) *Patt*	0	1	2	3
0	11001011	00101100	10110000	
1	01100101	10010110	01011000	
2	00110010	11001011	00101100	
3	00011001	01100101	10010110	
4	00001100	10110010	11001011	00000000
5	00000110	01011001	01100101	10000000
6	00000011	00101100	10110010	11000000
7	00000001	10010110	01011001	01100000

(C) *Last*
2
2
2
2
3
3
3
3

(B) *Mask*	0	1	2	3
0	11111111	11111111	11111000	
1	01111111	11111111	11111100	
2	00111111	11111111	11111110	
3	00011111	11111111	11111111	
4	00001111	11111111	11111111	10000000
5	00000111	11111111	11111111	11000000
6	00000011	11111111	11111111	11100000
7	00000001	11111111	11111111	11110000

Figure 6.2: Let P =110010110010110010110 and $k = 8$. **(A)** The matrix *Patt*. **(B)** The matrix *Mask*. **(C)** The array *Last*. In the tables *Patt* and *Mask*, bits belonging to P are underlined. Blocks containing a factor of P of length 8 have a shaded background.

for $i = 0, \ldots, |p| - k$.

Thus, a genuine solution to the *compressed string matching* problem consists in finding *all* the occurrences of a pattern P in a text T, over a common alphabet Σ, by operating directly on the block vectors B_t and B_p, representing the binary strings $t = \mathcal{E}(T)$ and $p = \mathcal{E}(P)$, respectively, (again, relative to a fixed block size k).

The algorithms for the compressed string matching problem in Huffman encoded texts, to be presented in Section 6.1.2, are based on a high-level model to process binary strings, adopted in [45, 43, 32], which we shall review in the following section.

A High-Level Model for Matching on Binary Strings

Let us assume that the block size k is fixed, so that all the references to both the text and the pattern will only be in terms of entire blocks of k bits. We refer to a k-bit block as a *byte*, though values larger than $k = 8$ can also be used.

We first define a matrix of bytes *Patt*, of size $k \times (\lceil m/k \rceil + 1)$, consisting of several copies of the pattern P stored in the form of a vector B_p of bytes, where $p = \mathcal{E}(P)$ and $m = |p|$. More precisely, the i-th row of the matrix *Patt*, for $i = 0, 1, \ldots, k - 1$, contains a copy of p shifted by i positions to the right, whose length in bytes is $m_i = \lceil (m + i)/k \rceil$. The i leftmost bits of the first byte remain undefined and are set to 0. Similarly, the rightmost $((k-((m+i) \bmod k)) \bmod k)$ bits of the last byte are set to 0.

Observe that each factor of p of length k appears exactly once in the table *Patt*. For instance, the factor of length k starting at position j of p is stored in $\mathit{Patt}[k-(j \bmod k), \lceil j/k \rceil]$.

The matrix *Patt* is paired with a matrix of bytes *Mask*, of size $k \times (\lceil m/k \rceil + 1)$, containing binary masks of length k, which allow to distinguish between significant and padding bits in *Patt*. In particular, a bit in the mask $\mathit{Mask}[i,h]$ is set to 1 if and only if the corresponding bit of $\mathit{Patt}[i,h]$ belongs to p.

Finally, we define a vector *Last*, of size k, where $\mathit{Last}[i]$ is the index of the last byte in the row $\mathit{Patt}[i]$, i.e., $\mathit{Last}[i] = m_i - 1$, for $0 \leq i < k$.

The procedure Preprocess used to precompute the above tables requires $\mathcal{O}(k \times \lceil m/k \rceil) = \mathcal{O}(m)$ time and $\mathcal{O}(m)$ extra-space. Fig. 6.2 shows the tables *Patt*, *Mask*, and *Last* relative to the pattern $P =$`110010110010110010110`, for a block size $k = 8$.

When the pattern is aligned with the s-th bit of the text, a match is reported if

$$\mathit{Patt}[i,h] = B_t[j+h] \ \& \ \mathit{Mask}[i,h]\,,$$

for $h = 0, 1, ..., \mathit{Last}[i]$, where

- B_t is the block representation of the text encoding $t = \mathcal{E}(T)$,
- $j = \lfloor s/k \rfloor$ is the starting byte position in t, and
- $i = (s \bmod k)$.

6.1.2 Skeleton tree based verification

The *skeleton tree* [44] is a pruned canonical Huffman tree, whose leaves correspond to minimal depth nodes in the Huffman tree which are roots of complete subtrees. It is useful to maintain at each leaf of a skeleton tree the common length of the codeword(s) sharing the prefix which labels the path from the root to it. A fast algorithm for building skeleton trees is described in [44]. Fig. 6.3 shows a canonical Huffman tree and its corresponding skeleton tree, for the set of symbols $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{g}, \mathtt{i}, \mathtt{k}, \mathtt{l}, \mathtt{r}, \mathtt{t}, \mathtt{u}\}$, relative to suitable character frequencies.

Skeleton trees allow a faster Huffman decoding because, once the codeword length has been retrieved at its leaves, it is possible to read a burst of bits to

b :	00
i :	01
d :	1000
t :	1001
a :	1010
r :	1011
l :	1100
c :	1101
g :	11100
k :	11101
u :	11110
e :	11111

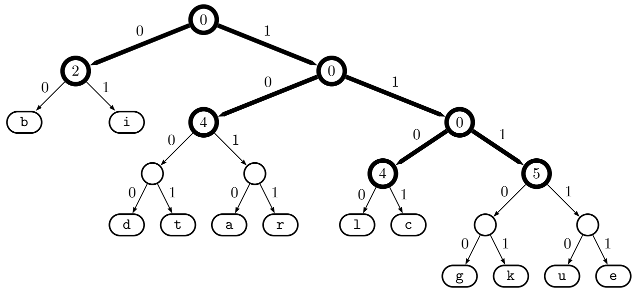


Figure 6.3: The Huffman tree induced by a Huffman code for the set of symbols $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{g}, \texttt{i}, \texttt{k}, \texttt{l}, \texttt{r}, \texttt{t}, \texttt{u}\}$. The skeleton tree is in bold.

complete the codeword, or just to skip them, if one is only interested in finding codeword boundaries.

Our approach consists in searching for the candidate occurrences of B_p in B_t, where we recall that B_p and B_t are the block vectors respectively associated to a given Huffman encoded pattern and text, using Boyer-Moore-like algorithms and then taking advantage of the skeleton tree to verify whether the candidate matches are codeword aligned. In this way we obtain a substantial speedup, especially when the frequency of the pattern in the text is low or when the length of the pattern increases.

For every candidate valid shift s found by the binary pattern matching algorithm, one must verify whether s is codeword aligned. For this purpose, we maintain an offset ρ pointing at the starting position of the last window where a skeleton tree verification took place. The offset ρ is then updated, with the aid of the skeleton tree, to a minimal position $\rho^* \geq s$ which is codeword aligned. Only if $\rho^* = s$ the current window is codeword aligned and s is a valid shift. Plainly, the performance of the algorithm depends on the number of skeleton tree verifications and on the relative distance between candidate valid shifts.

Fig. 6.4 shows the pseudocode for the procedure Sk-Align used to update ρ. In the pseudocode we assume that the starting value of ρ is codeword aligned and that a node x in the skeleton tree is a leaf if the corresponding key is nonzero, i.e., if $\texttt{Key}(x) > 0$. If $\texttt{Key}(x) = \ell > 0$ and c_x is the bit code which labels the path from the root to x, then all codewords c such that $c_x \sqsubseteq c$ have a length equal to ℓ. Thus, if we are only interested in the codeword boundaries, we can skip the

```
Sk-Align ($root$, $t$, $\rho$, $b$)
$x \leftarrow root$, $\ell \leftarrow 0$
while TRUE do
    $B = B_t[\lfloor \rho \;/\; k \rfloor] \ll (\rho \bmod k)$
    if $B < 2^{k-1}$ then $x \leftarrow$ Left($x$) else $x \leftarrow$ Right($x$)
    if Key($x$) $\neq 0$ then
        $\rho \leftarrow \rho +$ Key($x$) $- \ell$, $\ell \leftarrow 0$, $x \leftarrow root$
        if $\rho \geq b$ then break
    else $\rho \leftarrow \rho + 1$,   $\ell \leftarrow \ell + 1$
return $\rho$
```

Figure 6.4: Procedure Sk-Align($root$, t, ρ, b) which computes the next codeword alignment starting from position ρ, where $root$ is the root of the skeleton tree, t is the encoded text in binary form, b is a codeword boundary, and k is the block size.

