Twenty Years of Bit-Parallelism in String Matching

Simone Faro[†] and Thierry Lecroq[‡]

[†]Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy
[‡]Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr

Abstract. It has been twenty years since the publication of the two seminal papers of Baeza-Yates and Gonnet and of Wu and Manber in the September 1992 issue of the Communications of the ACM. The use of intrinsic parallelism of the bit operations inside a computer word, the so-called bit-parallelism, allows to cut down the number of operations that an algorithm performs by a factor up to ω, where ω is the number of bits in the computer word. This was then achieved by the Shift-Or and the Shift-And string matching algorithms. These two papers has inspired a lot of works and since 1992 a large number of papers were published describing string matching algorithms using this technique. In this survey we will review these solutions for exact single string matching, for exact multiple string matching and for approximate single string matching.

1 Introduction

String matching consists in finding one or more generally all the occurrences (exact or approximate) of a single string (or a finite set of strings) in a text. It is an extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry. String matching is a very important subject in the wider domain of text processing and algorithms for the problem are also basic components used in implementations of practical softwares existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science. Finally they also play an important role in theoretical computer science by providing challenging problems.

Although data are memorized in various ways, text remains the main form to exchange information. This is particularly evident in literature or linguistics where data are composed of huge corpus and dictionaries. This apply as well to computer science where a large amount of data are stored in linear files. And this is also the case, for instance, in molecular biology because biological molecules can often be approximated as sequences of nucleotides or amino acids.

Furthermore, the quantity of available data in these fields tend to double every eighteen months. This is the reason why algorithms should be efficient even if the speed and capacity of storage of computers increase regularly.

Solutions can be based on direct comparisons between symbols of the string and of the text, or on the use of various kinds of automata or by simulating these automata by using bit-parallelism.

Bit-parallelism is a technique firstly introduced in [29], and later revisited, twenty years ago, in [10, 64], which takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that

an algorithm performs by a factor up to ω, where ω is the number of bits in the computer word. Bit-parallelism is indeed particularly suitable for the efficient simulation of non-deterministic automata. In the following we will review the solutions, based on bit-parallelism, for exact single string matching, exact multiple string matching and for approximate single string matching.

The remaining of this paper is organized as follows. Section 2 introduces the basic definitions and the notations used throughout the remaining of the article. In Section 3 we present solutions for exact single string matching. Solutions for exact multiple string matching are presented in Section 4 while solutions for approximate single string matching are presented in Section 5.

2 Notions and Basic Definitions

A string P of length $|P| = m$ over a given finite alphabet 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 $(i+1)$-st character of P, for $0 \leq i < m$. Likewise, the substring of P contained between the $(i+1)$-st and the $(j+1)$-st 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. For any two strings P and P', we write $P.P'$ to denote the concatenation of P' to P, and $P' \sqsupset P$ to express that P' is a *proper* suffix of P, i.e., $P = P''.P'$ for some nonempty string P''. The notation $P' \sqsupseteq P$ will be used with the obvious meaning. Analogously, $P' \sqsubseteq P$ ($P' \sqsubset P$) expresses that P' is a (proper) prefix of P, i.e., $P = P'.P''$ for some (nonempty) string P''. We say that P' is a factor of P if $P = P''.P'.P'''$, for some strings $P'', P''' \in \Sigma^*$, and we denote by $\mathit{Fact}(P)$ the set of the factors of P. Likewise, we denote 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 $\mathcal{P}$, we put $\mathcal{P}^{\mathsf{r}} =_{\text{Def}} \{P^{\mathsf{r}} \mid P \in \mathcal{P}\}$ and $\mathcal{P}_l =_{\text{Def}} \{P[0\,..\,l-1] \mid P \in \mathcal{P}\}$. Also we put $\mathit{size}(\mathcal{P}) =_{\text{Def}} \sum_{P\in\mathcal{P}} |P|$ 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 $\mathit{Suff}(\mathcal{P}) =_{\text{Def}} \bigcup_{P\in\mathcal{P}} \mathit{Suff}(P)$.

The algorithms reviewed in this paper make use of bitwise operations, i.e. operations which operate on one or more bit vectors at the level of their individual bits. On modern architectures bitwise operations have the same speed as addition but are significantly faster than multiplication and division.