$\ell - |c_x|$ following bits and restore the skeleton-tree verification from the first bit of the next codeword.

Consider, as an example, the search of the pattern P = "bit" in the text T = "abigblackbugbitabigblackbear". Suppose, moreover, that codewords are defined by the Huffman tree of Fig. 6.3, so that $p = \mathcal{E}(P) =$"00011001". A first candidate valid shift is encountered at position 12 in t, as shown below:

```
     t  1010000111100001100101011011110100111101110000011001101000001[···]
     p              0̄00̄11̄001
verif.  1̄0--0̄-0̄-1̄11--0̄
```

The skeleton tree verification starts at position 0 and stops at position 13, skipping 6 bits over 14 (unprocessed bits are represented by the symbol "-"), showing that the occurrence at position 12 is not codeword aligned.

A second occurrence is found at the 45-th bit of t, as shown below

```
     t  [···]000110010101101111010011110111000001100110100001111000[···]
     p                                      0̄00̄11̄001
verif.        0̄-1̄10-1̄0--1̄10-1̄11--0̄-1̄11--1̄11--0̄
```

The skeleton tree verification restarts from position 14 and finds a codeword alignment at position 45. Thus the occurrence is codeword aligned and the shift is valid. The verification skips 12 bits over 32.

Finally, a third candidate valid shift is found at the 65-th bit of t. This time, the skeleton tree verification skips 10 bits over 22.

$$\begin{array}{rl}
t & [\cdots]\underline{\texttt{00011001}}\texttt{101000011110}\underline{\texttt{00011001}}\texttt{010110111101001111110101011} \\
p & \phantom{[\cdots]\texttt{00011001101000011110}}\overline{\texttt{0}}\texttt{0}\overline{\texttt{0}}\texttt{1}\overline{\texttt{1}}\texttt{001} \\
\textit{verif.} & \overline{\texttt{0}}\texttt{-}\overline{\texttt{0}}\texttt{-}\overline{\texttt{1}}\texttt{0--}\overline{\texttt{1}}\texttt{0--}\overline{\texttt{0}}\texttt{-}\overline{\texttt{0}}\texttt{-}\overline{\texttt{1}}\texttt{11--}\overline{\texttt{0}}
\end{array}$$

The strategy presented above for verifying codeword alignment is general and not specific to any algorithm.

6.1.3 Adapting two Boyer-Moore-like algorithms for searching Huffman encoded texts

Next we deal with the problem of searching for all candidate valid shifts. For this purpose, we present two algorithms which are adaptations to the case of Huffman encoded texts, along the lines of the high-level model outlined in Section 6.1.1, of the FED algorithm [43] and the Binary-Hash-Matching algorithm [32].

The Huffman-Hash-Matching algorithm

Algorithms in the q-Hash family for exact pattern matching have been introduced in [48], by adapting the Wu and Manber multiple string matching algorithm [69] to the single string matching problem. Recently, variants of the q-Hash algorithms have been proposed for searching on binary strings [32].

The first algorithm which we present, called Huffman-Hash-Matching, associates directly each binary substring of length q with its numeric value in the range $[0, 2^q - 1]$, without using any *hash* function. To exploit the block structure of the text, the algorithm considers substrings of length $q = k$.

To begin with, a function $Hs : \{0, 1, \ldots, 2^k - 1\}, \rightarrow \{0, 1, \ldots, m\}$, defined by

$$Hs(B) = \min\Big(\{0 \leq u < m \mid p[m-u-k \mathinner{.\,.} m-u-1] \sqsupseteq B\} \cup \{m\}\Big),$$

for each byte $0 \leq B < 2^k$, is computed during the preprocessing phase. Observe that if $B = p[m-k \mathinner{.\,.} m-1]$, then $Hs[B] = 0$.

For example, in the case of the pattern $P =$ `110010110010110010110` (see Fig. 6.2), we have $Hs[\texttt{01100101}] = 2$, $Hs[\texttt{11001011}] = 1$, and $Hs[\texttt{10010110}] = 0$. At variance with algorithms in the q-Hash family, where the maximum shift is $m - q$, in this case maximum shifts can reach the value m. Since we do not use a hash function but rather map directly the binary substrings of the pattern, the shift table can be modified by taking into account the prefixes of the patterns

```
HUFFMAN-HASH-MATCHING ($p$, $m$, $t$, $n$)
 1. $root \leftarrow$ BUILD-SK-TREE($\phi$)
 2. ($Patt$, $Last$, $Mask$) $\leftarrow$ PREPROCESS ($p$, $m$)
 3. $Hs \leftarrow$ COMPUTE-HASH($Patt$, $Last$, $Mask$, $m$)
 4. $\rho \leftarrow 0$
 5. $i \leftarrow (k - (m \bmod k)) \bmod k$
 6. $B \leftarrow Patt[i][Last[i]]$
 7. $shift \leftarrow Hs[B]$, $Hs[B] \leftarrow 0$
 8. $gap \leftarrow i + 1$,     $j \leftarrow m - 1$
 9. while $j < n$ do
10.    $s \leftarrow j/k$,     $s\ell \leftarrow j \bmod k$
11.    $B \leftarrow B_{t,j-k+1}$
12.    if $Hs[B] = 0$ then
13.       $i \leftarrow (s\ell + gap) \bmod k$
14.       $h \leftarrow Last[i]$,     $q \leftarrow s$
15.       while $h \geq 0$ and
16.             $Patt[i,h] = (B_t[q]$ & $Mask[i,h])$ do
17.          $h \leftarrow h - 1$,     $q \leftarrow q - 1$
18.       if $h < 0$ then
19.          $b \leftarrow (q + 1) \times k + i$
20.          $\rho \leftarrow$ SK-ALIGN($root, t, \rho, b$)
21.          if $\rho = b$ then PRINT($b$)
22.       $j \leftarrow j + shift$
23.    else $j \leftarrow j + Hs[B]$
```

Figure 6.5: The HUFFMAN-HASH-MATCHING algorithm for the compressed string matching problem on Huffman encoded texts. Parameters p and t stand for the Huffman compressed version of the pattern and text, respectively.

$Patt[i]$ of length $k-i$, with $1 \leq i \leq k-1$. Thus Hs can be conveniently computed by setting $Hs[B] = m - k + i$, where i is the minimum index such that $Patt[i][0]$ is a suffix of B, if it exists; otherwise $Hs[B]$ is set to m.

The code of the HUFFMAN-HASH-MATCHING algorithm is presented in Fig. 6.5. The preprocessing phase of the algorithm consists in computing the function Hs defined above and requires $\mathcal{O}(m + k2^{k+1})$-time complexity and $\mathcal{O}(m + 2^k)$ extra space.

During the search phase, the algorithm reads, for each shift position s of the pattern in the text, the block $B = t[s + m - k \,..\, s + m - 1]$ of k bits (line 11). If $Hs(B) > 0$, then a shift of length $Hs(B)$ takes place (line 23). Otherwise, if $Hs(B) = 0$, the pattern p is naively checked in the text block by block (lines 13-17). The verification step is performed using the procedure SK-ALIGN described before (lines 18-21).

After the test, an advancement of length *shift* takes place (line 22), where

$$\mathit{shift} = \min\Big(\{0 < u < m \mid p[m-u-k \mathinner{..} m-u-1] \sqsupset p[m-k \mathinner{..} m-1]\} \cup \{m\}\Big)\,.$$

The Huffman-Hash-Matching algorithm has an overall $\mathcal{O}(\lfloor m/k \rfloor n)$-time complexity and requires $\mathcal{O}(m + 2^k)$ extra space.