We use the C-like notations in order to represent bitwise operations. In particular:

"|" represents the bitwise operation Or; ((01101101) | (10101100) = (11101101));
"&" represents the bitwise operation And; ((01101101) & (10101100) = (00101100));
"$\sim$" represents the one's complement; ($\sim$ (01101101) = (10010010));
"$\ll$" represents the bitwise left shift; and ((01101101) $\ll$ 2 = (10110100));
"$\gg$" represents the bitwise right shift. ((01101101) $\gg$ 2 = (00011011)).

All operations listed above use a single CPU cycle to be computed. Moreover some of the algorithms described below make use of the following, non trivial, bitwise operations:

"reverse" represents the reverse operation; (reverse(01101101) = (10110110));
"bsf" represents the bit scan forward operation; (bsf(00010110) = 3);

"popcount" represents population count operation. (popcount(01101101) = 5);

Specifically, for a given bit-vector B, the reverse operation inverts the order of the bits in a bit-vector B and can be implemented efficiently with $\mathcal{O}(\log_2(\text{length}(B)))$-time, the bsf operation counts the number of zeros preceeding the leftmost bit set to one in B, while the popcount operation counts the number of bits set to one in B and can be performed in $\mathcal{O}(\log_2(\text{length}(B)))$-time. (see [9] for the detailed implementation of the operations listed above).

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. [1]

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 $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathscr{P}(Q)$ is the transition function ($\mathscr{P}(\cdot)$ is the powerset operator).[2] 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 $\text{ECLOSE}(D) = \bigcup_{q \in D} \text{ECLOSE}(q)$. In the case of an NFA without ε-transitions, we have $\text{ECLOSE}(q) = \{q\}$, for any $q \in Q$.

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

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

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

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

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:

$$\delta^*(q, u.v) = \delta^*(\delta^*(q, u), v), \text{ for all } u, v \in \Sigma^*\,. \tag{1}$$

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

1. distinct edges out of the same node are labeled by distinct characters,
2. all paths in $\mathcal{T}_\mathcal{P}$ from the root are labeled by prefixes of the strings in $\mathcal{P}$,
3. for each string P in $\mathcal{P}$ there exists a path in $\mathcal{T}_\mathcal{P}$ from the root labeled by P.

[1] 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 [9].

[2] In the case of NFAs with no ε-transitions, the transition function has the form $\delta : Q \times \Sigma \to \mathscr{P}(Q)$. For the basics on NFAs, the reader is referred to [40].

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.

The directed acyclic word graph (DAWG) 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 $end\text{-}pos(u)$ the set of all positions in $\mathcal{P}$ where an occurrence of u ends, for $u \in \Sigma^*$; more formally, we put

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

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

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

$$u \; R_{\mathcal{P}} \; v = \; end\text{-}pos(u) = end\text{-}pos(v)\,, \tag{2}$$

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

$$val(R_{\mathcal{P}}(u)) =_{\text{Def}} \text{the longest string in the equivalence class } R_{\mathcal{P}}(u)\,. \tag{3}$$

Then the DAWG for a finite set $\mathcal{P}$ of patterns is a directed acyclic graph (V, E) with an edge labeling function $lbl()$, where

$$V = \{R_{\mathcal{P}}(u) \,|\, u \in Fact(\mathcal{P})\}$$

$$E = \{(R_{\mathcal{P}}(u), R_{\mathcal{P}}(uc)) \mid u \in \Sigma^*, c \in \Sigma, uc \in Fact(\mathcal{P})\},$$

and $lbl(R_{\mathcal{P}}(u), R_{\mathcal{P}}(uc)) = c$, for $u \in \Sigma^*$, $c \in \Sigma$ such that $uc \in Fact(\mathcal{P})$ (cf. [13]).

3 Exact String Matching

Given a text t of length n and a pattern P of length m over some alphabet Σ of size σ, the *exact single string matching problem* consists in finding *all* occurrences of the pattern P in the text t.

Applications require two kinds of solutions depending on which string, the pattern or the text, is given first. Algorithms based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern and solve the first kind of problem. This kind of problem is generally referred as *online* string matching. The notion of indexes realized by trees or automata is used instead in the second kind of problem, generally referred as *offline* string matching. In this paper we are only interested in algorithms of the first kind.

Online string matching algorithms (hereafter simply string matching algorithms) can be divided into three classes: algorithms which solve the problem by making

use only of comparisons between characters, algorithms which make use of deterministic automata and algorihtms which simulate nondeterministic automata using bit-parallelism.

Most string matching algorithms generally work as follows. They scan the text with the help on a window of the text whose size is generally equal to m. For each window of the text they check the occurrence of the pattern (this specific work is called an *attempt*) by comparing the characters of the window with the characters of the pattern, or by performing transitions on some kind of automaton, or by using some kind of filtering method. After a whole match of the pattern or after a mismatch they shift the window to the right by a certain number of positions. This mechanism is usually called the *sliding window mechanism*. At the beginning of the search they align the left ends of the window and the text, then they repeat the sliding window mechanism until the right end of the window goes beyond the right end of the text. We associate each attempt with the position s in the text where the window is positioned, i.e., $T[s\,..\,s+m-1]$.

The worst case lower bound of the string matching problem is $\mathcal{O}(n)$. The first algorithm to reach the bound was given by Morris and Pratt [51] later improved by Knuth, Morris and Pratt [48]. The reader can refer to Section 7 of [48] for historical remarks. Linear algorithms have been developed also based on bit-parallelism [10]. An average lower bound in $\mathcal{O}(n \log m/m)$ (with equiprobability and independence of letters) has been proved by Yao in [66].

Many string matching algorithms have been also developed to obtain sublinear performance in practice (see [24]). Among them the Boyer-Moore algorithm [14] deserves a special mention, since it has been particularly successful and has inspired much work. Among the most efficient comparison based algorithms we mention the well known Horspool [41] and Quick-Search [63] algorithms which, despite their quadratic worst case time complexity, show a sublinear behavior.

Automata based solutions have been also developed to design algorithms which have optimal sublinear performance on average. This is done by using factor automata [12, 2], data structures which identify all factors of a word. Among them the Backward Oracle Matching algorithm [2] is one of the most efficient algorithms especially for long patterns.

For short patterns, algorithms based on bit-parallelism are among the most efficient in practice. We will review them, especially the most recent ones, in the following. The reader is referred to [34] for a recent survey and to [33] for an experimental comparison study.

3.1 Standard Algorithms

The Shift-And algorithm simulates the behavior of the non-deterministic string matching automaton (*NSMA*, for short) that recognizes the language $\Sigma^* P$ for a given string P of length m.

The bit-parallel representation of the automaton $NSMA(P)$ uses an array B of $|\Sigma|$ 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 $c \in \Sigma$, $0 \leq i < m$. Automaton configurations $\delta^*(q_0, S)$ on input $S \in \Sigma^*$ are then 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 iff state q_{i+1} is active, i.e. $q_{i+1} \in \delta^*(q_0, S)$, for $i = 0, \ldots, m-1$. For a configuration D of the

NFA, a transition on character c can then be implemented by the following bitwise operations

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

The bitwise Or with 1 (represented as $0^{m-1}1$) is performed to take into account the self-loop labeled with all the characters in Σ on the initial state. When a search starts, the initial configuration D is initialized to 0^m. Then, while the text is read from left to right, the automaton configuration is updated for each text character, as described before. If the final state is active after reading character at position j in the text we report a match at position $j - m + 1$.

The Shift-Or algorithm [10] uses the complementary technique of the Shift-And algorithm. In particular an active state of the automaton is represented with a zero bit while ones represent non active states. The algorithm updates the state vector D in a similar way as in the Shift-And algorithm, but is based on the following basic shift-or operation:

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

Then an occurrence of the pattern is reported if the bit which identifies the final state is set to 0. Both Shift-And and Shift-Or algorithms achieve $O(n\lceil m/\omega \rceil)$ worst-case time and require $O(\sigma\lceil m/\omega \rceil)$ extra-space.

The Backward-Non-deterministic-DAWG-Matching algorithm (BNDM) simulates the non-deterministic factor automaton for $\bar{P}$ with the bit-parallelism technique, using an encoding similar to the one described before for the Shift-And algorithm.

The BNDM algorithm encodes configurations of the automaton in a bit-vector D of m bits (the initial state and state q_0 are not represented). The i-th bit of D is set 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

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

The BNDM 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. Each time a suffix of $\bar{P}$ (i.e., a prefix of P) is found, namely when prior to the left shift the m-th bit of $D\&B[c]$ is set, the window position is recorded. An attempt ends when either D becomes zero (i.e., when no further prefixes of P can be found) or the algorithm has performed m iterations (i.e., when a match has been found). The window is then shifted to the start position of the longest recognized proper prefix. The time and space complexities of the BNDM algorithm are $\mathcal{O}(\lceil m^2/\omega \rceil)$ and $\mathcal{O}(\sigma\lceil m/\omega \rceil)$, respectively.