For blocks of length k, the size of the *Hs* table is 2^k, which seems reasonable for $k = 8$ or even 16. For larger values of k it is possible to adapt the algorithm so as to choose the desired time/space tradeoff by introducing a new parameter $K \leq k$, representing the number of bits taken into account for the computation of the shift advancement. Roughly speaking, only the K rightmost bits of the current window of the text are taken into account, reducing the total size of the tables to 2^K, at the price of possibly getting shift advancements of the pattern shorter than the ones that would have been obtained if the full length of blocks had been taken into consideration.

The Huffman-FED algorithm

The FED algorithm [43] (Fast matching with Encoded DNA sequences) is a string matching algorithm specifically designed for matching DNA sequences compressed with a fixed-length encoding, requiring two bits for each character of the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. It combines a multi-pattern version of the Quick-Search algorithm [63] and a simplified version of the Commentz-Walter algorithm [27]. However, its strategy is general enough to be adapted to different encodings, including the Huffman one.

The resulting algorithm, which we call Huffman-FED, makes use of a shift table δ and a hash table λ, both of size 2^k.

More specifically, the shift table δ is defined as follows. For $0 \leq i < k$ and $c \in \Sigma$, we first define the Quick-Search shift table $qs[i][c]$, by putting

$$qs[i][c] = \min\Big(\{m_i{-}2{+}1\} \cup \{m_i{-}2{+}1{-}h \mid \mathit{Patt}[i][h] = c \text{ and } 1 \leq h \leq \mathit{Last}[i]{-}1\}\Big).$$

Then, we put $\delta[c] = \min\{qs[i][c], 0 \leq i < k\}$, for $c \in \Sigma$.

The algorithm also maintains, for each block $B \in \{0 \ldots 2^k - 1\}$, a linked list λ which is used to find candidate patterns. In particular, for each block

$B \in \{0, \dots, 2^k - 1\}$, the entry $\lambda[B]$ is a set of indexes, defined by

$$\lambda[B] = \{0 \le i < k \mid Patt[i][Last[i] - 1] = B\}.$$

In practical cases, each set in the table can be implemented as a linked list.

The code of the Huffman-FED algorithm is presented in Fig. 6.6. The preprocessing phase of the algorithm consists in computing the shift table δ and the hash table λ defined above and, as in the Huffman-Hash-Matching algorithm, it requires $\mathcal{O}(m + k2^{k+1})$-time complexity and $\mathcal{O}(m + 2^k)$ extra space.

During the searching phase, the algorithm performs a fast loop using the shift table δ to locate a candidate alignment of the pattern (lines 18-20). In particular, the algorithm checks whether $\delta[B_t[s]] \neq 1$ and, if this is the case, it advances the shift by $\delta[B_t[s + 1]]$ positions to the right.

If $\delta[B_t[s]] = 1$ then, by definition of δ, we have $B_t[s] = Patt[i, Last[i] - 1]$, for some $0 \le i < k$. In this case the last byte of the current window is used as an index in the hash table and all patterns $Patt[i]$, such that $i \in \lambda[B_t[s]]$, are checked naively against the window (line 7). For each alignment i found, the pattern $Patt[i]$ is compared block by block with the text.

As in the Huffman-Hash-Matching algorithm, one has also to verify that the window is codeword aligned (line 14-17).

The Huffman-FED algorithm has a $\mathcal{O}(\lceil m/k \rceil n)$-time complexity and requires $\mathcal{O}(m + 2^k)$ extra space.

6.1.4 Experimental evaluation

Next, we present experimental results which allow to compare, in terms of running times and percentage of processed bits, the following algorithms:
- the Huffman-KMP algorithm (Hkmp) [62];
- the Huffman-Hash-Matching algorithm (Hhm), presented in Section 6.1.3;
- the Huffman-FED algorithm (Hfed), presented in Section 6.1.3.

In addition, we also tested an algorithm based on the *decompress-and-search* method (D&S for short) that makes use of the 3-*Hash* algorithm [48] for classical exact pattern matching, which is considered among the most efficient algorithms for the problem. The tests have been performed on a 1.5 GHz PowerPC G4. We used the input files (i), (ii) and (v) (see Section 2.6). For each input file, we have generated sets of 100 patterns of fixed length m, for m ranging in the set {4, 8, 16, 32, 64, 128, 256}, randomly extracted from the text. For each set of

```
HUFFMAN-FED ($p$, $m$, $t$, $n$)
$root \leftarrow$ BUILD-SK-TREE($\phi$)
($Patt$, $Last$, $Mask$) $\leftarrow$ PREPROCESS ($p$, $m$)
$(\delta, \lambda) \leftarrow$ COMPUTE-FED($Patt, Last, m$)
$\rho \leftarrow 0$
$s = m/k$
while $s < n$ do
    for $i \in \lambda[B_t[s]]$ do
        $h \leftarrow Last[i]$
        $q \leftarrow s + 1$
        while $h \geq 0$ and
            $Patt[i][h] = B_t[q]$ & $Mask[i][h]$ do
                $h \leftarrow h - 1$
                $q \leftarrow q - 1$
        if $h < 0$ then
            $b \leftarrow (q + 1) \times k + i$
            $\rho \leftarrow$ SK-ALIGN($root, t, \rho, b$)
            if $\rho = b$ then PRINT($b$)
    do
        $s \leftarrow s + \delta[B_t[s + 1]]$
    while $s < n$ and $\delta[B_t[s]] \neq 1$
```

Figure 6.6: The HUFFMAN-FED algorithm for the compressed string matching problem on Huffman encoded texts. Parameters p and t stand for the Huffman compressed version of the pattern and text, respectively.

patterns we have reported the mean over the running times of the 100 runs. The tables also show the minimum ($l_{\min}$) and maximum ($l_{\max}$) length in bits of the compressed patterns. For each set of patterns we have also computed the average number of processed bits.

In the following tables, running times are expressed in milliseconds whereas the number of processed bits is expressed as a percentage of the total number of bits in the text.

The experimental results show that the HUFFMAN-HASH-MATCHING and HUFFMAN-FED algorithms always achieve the best running times. In addition, the HUFFMAN-HASH-MATCHING algorithm always obtains better results than the HUFFMAN-FED algorithm. In particular the running times of both algorithms decrease as the length of the pattern increases, since, as is reasonable to expect, the frequency of the patterns, and thus the number of skeleton tree verifications, is inversely proportional to m.

As expected, the HUFFMAN-KMP algorithm maintains the same performance independently of the pattern frequency. The gain of our algorithms compared

Running times					
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S
4	$[17, 31]$	188.64	134.79	146.34	502.82
8	$[36, 53]$	185.77	105.59	112.98	491.87
16	$[79, 102]$	185.99	76.04	81.25	489.03
32	$[164, 204]$	184.23	65.78	70.31	487.74
64	$[336, 378]$	185.36	64.71	68.91	489.27
128	$[694, 768]$	187.73	72.00	77.11	487.31
256	$[1383, 1545]$	184.09	61.45	65.77	488.46

Processed bits			
m	HKMP	HHM	HFED
4	0.75	0.81	0.95
8	0.76	0.68	0.77
16	0.75	0.45	0.53
32	0.75	0.42	0.48
64	0.75	0.38	0.42
128	0.75	0.34	0.37
256	0.76	0.34	0.36

Table 6.1: Running times (ms) on the Huffman encoded version of the King James version of the Bible.

Running times					
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S
4	$[18, 29]$	96.35	64.13	74.23	296.69
8	$[38, 53]$	95.50	49.38	56.47	289.82
16	$[77, 108]$	95.23	39.26	45.03	287.78
32	$[162, 207]$	94.74	33.55	38.53	287.34
64	$[327, 392]$	94.99	34.21	39.39	287.85
128	$[662, 761]$	94.42	28.54	32.92	287.51
256	$[1347, 1610]$	94.39	29.67	34.18	287.21

Processed bits			
m	HKMP	HHM	HFED
4	0.67	0.74	0.98
8	0.66	0.55	0.68
16	0.66	0.43	0.50
32	0.65	0.35	0.40
64	0.65	0.35	0.38
128	0.64	0.29	0.31
256	0.65	0.30	0.32

Table 6.2: Running times (ms) on the Huffman encoded version of the CIA World Fact Book.