The bit-parallel encoding used in the Shift-And and BNDM algorithms requires one bit per pattern symbol, for a total of $\lceil m/\omega \rceil$ computer words. Bit-parallel algorithms are extremely fast as long as a pattern fits in a computer word. Specifically, when $m \leq \omega$, the Shift-And and BNDM algorithms achieve $\mathcal{O}(n)$ and $\mathcal{O}(nm)$ time complexity, respectively, and require $\mathcal{O}(\sigma)$ extra space.

When the pattern size m is larger than ω, the configuration bit-vector and all auxiliary bit-vectors need to be splitted over $m\omega$ multiple words. For this reason the

performance of the Shift-And and BNDM algorithms, and of bit-parallel algorithms more in general, degrades considerably as $m\omega$ grows.

Much work has been done in recent years in order to improve the performance of bit parallel algorithms. In the following sections we review in details such solutions.

3.2 Fast Variants of the BNDM Algorithm

A simplified version of the BNDM algorithm (SBNDM for short) has been presented in [61]. Independently, Navarro has adopted a similar approach earlier in the code of his NR-grep [55]. The SBNDM algorithm differs from the original algorithm in the main loop where it skips the examining of longest prefixes. If s is the current alignment position in the text and j is the number of updates done in the window, then the algorithm simply sets $s + m - j + 1$ to be the start position of the next alignment. Moreover, if a complete match is found, the algorithm advances the window of δ positions to the right, where the value δ is the length of the longest prefix of the pattern which is also a suffix of P. The value of δ is computed in a preprocessing phase in $\mathcal{O}(m)$-time.

Despite the fact that the average length of the shift is reduced, the innermost loop of the algorithm becomes simpler leading to better performance in practice.

Another fast variant of the BNDM algorithm was presented during a talk [39]. When the pattern is aligned with the text window $T[s \,..\, s + m - 1]$ the state vector D is not initialized to 1^m, but is initialized according with the rightmost 2 characters of the current window. More formally the algorithm initializes the bit mask D as

$$D \leftarrow (B[T[s + m - 1]] \ll 1) \;\&\; B[T[s + m - 2]].$$

Then the main loop starts directly with a test on D. If $D = 0$ the algorithm performs directly a shift of $m - 1$ positions to the right. Otherwise a standard BNDM loop is executed starting at position $s + m - 3$ of the text.

The resulting algorithm, called BNDM2, turns out to be faster than the original BNDM algorithm in practical cases. Moreover the same improvements presented above can be applied also to BNDM2, obtaining the variant SBNDM2.

In [39] the authors presented also two improvements of the BNDM algorithm by combining it with the Horspool algorithm [41] according to the dominance of either methods. If the BNDM algorithm dominates then they suggest to use for shifting a simple modification of the Horspool bad character rule [41]. In particular, if the test in the main loop finds D equal to 0, the algorithm shifts the current window of $d[T[s + 2m - 1]]$ positions to the right, where the function $d : \Sigma \to \{m + 1, \ldots, 2m\}$ is defined as $d(c) = m + hbc_{_P}(c)$, for $c \in \Sigma$. If D if found to be greater than 0, then a standard loop of the BNDM algorithm is performed followed by a standard shift. The resulting algorithm is called BNDM-BMH.

Otherwise, if the Horspool algorithm dominates, the authors suggest to replace the standard naive check of Horspool algorithm with a loop of the BNDM algorithm, which generally is faster and simpler. At the end of each verification the pattern is advanced according to the shift proposed by the BNDM algorithm, which increases the shift defined by table d for the last symbol of the pattern. The resulting variant is called BMH-BNDM.

All variants of the BNDM algorithm listed above maintain the same $\mathcal{O}(n\lceil m/w \rceil)$-time and $\mathcal{O}(\sigma\lceil m/w \rceil)$-space complexity of the original algorithm.

3.3 Forward SBNDM Algorithm

The Forward-SBNDM algorithm [32] (FSBNDM for short) is the bit-parallel version of the Forward-BOM algorithm [32].

The algorithm makes use of the non-deterministic automaton $NDawg(\bar{P})$ augmented of a new initial state in order to take into account the *forward character* (character next to the right of the window) of the current window of the text. The resulting automaton has $m+1$ different states and needs $m+1$ bits to be represented. Thus the FSBNDM algorithm is able to search only for patterns with $1 \leq m < \omega$, where ω is the dimension of a word in the target machine.