Running times					
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S
4	$[18, 35]$	122.25	87.44	95.44	308.56
8	$[37, 60]$	119.33	73.69	79.74	302.38
16	$[83, 140]$	120.35	45.99	49.67	300.53
32	$[171, 216]$	120.12	45.97	49.70	299.81
64	$[348, 525]$	119.20	41.86	45.26	300.90
128	$[712, 1068]$	117.55	37.80	40.83	299.78
256	$[1439, 1773]$	124.10	38.43	41.45	300.36

Processed bits			
m	HKMP	HHM	HFED
4	0.75	0.81	0.95
8	0.76	0.68	0.77
16	0.75	0.45	0.53
32	0.75	0.42	0.48
64	0.75	0.38	0.42
128	0.75	0.34	0.37
256	0.76	0.34	0.36

Table 6.3: Running times (ms) on the Huffman encoded version of “Don Quixote”.

to Huffman-KMP is at least about 20% and grows as the pattern frequency decreases and the pattern length increases.

Observe that, with the exception of very short patterns, the percentage of bits processed by our newly presented algorithms is always lower than that of the Huffman-KMP algorithm and, in many cases, the gain is almost 50%.

6.2 String matching on BWT-encoded texts

In this section we consider the problem of searching for a given pattern in a text encoded by the Burrows-Wheeler Transform [15]. The Burrows-Wheeler transform (BWT) is a reversible transformation which yields a permutation of the text that can be better compressed using the combination of a locally-adaptive encoding, such as move-to-front [10], and statistical methods [40, 67].

There are two main approaches to searching in an encoded text, *offline* and *online*. In the offline approach the encoded text can be modified and preprocessed for "free" before searching it. Thus solutions in the offline approach generally consist in building at encoding time an index of some sort of the encoded data which can be used to efficiently search arbitrary substrings of the indexed text. An index should support at least two types of queries: counting the occurrences of a pattern and locating their positions. Most offline algorithms [55, 33] are based on the relationship between the Burrows-Wheeler transform and the suffix array of the text [50]. They consist in creating an index, based on the compression of the suffix array, which contains the indexed text.

In the online approach the text comes directly in its encoded form and any preprocessing of the text and pattern can take place only at search time. For this reason solutions in the online approach have generally a slower searching time than those based on the offline approach. Traditionally, in the online approach, only the pattern should be preprocessed. However, existing online algorithms for the Burrows-Wheeler transform perform a preprocessing also on the encoded text; in this respect, they are not strictly online. The difference is that in this case the index is built at search time and resides in main memory. The major drawback is that they require more than one iteration over the encoded text and that the size of the preprocessed data is linear in the size of the text.

We propose a new type of preprocessing for *online* algorithms to answer count queries. While still requiring linear space in the worst case, it uses less space on average when the alphabet is moderately large, and requires just one iteration

i	M $(id = 5)$	F	L	V	W	I	Hr
1	`imississipp`	`i`	`p`	6	5	11	5
2	`ippimississ`	`i`	`s`	8	7	8	4
3	`issippimiss`	`i`	`s`	9	10	5	11
4	`ississippim`	`i`	`m`	5	11	2	9
5	`mississippi`	`m`	`i`	1	4	1	3
6	`pimississip`	`p`	`p`	7	1	10	10
7	`ppimississi`	`p`	`i`	2	6	9	8
8	`sippimissis`	`s`	`s`	10	2	7	2
9	`sissippimis`	`s`	`s`	11	3	4	7
10	`ssippimissi`	`s`	`i`	3	8	6	6
11	`ssissippimi`	`s`	`i`	4	9	3	1

	C
`i`	1
`m`	5
`p`	6
`s`	8

Figure 6.7: The matrix M and related arrays for the string "`mississippi`".

over the encoded text. We also illustrate variants based on this new preprocessing of existing online algorithms and present experimental results under various conditions. It turns out that such variants show better time and space behaviour, as compared with solutions currently available in literature.

6.2.1 The Burrows-Wheeler transform

Given a string T of length n, a rotation of T is a string $T[i \mathinner{..} n]T[1 \mathinner{..} i-1]$, for any $i = 1, \ldots, n$. To define the Burrows-Wheeler transform of a given string $T[1 \mathinner{..} n]$, we introduce a conceptual matrix M whose rows are the rotations of T sorted in lexicographic order. We indicate the i-th row of M with M_i, for $1 \leq i \leq n$. Fig. 6.7 shows the matrix M corresponding to the string "`mississippi`". Note that each column of M is a permutation of the characters of T. Let F and L be the first and the last columns of M, respectively. Hence F, by definition of M, can be obtained by sorting lexicographically the characters of T, and can thus be computed from any column of M, in particular from L. Then the BWT-encoding of T is defined as the pair (L, id), where id is the index in L corresponding to character $T[n]$. It turns out that in general the string L is highly compressible, as it contains with high probability long runs of identical characters.

The BWT-encoding of a string of length n can be performed in $\mathcal{O}(n^2 \log n)$-time by the naive method outlined above. If we compute the BWT-encoding of the string $T^{'} = T\#$, obtained by appending to T a special character $\#$ that is lexicographically smaller than any other character in T, then id is not needed since the index in L of character $T[n] = T'[n-1]$ is fixed and equal to 1. Moreover, in this case sorting the rotations is equivalent to sorting the suffixes which can be performed using suffix trees [51] or suffix arrays [50].

The reverse BWT can be computed by a simple method, based on n subsequent sortings. However, a more efficient approach, proposed by Burrows and Wheeler, requires only two iterations over the data and has a linear cost. This consists in building an array V with the property that, for any character $L[i]$, the preceding character in the text is given by $L[V[i]]$. Thus, the array V can be used to decode the text backwards as follows

$$T[n-i] = L[V^i[id]]\,, \quad \text{for } 0 \leq i \leq n-1\,,$$

where $V^0[j] = j$ and, recursively, $V^{i+1}[j] = V[V^i[j]]$.

Plainly, such transformation is practical only for decoding the entire text backwards. For left-to-right scanning, one can use a forward transformation, which is defined by iterating the inverse function $W = V^{-1}$ as follows

$$T[i] = L[W^i[id]]\,, \quad \text{for } 1 \leq i \leq n\,.$$

Both V and W can be computed in linear time, as shown in Fig. 6.8, based on the following formula

$$V[i] = C[L[i]] + r_i\,, \tag{6.1}$$

where r_i is the number of occurrences of character $L[i]$ in $L[1\,..\,i]$ and C is an array, of dimension σ, such that $C[c]-1$ is the number of all occurrences in T of the characters which precede alphabetically c, for $c \in \Sigma$. (Notice that here and in the following we are implicitly identifying the ordered alphabet Σ with the integer range $[1\,..\,\sigma]$, where $\sigma = |\Sigma|$.) By definition of M and C, $C[c]$ turns out to be the index of the first row in M starting with c. Observe that C can be used to implicitly define the column F since $C[c]$ and $C[c+1]-1$ are the indexes in F of the first and last occurrence of the character c, respectively, for each $c \in \Sigma$.

6.2.2 Searching on BWT-encoded texts

Let P be a pattern of length m, and let L be the BWT-encoding of a text T of length n, both over a finite alphabet Σ of dimension σ. In this section we describe the existing solutions for the (online) problem of searching the pattern P in T via L. Note that a direct online solution consists in decoding L and then using any classical string matching algorithm for the searching phase.

The first nontrivial online solution [9] is based on a function *Hr* which maps all characters from T to F and allows to access text positions in random order. Formally, the mapping Hr and its inverse I are defined so as to satisfy

$$T[i] = F[Hr[i]] \quad \text{and} \quad T[I[i]] = F[i], \quad \text{for} \quad 1 \leq i \leq n.$$

The construction of Hr and I (cf. Fig. 6.8) requires three iterations over the BTW-encoded text and the computation of the arrays W and C, yielding an overall extra space of size $(3n + \sigma)$.