For each character $c \in \Sigma$, a bit vector $B[c]$ of length $m+1$ is initialized in the preprocessing phase. The i-th bit is 1 in this vector if c appears in the reversed pattern in position $i-1$, otherwise it is 0. The bit of position 0 is always set to 1.

According to the SBNDM and FBOM algorithms the main loop of each iteration starts by initializing the state vector D with two consecutive text characters (including the forward character) as follows

$$D \leftarrow (B[T[j+1]] \ll 1) \ \& \ B[T[j]]$$

where j is the right end position of the current window of the text.

Then, if $D \neq 0$, the same kind of right to left scan in a window of size m is performed as in the SBNDM, starting from position $j-1$. Otherwise, if $D = 0$, the window is advanced m positions to the right, instead of $m-1$ positions as in the SBNDM algorithm. The resulting algorithm obtains the same $\mathcal{O}(n\lceil m/w \rceil)$-time complexity of the BNDM algorithm but turns out to be faster in practical cases, expecially for small alphabets and short patterns.

3.4 Two-Way-Non-deterministic-DAWG-Matching Algorithm

The Two-Way-Non-deterministic-DAWG-Matching algorithm (TNDM for short) was introduced in [61]. It is a two way variant of the BNDM algorithm which uses a backward search and a forward search alternately.

Specifically, when the pattern P is aligned with the text window $T[j-m+1\,..\,j]$, different cases can be distinguished. If $P[m-1]$ is equal to $T[j]$ or if $T[j]$ does not occur in P the algorithm works as in BNDM by scanning the text from right to left with the automaton $NDawg(\bar{P})$. In contrast, when $T[j] \neq P[m-1]$, but $T[j]$ occurs elsewhere in P, the TNDM algorithm scans forward starting from character $T[j]$.

The main idea is related with the Quick-Search algorithm which uses the text position immediately to the right of the current window of the text for determining the shift advancement. Because $T[j] \neq P[m-1]$ holds, we know that there will be a shift forward anyway before the next occurrence is found. Thus the algorithm examines text characters forward one by one until it finds the first position k such that $T[j\,..\,k]$ does not occur in P or $T[j\,..\,k]$ is a suffix of P. This is done by scanning the text, from left to right, with the automaton $NDawg(P)$.

If a suffix is found the algorithm continues to examine backwards starting from the text position $j-1$. This is done by resuming the standard BNDM operation. To be able to resume efficiently examining backwards, the algorithm preprocesses the length of the longest prefixes of the pattern in the case a suffix of the pattern has been

recognized by the BNDM algorithm. This preprocessing can be done in $\mathcal{O}(m)$-time and -space complexity.

Experimental results presented in [61] indicate that on the average the TNDM algorithm examines less characters than BNDM. However average running time is worse than BNDM.

In order to improve the performance the authors proposed further enhancements of the algorithm. In particular if the algorithm finds a character which does not occur in P, while scanning forward, it shifts the pattern entirely over it. This test is computationally light because after a forward scan only the last character can be missing from the pattern. The test reduces the number of fetched characters but is beneficial only for large alphabets. The resulting algorithm has been called TNBMa.

Finally in [39] the authors proposed a further improvement of the TNBM algorithm. They observed that generally the forward scan for finding suffixes dominates over the BNDM backward scan. Thus they suggested to substitute the backward BNDM check with a naive check of the occurrence, when a suffix is found. The algorithm was called Forward-Non-deterministic-DAWG-Matching (FNDM for short). It achieves better results on average than the TNDM algorithm.

All the algorihtms listed above have an $\mathcal{O}(n\lceil m/w\rceil)$-time complexity and require $\mathcal{O}(\sigma\lceil m/w\rceil)$ extra space.

3.5 Bit Parallel Wide Window Algorithm

The Bit Parallel Wide Window algorithm [38] (BPWW for short) is the bit parallel version of the Wide-Window algorithm [38].

The BPWW algorithm divides the text in $\lceil n/m\rceil$ consecutive windows of length $2m-1$. Each search attempt, on the text window $T[j-m+1\,..\,j+m-1]$ centered at position j, is divided into two steps. The first step consists in scanning the m rightmost characters of the window, i.e. the subwindow $T[j\,..\,j+m-1]$, from left to right, using the automaton $\mathit{NDawg}(P)$, until a full match occurs or the vector state which encodes the automaton becomes zero. At the end of the first step the BPWW algorithm has computed the length ℓ of the longest suffix of P in the right part of the window. If $\ell > 0$, the second step is performed. It consists in scanning the $m-1$ leftmost characters of the window, i.e. the subwindow $T[j-m+1\,..\,j-1]$, from right to left using the $\mathit{NSMA}(\bar{P})$ and starting with state ℓ of the automaton. This is done until the length of the remembered suffix of p, given by ℓ, is too small for finding an occurrence of p. Occurrences of p in T are only reported during the second phase.

The BPWW algorithm requires $\mathcal{O}(\sigma\lceil m/w\rceil)$ extra space and inspects $\mathcal{O}(n)$ text characters in the worst case. Moreover it inspects $\mathcal{O}(n\log m/m)$ characters in the average case.

3.6 Shift Vector Matching Algorithm

Many bit parallel algorithms do not remember text positions which have been checked during the previous alignments. Thus, in certain cases, if the shift is shorter than the pattern length some alignment of the pattern may be tried in vain. In [61] an algorithm, called Shift-Vector-Matching (SVM for short), was introduced which maintains partial memory. Specifically the algorithm maintains a bit vector S, called *shift-vector*,

which tells those positions where an occurrence of the pattern can or cannot occur. A text position is *rejected* if we have an evidence that an occurrence of the pattern cannot end at that position. For convention a bit set to zero denotes a text position not yet rejected.

The shift-vector has length m and maintains only information corresponding to text positions in the current window. While moving the pattern forward the algorithm shifts also the shift-vector so that the bits corresponding to the old knowledge go off from the shift-vector. Thus the bit corresponding to the end of the pattern is the lowest bit and the shift direction is to the right.

During the preprocessing phase the SVM algorithm creates a bit-vector C for each character of the alphabet. In particular for each $c \in \Sigma$, the bit-vector $C[c]$ has a zero bit on every position where the character c occurs in the pattern, and one elsewhere. Moreover the algorithm initializes the shift-vector S, in order to keep track of possible end positions of the pattern, by setting all bits of S to zero.

During the searching phase the algorithm updates the shift-vector by taking OR with the bit-vector corresponding to text character aligned with the rigthmost character of the pattern. Then, if the lowest bit of S is one, a match cannot end here and the algorithm shifts the pattern of ℓ positions to the right, where ℓ is the number of ones which preceed the rightmost zero in the shift-vector S. In addition the SVM algorithm also shifts the shift-vector of ℓ positions to the right.

Otherwise, if the lowest bit in S is zero the algorithm continues by naively checking text characters for the match. In addition the shift-vector S is updated with all characters that were fetched during verifying of alignments.

The value of ℓ is efficiently computed by using the bitwise operations $\ell = \mathsf{bsf}(\sim(S \gg 1)) + 1$, where we recall that the bsf function returns the number of zero bits before the leftmost bit set to 1.

The resulting algorithm has an $\mathcal{O}(n\lceil m/w \rceil)$ worst case time complexity and requires $\mathcal{O}(\sigma\lceil m/w \rceil)$ extra space. However the SVM algorithm is sublinear in practice, because at the same alignment it fetches the same text characters as the Horspool algorithm and can never make shorter shifts.

3.7 Average Optimal Algorithms

Fredriksson and Grabowski presented in [35] a new bit-parallel algorithm, based on Shift-Or, with an optimal average running time, as well as optimal $\mathcal{O}(n)$ worst case running time, if we assume that the pattern representation fits into a single computer word. The algorithm is called Average-Optimal-Shift-Or algorithm (AOSO for short). Experimental results presented by the authors show that the algorithm is the fastest in most of the cases in which it can be applied displacing even the BNDM family of algorithms.

Specifically the algorithm takes a parameter q, which depends on the length of the pattern. Then from the original pattern P a set $\mathcal{P}$ of q new patterns is generated, $\mathcal{P} = \{P^0, P^1, \ldots, P^{q-1}\}$, where each P^j has length $m' = \lfloor m/q \rfloor$ and is defined as

$$P^j[i] = P[j + iq], \quad j = 0, \ldots, q-1, \quad i = 0, \ldots, \lfloor m/q \rfloor - 1$$

The total length of the pattern P^j is $q\lfloor m/q \rfloor \leq m$.