The resulting method turns out to be simple and flexible since one can use any existing string matching algorithm *as it is*, including non-standard pattern matching algorithms. In particular, [9] has chosen to adapt the Boyer-Moore [14] algorithm, which has a $\mathcal{O}(nm)$ worst-case time complexity, but a sublinear behavior on average. However, the use of a linear algorithm may lead to an overall $\mathcal{O}(n)$ worst-case time algorithm.

A more remarkable result is the Binary-Search algorithm [9], which is based on the following observation. Since all rows M_i such that $P \sqsubseteq M_i$ are contiguous, it is possible to count the occurrences of the pattern P by locating the first and last matching rows. The idea is then to use binary search to locate the pattern in the range of rows $[C[P[1]] \mathinner{..} C[P[1]+1]-1]$. Rows are decoded as needed, using the array W, and are lexicographically compared with the pattern to update the current interval as in standard binary search. Once a matching row M_i is found, the first and last rows are searched using a slightly modified binary search in the ranges $[C[P[1]] \mathinner{..} i-1]$ and $[i \mathinner{..} C[P[1]+1]-1]$, respectively.

Once the range has been found, it is possible to query the corresponding positions in constant time, using the mapping I, since this, by the property above, represents a mapping between F and T. The computation of the mapping I requires, as for Hr, three iterations over the encoded text. However, if one is only interested in counting the matching occurrences, the preprocessing just requires two iterations, to build W.

The search time of the Binary-Search algorithm is $\mathcal{O}(m \log n)$ in the worst case and decreases to $\mathcal{O}(m \log n/\sigma)$ on average. Thus, the overall worst-case time complexity of the algorithm is $\mathcal{O}(n + m \log n)$.

The indexing data structures for the BWT are based on the compression of the suffix array of the text, which is strictly related to the BWT. In fact, they are more than a traditional index in that they also encode the indexed text. Such indexes allow one to efficiently compute two useful generic operations on symbol

```
Build-C ($L$)
for $i \leftarrow 1$ to $\sigma$
    $K[i] \leftarrow 0$
for $i \leftarrow 1$ to $n$
    $K[L[i]] \leftarrow K[L[i]] + 1$
$sum \leftarrow 1$
for $i \leftarrow 1$ to $\sigma$
    $C[i] \leftarrow sum$
    $sum \leftarrow sum + K[i]$
return $C$
```

```
Build-V-W ($L$, $C$)
for $i \leftarrow 1$ to $n$
    $V[i] \leftarrow C[L[i]]$
    $W[C[L[i]]] \leftarrow i$
    $C[L[i]] \leftarrow C[L[i]] + 1$
return $(V, W)$
```

```
BWT-Decode ($L$, $W$, $id$)
$i \leftarrow id$
for $j \leftarrow 1$ to $n$
    $i \leftarrow W[i]$
    $T[j] \leftarrow L[i]$
return $T$
```

```
Build-I-Hr ($W$, $id$)
$i \leftarrow id$
for $j \leftarrow 1$ to $n$
    $Hr[j] \leftarrow i$
    $I[i] \leftarrow j$
    $i \leftarrow W[i]$
return $(I, Hr)$
```

Figure 6.8: Algorithms to compute the C, V, W, I, Hr and T arrays.

sequences:

- *rank*(L, c, i), which returns the number of occurrences of the character c in the prefix $L[1\,..\,i]$;
- *select*(L, c, i), which returns the index in L of the i-th occurrence of the character c.

Note that the *rank* function allows one to compute the array V using formula 6.1, as $r_i = rank(L, L[i], i)$, for $1 \leq i \leq n$.

Recently, much attention has been devoted to the efficient implementation, time and space wise, of this data structure. One well known implementation is the FM-Index [34]. In [34] Ferragina and Manzini presented an efficient algorithm based on *rank* queries, which finds, as in binary search, the range of rows having P as a prefix. In particular they showed that $\mathcal{O}(m)$ *rank* queries on the BWT are needed to count the occurrences of a pattern of length m.

6.2.3 A new efficient approach for online searching BWT-encoded texts

We present now a new approach for online searching BWT-encoded texts, which yields algorithms with sublinear extra space in most practical cases, especially in the case of large alphabets and short patterns. The main idea consists in building a data structure which allows to efficiently implement *select* queries only for characters occurring within the pattern.

To this end, let P be, as above, a pattern of length m over a finite ordered alphabet Σ of size σ and let L be the BWT-encoding of a text T of length n, over the same alphabet. Let $\Sigma_P \subseteq \Sigma$ be the collection of the characters occurring within P. Trivially, $|\Sigma_P| \leq m$.

We construct a *Partial-select* data structure, implemented as an array Ps of size σ, where the entry $Ps[c]$ points to a list containing all positions in L of the character c, in increasing order, for each $c \in \Sigma_P$. Thus, it turns out that $\mathit{select}(L, c, i) = Ps[c, i]$, where $Ps[c, i]$ is the i-th entry in the list pointed to by $P[c]$. If occurrences lists are implemented as arrays, each *select* query can be answered in constant time. For a given character c, let $K[c]$ be the number of the occurrences of c in L. The extra space needed for computing Ps is given by

$$\sigma + \sum_{c \in \Sigma_P} K[c] \leq \sigma + n \,,$$

where the equality holds if $\Sigma_P = \Sigma$. Of course, if the alphabet is small, the gain, if any, is negligible, but for moderately large alphabets, or when m is smaller than σ, it favorably compares with the total size of the text.

The *Partial-select* data structure, in combination with the array C introduced in Section 6.2.1, allows one to compare in an efficient way a pattern P of length m with any row M_i. In particular, one can check whether P is lexicographically smaller, equal, or greater than the prefix of length m of row M_i, for $1 \leq i \leq n$.

To this end, let us suppose that we have succesfully compared a prefix $P[1\mathinner{..}k]$ of the pattern with row M_i, with $k < m$, and also assume that j is the index of $P[k]$ in L, i.e., $L[j] = P[k] = M_i[k]$. In order to compare $P[k+1]$ with $M_i[k+1]$, observe that the index of $M_i[k+1]$ in F is actually j; we thus need to know whether $F[j] = P[k+1]$. This can be done by exploiting the properties of the array C. In particular, given an index j and a character c, it can be verified in constant time if $F[j] = c$, by checking whether j is contained in the interval $[C[c]\mathinner{..}C[c+1]-1]$. If the answer is negative, we can also verify if $F[j]$ is smaller or greater than c by checking whether j is smaller than $C[c]$ or greater than $C[c+1]-1$, respectively.

We use this tecnique to check whether $M_i[k+1] = P[k+1]$. If the answer is positive, we reiterate the same procedure on $L[j']$, where j' is the index of $M_i[k+1]$ in L.

To compute j', we observe that, by definition of M, the i-th occurrence of a character c in F and in L maps onto the same character in T. Moreover, we note that the index j in F corresponds to the $(j - C[F[j]])$-th occurrence of the character $F[j]$ in F. It thus turns out that the index j' can be computed in

```
Build-C-Ps $(P, L)$
$m \leftarrow len(P)$
for $i \leftarrow 1$ to $\sigma$
    $K[i] \leftarrow 0$
    $H[i] \leftarrow 0$
for $i \leftarrow 1$ to $m$
    $H[P[i]] \leftarrow 1$
for $i \leftarrow 1$ to $n$
    $K[L[i]] \leftarrow K[L[i]] + 1$
    if $H[L[i]] > 0$ then
        $Ps[L[i], H[L[i]]] \leftarrow i$
        $H[L[i]] \leftarrow H[L[i]] + 1$
$sum \leftarrow 1$
for $i \leftarrow 1$ to $\sigma$
    $C[i] \leftarrow sum$
    $sum \leftarrow sum + K[i]$
return $(C, Ps)$
```

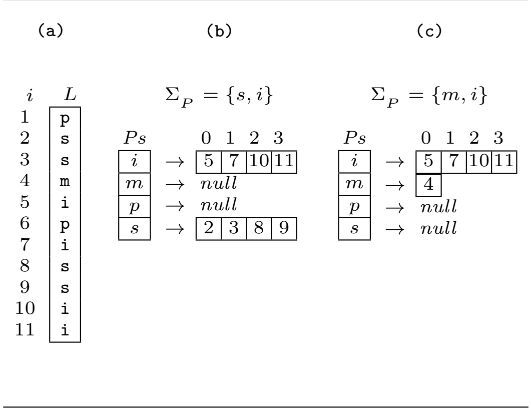


Figure 6.9: left: procedure Build-C-Ps for computing the new data structure Ps. **right**: **(a)** an example: the BWT-encoding of the string "mississippi"; **(b)** the data structure Ps relative to L, for the alphabet $\Sigma_P = \{i, s\}$; **(c)** the data structure Ps relative to L, for the alphabet $\Sigma_P = \{i, m\}$.

constant time by quering the Ps data structure as follows:

$$j' = Ps\Big[P[k+1], j - C\big[P[k+1]\big]\Big].$$

A comparison function, named Ps-Strcmp and based on the Ps data structure, is shown in Fig. 6.10. It requires $\mathcal{O}(m)$-time for comparing P with $M_i[1\,..\,m]$.

Figure 6.9 also shows the code of the procedure for computing the *Partial-select* data structure Ps. It requires a single iteration on the BWT-encoded text and has a $\mathcal{O}(n)$-time and -space complexity.

A Standard-Search algorithm

Our first algorithm works as a standard pattern matching algorithm; it compares the pattern P with the windows $T[i\,..\,i+m-1]$ of the text, for $1 \leq i \leq n-m$. It exploits the fact that to locate all the occurrences of P in T it is enough to compare the pattern with all the windows starting with character $T[i] = P[1]$. This corresponds to comparing the pattern with all the rows M_i starting with $P[1]$. Trivially, M_i starts with symbol $P[1]$ if and only if i is in the range $[C[P[1]]\,..\,C[P[1]+1]-1]$. Thus, our proposed algorithm exploits the property that the i-th occurrence in L of character $P[1]$ is found at position $Ps[P[1], i]$.

The resulting method is simple and flexible and can be used in combination with any existing string matching algorithm which processes the text from left to right, including non-standard pattern matching algorithms.

Figure 6.10 (on the top) shows the code of the Standard-Search algorithm, where the procedure Ps-Strcmp is used as a subroutine for comparing the pattern with a row M_i. Despite its worst-case $\mathcal{O}(nm)$-time complexity, the algorithm turns out to be efficient in practice, expecially when the number of occurrences of the character $P[1]$ is small. In particular, for a given pattern P, the searching phase has a $\mathcal{O}(mK[P[1]])$-time complexity.

A Binary-Search algorithm

Our second proposed solution is a variant of the Binary-Search algorithm, described in Section 6.2.2 (cf. [9]). It makes use of the data structure *Ps* to facilitate the comparison of the pattern P with the rows of the matrix M. The resulting algorithm, whose code is shown in Fig. 6.10 (left), has the same structure of the Binary-Search algorithm. As in the Standard-Search algorithm, we search the pattern in the range of rows $[C[P[1]] \mathinner{..} C[P[1]+1]-1]$. A first binary search is applied (lines 4-11) to locate a matching row M_i such that $P \sqsubseteq M_i$. When a matching row M_i is found (line 12), a slightly modified binary search is used to locate the first row in the range $[C[P[1]] \mathinner{..} i]$ (lines 13-20) and to locate the last row in the range $[i \mathinner{..} C[P[1]+1]-1]$ (lines 21-28).

The new Binary-Search method, as the original algorithm, achieves a $\mathcal{O}(n + m \log n)$ overall time complexity. However, since $|Ps| + |C| \leq |W| + |C|$, in practical cases it uses less space than the Binary-Search algorithm, as shown in Section 6.2.4.

A Rank-Search algorithm

Our last solution for online searching BWT-encoded data is based on the use of the *rank* function, as done in standard indexing algorithms.

The Rank-Search algorithm counts the number of occurrences of P in T by locating two indexes, *sp* and *ep*, such that $P \sqsubseteq M_i$, for all i in the range $[sp \mathinner{..} ep]$. This can be done with $O(m)$ *rank* queries. The code of the Rank-Search algorithm is presented in Fig. 6.10 (right).

The new approach uses the subroutine Ps-Rank to exploit the *Ps* data structure in order to compute efficiently any *rank* query on L. In particular, for $c \in \Sigma_P$,

```
Standard-Search $(P, L)$
$count \leftarrow 0$
$(C, Ps) \leftarrow$ Build-C-Ps$(P, L)$
for $i \leftarrow C[P[1]]$ to $C[P[1]+1]-1$
    if Ps-Strcmp$(P, C, Ps, i) = 0$ then
        $count \leftarrow count + 1$
return $count$
```

```
Ps-Strcmp $(P, C, Ps, i)$
$m \leftarrow len(P)$, $c \leftarrow P[1]$
for $j \leftarrow 2$ to $m$
    $i \leftarrow Ps[c, i - C[c]]$, $c \leftarrow P[j]$
    if $i < C[c]$ then return $1$
    if $i >= C[c+1]$ then return $-1$
return $0$
```

```
Binary-Search $(P, L)$
$count \leftarrow 0$
$(C, Ps) \leftarrow$ Build-C-Ps$(P, L)$
$c \leftarrow P[1]$
$low \leftarrow C[c]$
$high \leftarrow C[c+1] - 1$
while $low < high$
    $mid \leftarrow (low + high)/2$
    $cmp \leftarrow$ Ps-Strcmp$(P, C, Ps, mid)$
    if $cmp = 0$ then break
    if $cmp > 0$ then $low \leftarrow mid + 1$
    else $high \leftarrow mid$
if $cmp = 0$ then
    $h \leftarrow mid - 1$
    while $low < h$
        $m \leftarrow (low + h)/2$
        if Ps-Strcmp$(P, C, Ps, m) > 0$
        then $low \leftarrow m + 1$
        else $h \leftarrow m$
    if Ps-Strcmp$(P, C, Ps, low) \neq 0$
    then $low \leftarrow mid$
    $l \leftarrow mid + 1$
    while $l < high$
        $m \leftarrow (l + high + 1)/2$
        if Ps-Strcmp$(P, C, Ps, l) \geq 0$
        then $l \leftarrow m$
        else $high \leftarrow m - 1$
    if Ps-Strcmp$(P, C, Ps, high) \neq 0$
    then $high \leftarrow mid$
    $count \leftarrow high - low + 1$
return $count$
```

```
Rank-Search $(P, L)$
$count \leftarrow 0$
$(C, Ps) \leftarrow$ Build-C-Ps$(P, L)$
$i \leftarrow len(P)$
$c \leftarrow P[i]$
$sp \leftarrow C[c]$
$ep \leftarrow C[c+1] - 1$
while $sp \leq ep$ and $i \geq 2$
    $c \leftarrow P[i-1]$
    $sp \leftarrow C[c] +$ Ps-Rank$(Ps, c, sp - 1) + 1$
    $ep \leftarrow C[c] +$ Ps-Rank$(Ps, c, ep)$
    $i \leftarrow i - 1$
if $ep \geq sp$ then
    $count \leftarrow ep - sp + 1$
return $count$
```

```
Ps-Rank $(Ps, c, i)$
$low \leftarrow 1$
$high \leftarrow len(Ps[c])$
while $low < high$
    $mid \leftarrow (low + high)/2$
    if $Ps[c][mid] > i$ then
      $high \leftarrow mid$
    else if $Ps[c][mid] < i$ then
            $low \leftarrow mid + 1$
         else return $mid$
return $low$
```

Figure 6.10: Algorithms for online searching BWT-encoded texts. **top:** the Standard-Search algorithm. **left:** the Binary-Search algorithm. **right:** the Rank-Search algorithm.

the query $rank(L, c, i)$ can be answered via the following relation

$$rank(L, c, i) = \max\{j \mid Ps[c, j] \leq i\} \cup \{0\}\,.$$

Since the occurrences lists in Ps are in increasing order, it is possible to use a binary search for locating the value of $rank(L, i, c)$. The procedure Ps-Rank achieves a $\mathcal{O}(\log K[c])$-time complexity for answering any *rank* query on characters occurring in the pattern, where we recall that $K[c]$ is the number of occurrences of c in T. The Rank-Search algorithm achieves a $\mathcal{O}(n + m \log n)$ overall time complexity.