The set of patterns is then searched simultaneously using the Shift-Or algorithm. All the patterns are preprocessed together, in a similar way as in the Shift-Or algorithm, as if they were concatenated in a new pattern $P' = P^0 P^1 \cdots P^{q-1}$. Moreover the algorithm initializes a mask M which has the bits of position $(j+1)m'$ set to 1, for all $j = 0, \ldots, q-1$.

During the search the set $\mathcal{P}$ is used as a filter for the pattern P, so that the algorithm needs only to scan every q-th character of the text. If the pattern P^j matches, then the $(j+1)m'$-th bit in the bit vector D is zero. This is detected with the test $(D\&M) \neq M$. The bits in M have also to be cleared in D before the shift operation, to correctly initialize the first bit corresponding to each of the successive patterns. This is done by the bitwise operation $(D\& \sim M)$.

If P^j is found in the text, the algorithm naively verifies if P also occurs, with the corresponding alignment. To efficiently identify which patterns in $\mathcal{P}$ match, the algorithm sets $D \leftarrow (D\&M)^\wedge M$, so that the $(j+1)m'$-th bit in D is set to 1 if P^j matches and all other bits are set to 0. Then the algorithm extracts the index j of the highest bit in D set to 1 with the operation

$$b \leftarrow \lfloor \log_2(D) \rfloor, \quad j \leftarrow \lfloor b/m' \rfloor$$

The corresponding text alignment is then verified. Finally, the algorithm clears the bit b in D and repeats the verification until D becomes 0.

In order to keep the total time at most $\mathcal{O}(n/q)$ on average, it is possible to select q so that $n/q = mn/\sigma^{m/q}$, i.e. $q = \mathcal{O}(m/log_\sigma m)$. the total average time is therefore $\mathcal{O}(n \log_\sigma m/m)$, which is optimal.

3.8 Bit-Parallel Algorithms with *q*-grams

The idea of using q-grams for shifting was applied successfully to bit parallel algorithms in [30].

First, the authors presented a variation of the BNDM algorithm called BNDMq. Specifically, at each alignment of the pattern with the current window of the text ending at position j, the BNDMq algorithm first reads a q-gram before testing the state vector D. This is done by initializing the state vector D at the beginning of the iteration in the following way

$$D \leftarrow B[T[j-q+1]] \ \& \ (B[T[j-q]] \ll 1) \ \& \ \cdots \ \& \ (B[T[j]] \ll (q-1)).$$

If the q-gram is not present in the pattern the algorithm quickly advances the window of $m-q+1$ positions to the right. Otherwise the inner while loop of the BNDM algorithm checks the alignment of the pattern in the right-to-left order. In the same time the loop recognizes prefixes of the pattern. The leftmost found prefix determines the next alignment of the algorithm.

The authors presented also a simplified variant of the BNDMq algorithm (called SBNDMq) along the same line of the SBNDM algorithm [61, 55] (see Section 3.2).

Finally the authors presented also an efficient q-grams variant of the Forward-Non-deterministic-DAWG-Matching algorithm [39]. The resulting algorithm is called UFNDMq, where the letter U stands for *upper bits* because the algorithm utilizes those in the state vector D.

The idea of the original algorithm is to read every m-th character c of the text while c does not occur in the pattern. If c is present in the pattern, the corresponding alignments are checked by the naive algorithm. However, while BNDM and its descendants apply the Shift-And approach, FNDM uses Shift-Or.

Along the same line of BNDMq, the UFNDMq algorithm reads q characters before testing the state vector D. Formally the state vector D is initialized as follow

$$D \leftarrow B[T[j]] \mid (B[T[j-1]] \ll 1) \mid \cdots \mid (B[T[j-q+1]] \ll (q-1)).$$

A candidate is naively checked only if at least q characters are matched.

Finally we notice that a similar approach adopted in [47] can be used for BNDMq and SBNDMq. In particular in [30] the authors developed three versions for both BNDMq and SBNDMq.

3.9 Bit-(Parallelism)^2 Algorithms for Short Patterns

Bit-parallelism^2 is a technique recently introduced in [22] which increases the *instruction level parallelism* in string matching algorithms, a measure of how many operations in an algorithm can be performed simultaneously.

The idea is quite simple: when the pattern size is small enough, in favorable situations it becomes possible to carry on in parallel the simulation of multiple copies of a same NFA or distinct NFAs, thus getting to a second level of parallelism.

Two different approaches have been presented. According to the first approach, if the algorithm searches for the pattern in fixed-size text windows then, at each attempt, it processes simultaneously two (adjacent or partially overlapping) text windows by using in parallel two copies of a same automaton.