6.2.4 Experimental evaluation

Next, we illustrate and comment on some experimental results, in terms of space usage and running times (preprocessing + searching times), of the following online algorithms for searching BWT-encoded texts:

- HS: the Horspool algorithm which searches the text using the *Hr* mapping;
- D&S: the Decode-and-Search method with the Horspool algorithm;
- BS: the Binary-Search algorithm;
- SS: our Standard-Search algorithm;
- BS2: our modified variant of the Binary-Search algorithm;
- RS: our *rank* based algorithm.

In the case of the HS and D&S algorithms, we used the Horspool algorithm [39] which is a simple and efficient variant of the Boyer-Moore algorithm. The tests have been performed on a 1.5 GHz PowerPC G4. We used the input files (i), (ii), (iii) and (iv) (see Section 2.6). For each input file, we have generated sets of 100 patterns, randomly extracted from the text, of fixed length m, with m ranging in the set $\{4, 8, 12, 16, 20, 24, 28, 32\}$.

Average space usage

For each set of patterns, we computed the space used during preprocessing, expressed as number of bytes for text character. In particular, integers have been represented in our tests by 4 bytes and characters by 1 byte. Hence, the arrays L and W require n bytes and $4n$ bytes, respectively.

Note that the space required by the BS, HS, and D&S algorithms is independent of the alphabet size and of the pattern size.

From the above experimental results, it turns out that the extra space required by our newly presented variants is up to four times smaller than that required by the BS algorithm, whose space-performance is better than those of the algorithms HS and D&S. As expected, the best results are obtained for large alphabets and short patterns. The gap relative to the BS algorithm decreases with the size of

	RS, BS2 and SS										
σ	4	8	12	16	20	24	28	32	HS	D&S	BS
63	1.16	1.76	2.16	2.52	2.68	2.88	2.92	3.04	8.00	5.00	4.00
94	0.72	1.32	1.64	1.84	2.04	2.24	2.40	2.56	8.00	5.00	4.00
20	0.84	1.56	1.96	2.40	2.68	2.96	3.16	3.28	8.00	5.00	4.00
4	2.80	3.44	3.68	3.84	3.92	3.92	3.96	3.96	8.00	5.00	4.00

Table 6.4: Average extra space required by the algorithms in bytes for text character.

m	HS	D&S	BS	BS2	RS	SS
4	354.1	280.9	118.9	**78.8**	79.1	79.6
8	346.6	278.9	119.9	**90.5**	90.8	93.0
12	357.9	294.2	124.9	101.2	**101.1**	102.3
16	371.6	299.0	125.2	108.9	**108.6**	110.2
20	347.2	277.5	120.0	106.3	**106.0**	107.2
24	344.8	276.9	120.5	110.5	**109.9**	111.3
28	339.0	274.1	119.4	**110.1**	110.4	111.6
32	356.8	297.1	125.7	117.8	**117.2**	119.3

Table 6.5: Running times (ms) on the King James version of the Bible.

the alphabet. In particular, for an alphabet of dimension 4 and long patterns the space required by the *Ps* data structure is almost the same as in the BS algorithm.

Overall running times

In the following table we compare the overall running times of the algorithms under consideration. For each set of patterns, we report the mean over the running times of 100 runs. In the following tables, running times are expressed in milliseconds and the best results are bold-faced.

From the above experimental results, it turns out that in most cases the algorithm BS2 achieves the best results, especially in the case of large alphabets, and is second only to the algorithm BS, in the case of 4 characters alphabets. Note, however, that when $\sigma > 4$, our newly presented algorithms do always perform better than previously available solutions.

The HS algorithm, based on the *Hr* mapping, turns out to be the worst, even worse than the D&S method. Accessing the text in a non-sequential way is not cache friendly and since the space needed to decode the text is the same as the one required by *Hr*, there is no reason to prefer the algorithm HS to the D&S

m	HS	D&S	BS	BS2	RS	SS
4	236.2	177.3	74.8	43.3	**43.0**	43.7
8	231.6	175.3	76.1	**49.5**	50.3	50.1
12	228.4	175.0	75.9	**53.8**	54.2	53.9
16	238.8	185.2	78.1	**57.4**	57.7	57.8
20	235.0	185.6	78.3	**59.3**	**59.3**	59.9
24	233.1	186.0	78.7	**61.8**	62.2	62.5
28	229.6	185.4	78.4	63.6	**63.5**	64.1
32	233.9	188.6	78.7	**65.4**	65.3	70.0

Table 6.6: Running times (ms) on the CIA World Fact Book.

m	HS	D&S	BS	BS2	RS	SS
4	253.5	217.3	90.2	**57.1**	58.2	57.4
8	256.7	225.7	91.3	**70.7**	71.6	72.3
12	253.3	219.3	90.6	**76.6**	77.0	76.8
16	255.6	225.3	91.7	**83.0**	84.0	84.0
20	258.1	226.8	91.3	87.7	88.6	**86.7**
24	246.6	218.1	91.7	87.5	**87.3**	88.2
28	250.1	227.8	92.0	**91.2**	92.0	91.6
32	253.4	223.5	92.2	**90.7**	91.4	91.8

Table 6.7: Running times (ms) on a protein sequence.

m	HS	D&S	BS	BS2	RS	SS
4	381.3	360.4	**138.5**	143.3	143.0	145.3
8	390.3	367.5	**139.9**	152.9	153.0	152.2
12	375.7	352.1	**137.4**	153.2	152.0	152.2
16	382.3	354.7	**142.1**	159.0	159.6	158.3
20	380.7	358.5	**138.1**	159.1	157.2	157.2
24	370.6	352.2	**140.3**	158.3	161.5	157.7
28	381.0	372.6	**141.3**	166.6	161.3	160.7
32	376.0	350.6	**137.7**	158.8	157.7	157.7

Table 6.8: Running times (ms) on a DNA sequence.

method. The algorithms BS2 and the RS always achieve better running times as compared with the Binary-Search algorithm. The algorithm SS can be faster than the other algorithms when the frequency of the first character of the pattern is very low, but on average is always slower.

Chapter 7

Conclusions

In this thesis we have presented new results for the *string matching* problem and some of its variants. In particular, we have introduced a novel encoding, based on the bit-parallelism technique, for nondeterministic finite automata with a regular structure like those relative to the languages Σ^*P and $\mathit{Suff}(P)$ of a string P. Our representation exploits a particular factorization of strings to encode automata by smaller bit-vectors. The resulting algorithms scale much better with the length of the pattern. This result leaves some questions open. The first one is whether there exist other factorizations that can be used to obtain efficient automata simulations. Another one is whether such a technique, which is less flexible than bit-parallelism, can be extended to more complex patterns like wildcards or regular expressions.

We have also illustrated an encoding, based on bit-parallelism, of the nondeterministic automata for the languages $\Sigma^*\mathcal{P}$ and $\mathit{Suff}(\mathcal{P})$ of a set of strings $\mathcal{P}$. This representation exploits the relation between reachable configurations of the automata and the associated failure function, and encodes the transition function in polynomial space in the size of the automaton.

We have further presented practical bit-parallel variants of the Wide-Window algorithm for the *string matching* problem that use the bit-level parallelism to simulate two automata in parallel. In one case this technique doubles the shift performed by the algorithm. This result shows that the method of using a fixed length shift allows for an easy parallelization of the algorithm as opposed to BDM-like algorithms; indeed, a variable shift requires a more complex simulation to parallelize the algorithm. For this reason, this method is worth further investigations.

As for the *approximate string matching* problem, we have presented a new distance function based on edit operations involving substrings rather than single characters, i.e., swaps of equal length adjacent substrings and reversals of substrings. We have also introduced an algorithm, based on dynamic programming and on the DAWG data structure, that solves the *approximate string matching* under this distance. A possible future work will be to investigate variants of the proposed distance and more efficient algorithms.

In addition, we have also discussed a simple variant of an algorithm for the *string matching with swaps* problem that is able to count, for each occurrence of the pattern, the corresponding number of swaps without any time and space overhead.

In relation to the *compressed string matching* problem, we have presented adaptations of Boyer-Moore like algorithms relative to the Huffman encoding. An idea that might be worth investigating is whether this approach can be improved by decoding byte-wise rather than bit-wise while still being able to skip bits, as this modification would speed up the decoding phase considerably. Finally, we have presented a more efficient, both time and space wise, preprocessing method to search for a pattern in Burrows-Wheeler encoded texts.

Bibliography

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

[2] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: A new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM '99, Theory and Practice of Informatics, 26th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 27 - December 4, 1999, Proceedings*, volume 1725 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 1999.

[3] A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Journal of Algorithms*, 37(2):247–266, 2000.

[4] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. *Information and Computation*, 181(1):57–74, 2003.

[5] A. Amir, G. M. Landau, M. Lewenstein, and N. Lewenstein. Efficient special cases of pattern matching with swaps. *Information Processing Letters*, 68(3):125–132, 1998.

[6] A. Amir, M. Lewenstein, and E. Porat. Approximate swapped matching. *Information Processing Letters*, 83(1):33–39, 2002.

[7] J. Arndt. *Matters Computational*. Springer, 2011. `http://www.jjj.de/fxt/`.

[8] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[9] T. Bell, M. Powell, A. Mukherjee, and D. Adjeroh. Searching BWT compressed text with the Boyer-Moore algorithm and binary search. In *2002*

Data Compression Conference (DCC 2002), 2-4 April, 2002, Snowbird, UT, USA, pages 112–121. IEEE Computer Society, 2002.

[10] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[11] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[12] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[13] A. Blumer, A. Ehrenfeucht, and D. Haussler. Average sizes of suffix trees and dawgs. *Discrete Applied Mathematics*, 24(1-3):37 – 45, 1989.

[14] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[15] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[16] M. Campanelli, D. Cantone, and S. Faro. A new algorithm for efficient pattern matching with swaps. In J. Fiala, J. Kratochvíl, and M. Miller, editors, *Combinatorial Algorithms, 20th International Workshop, IWOCA 2009, Hradec nad Moravicí, Czech Republic, June 28-July 2, 2009, Revised Selected Papers*, volume 5874 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2009.

[17] M. Campanelli, D. Cantone, S. Faro, and E. Giaquinta. An efficient algorithm for approximate pattern matching with swaps. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2009*, pages 90–104, Czech Technical University in Prague, Czech Republic, 2009.

[18] M. Campanelli, D. Cantone, S. Faro, and E. Giaquinta. Pattern matching with swaps in practice. *International Journal of Foundations of Computer Science*, 2010. in press.

[19] D. Cantone and S. Faro. A space efficient bit-parallel algorithm for the multiple string matching problem. *International Journal of Foundations of Computer Science*, 17(6):1235–1252, 2006.

[20] D. Cantone and S. Faro. Pattern matching with swaps for short patterns in linear time. In M. Nielsen, A. Kucera, P. B. Miltersen, C. Palamidessi, P. Tuma, and F. D. Valencia, editors, *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*, volume 5404 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2009.

[21] D. Cantone, S. Faro, and E. Giaquinta. Adapting boyer-moore-like algorithms for searching huffman encoded texts. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2009*, pages 29–39, Czech Technical University in Prague, Czech Republic, 2009.

[22] D. Cantone, S. Faro, and E. Giaquinta. Adapting boyer-moore-like algorithms for searching huffman encoded texts. *International Journal of Foundations of Computer Science*, 2010. in press.

[23] D. Cantone, S. Faro, and E. Giaquinta. Approximate string matching allowing for inversions and translocations. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2010*, pages 37–51, Czech Technical University in Prague, Czech Republic, 2010.

[24] D. Cantone, S. Faro, and E. Giaquinta. Bit-(parallelism)^2: Getting to the next level of parallelism. In P. Boldi and L. Gargano, editors, *Fun with Algorithms, 5th International Conference, FUN 2010, Ischia, Italy, June 2-4, 2010. Proceedings*, volume 6099 of *Lecture Notes in Computer Science*, pages 166–177. Springer, 2010.

[25] D. Cantone, S. Faro, and E. Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In A. Amir and L. Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, volume 6129 of *Lecture Notes in Computer Science*, pages 288–298. Springer, 2010.

[26] J. D. Cohen. Recursive hashing functions for n-grams. *ACM Transactions on Information Systems*, 15(3):291–320, 1997.

[27] B. Commentz-Walter. A string matching algorithm fast on the average. In H. A. Maurer, editor, *Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings*, volume 71 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 1979.

[28] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

[29] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. *Information Processing Letters*, 71(3-4):107–113, 1999.

[30] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[31] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[32] S. Faro and T. Lecroq. Efficient pattern matching on binary strings. *arXiv*, 0810.2390, 2008. `http://arxiv.org/abs/0810.2390`.

[33] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics*, 13:1.12–1.31, 2009.

[34] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[35] A. S. Fraenkel and S. T. Klein. Bidirectional Huffman coding. *The Computer Journal*, 33(4):296–307, 1990.

[36] K. Fredriksson. Shift-or string matching with super-alphabets. *Information Processing Letters*, 87(4):201–204, 2003.

[37] L. He, B. Fang, and J. Sui. The wide window string matching algorithm. *Theoretical Computer Science*, 332(1-3):391–404, 2005.

[38] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2001.

[39] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.

[40] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. I.R.E.*, 40:1098–1101, 1951.

[41] C. S. Iliopoulos and M. S. Rahman. A new model to solve the swap matching problem and efficient algorithms for short patterns. In V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, and M. Bieliková, editors, *SOFSEM 2008: Theory and Practice of Computer Science, 34th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 19-25, 2008, Proceedings*, volume 4910 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 2008.

[42] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[43] J. W. Kim, E. Kim, and K. Park. Fast matching method for dna sequences. In B. Chen, M. Paterson, and G. Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium, ESCAPE 2007, Hangzhou, China, April 7-9, 2007, Revised Selected Papers*, volume 4614 of *Lecture Notes in Computer Science*, pages 271–281. Springer, 2007.

[44] S. T. Klein. Skeleton trees for the efficient decoding of Huffman encoded texts. *Information Retrieval*, 3(1):7–23, 2000.

[45] S. T. Klein and M. K. Ben-Nissan. Accelerating Boyer Moore searches on binary texts. In J. Holub and J. Žďárek, editors, *Implementation and Application of Automata, 12th International Conference, CIAA 2007, Praque, Czech Republic, July 16-18, 2007, Revised Selected Papers*, volume 4783 of *Lecture Notes in Computer Science*, pages 130–143. Springer, 2007.

[46] S. T. Klein and D. Shapira. Pattern matching in Huffman encoded texts. *Information Processing and Management*, 41(4):829–841, 2005.

[47] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[48] T. Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007.

[49] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

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

[51] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

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

[53] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[54] G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *Theoretical Computer Science*, 321(2-3):283–290, 2004.

[55] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.

[56] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *Journal of Experimental Algorithmics*, 5:4, 2000.

[57] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[58] G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2005.

[59] H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, editors, *String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003, Manaus, Brazil, October 8-10, 2003, Proceedings*, volume 2857 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2003.

[60] L. Salmela, J. Tarhio, and J. Kytöjoki. Multipattern string matching with q-grams. *Journal of Experimental Algorithmics*, 11:1.1, 2006.

[61] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[62] D. Shapira and A. Daptardar. Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts. *Information Processing and Management*, 42(2):429–439, 2006.

[63] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[64] M. Takeda, S. Miyamoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Processing text files as is: Pattern matching over compressed texts, multi-byte character texts, and semi-structured texts. In A. H. F. Laender and A. L. Oliveira, editors, *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002, Lisbon, Portugal, September 11-13, 2002, Proceedings*, volume 2476 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2002.

[65] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[66] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993.

[67] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[68] S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[69] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[70] A. C. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979.