Differently, according to the second approach, if each search attempt of the algorithm can be divided into two steps (which possibly make use of two different automata) then it executes simultaneously the two steps.

By way of demonstration the authors applied the two approaches to the bit-parallel version of the Wide-Window algorithm [38], but their approaches can be applied as well to other (more efficient) solutions based on bit-parallelism.

In both variants of the BPWW algorithm(see Section 3.5), a word of ω bits is divided into *two* blocks, each being used to encode a *NDawg*. Thus, the maximum length of the pattern gets restricted to $\lfloor \omega/2 \rfloor$. Moreover both of them searches for all occurrences of the pattern by processing text windows of fixed size $2m-1$, where m is the length of the pattern. For each window, centered at position j, the two algorithms computes the sets $\mathcal{S}_j$ and $\mathcal{P}_j$, defined as the sets all starting positions (in P) of the suffixes (and prefixes, respectively) of P aligned with position j in T.

More formally

$$\mathcal{S}_j = \{0 \le i < m \mid P[i \,..\, m-1] = T[j \,..\, j+m-1-i]\}\,;$$
$$\mathcal{P}_j = \{0 \le i < m \mid P[0 \,..\, i] = T[j-i \,..\, j]\}\,.$$

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 $|\mathcal{S}_j \cap \mathcal{P}_j|$.

In the bit-parallel implementation of the two variants of the BPWW algorithm the sets $\mathcal{P}$ and $\mathcal{S}$ are encoded by two bit masks PV and SV, respectively. The

nondeterministic automata *NDawg*(P) and *NDawg*($\bar{P}$) 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 final state configuration can be encoded by the bit masks D and $M = (1 \ll (m-1))$, so that $(D \,\&\, M) \neq 0$ 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 *NDawg*(P) or of the automaton *NDawg*($\bar{P}$). Whenever a suffix (resp., a prefix) of length $(\ell+1)$ is found (with $\ell = 0, 1, \ldots, m-1$), the bit $SV[m-1-\ell]$ (resp., the bit $PV[\ell]$) is set by one of the following bitwise operations:

$$\begin{array}{lll} SV \leftarrow SV \;\mid\; ((D \,\&\, M) \gg \ell) & & \text{(in the suffix case)} \\ PV \leftarrow PV \;\mid\; ((D \,\&\, M) \gg (m-1-\ell)) & & \text{(in the prefix case)}\,. \end{array}$$

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 $(SV \,\&\, PV)$. This can be done in $\log_2(\omega)$ operations by using a popcount function, where ω is the size of the computer word in bits (see [9]). Otherwise, if we want also to retrieve the matching positions of P in T, we can iterate over the bits set in $(SV \,\&\, PV)$ by repeatedly computing the index of the highest bit set and then masking it. The function that computes the highest bit set of a register x is $\lfloor \log_2(x) \rfloor$, and can be implemented efficiently in either a machine dependent or machine independent way (see again [9]).

In the first variant (based on the first approach), named Bit-Parallel Wide-Window^2 (BPWW2, for short), two partially overlapping windows size $2m-1$, centered at consecutive attempt positions $j-m$ and j, are processed simultaneously. Two automata are represented in a single word and updated in parallel.

Specifically, each search phase is again divided into two steps. During the first step, two copies of *NDawg*(P) are operated in parallel to compute simultaneously the sets $\mathcal{S}_{j-m}$ and $\mathcal{S}_j$. Likewise, in the second step, two copies of *NDawg*($\bar{P}$) are operated in parallel to compute the sets $\mathcal{P}_{j-m}$ and $\mathcal{P}_j$.

To properly detect suffixes in both windows, the bit mask M is initialized as

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

and transitions are performed in parallel with the following bitwise operations

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

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

Since two windows are simultaneously scanned at each search iteration, the shift becomes $2m$, doubling the length of the shift with respect to the BPWW algorithm.

The second variant of the BPWW algorithm (based on the second approach) was named Bit-Parallel^2 Wide-Window algorithm (BP2WW, for short). The idea behind it consists in processing a single window at each attempt (as in the original BPWW algorithm) but this time by scanning its left and right sides simultaneously.

Automata state configurations are again encoded simultaneously in a same bit mask D. Specifically, the most significant k bits of D encode the state of the suffix automaton *NDawg*(P), while the least significant k bits of D encode the state of

the suffix automaton $NDawg(\bar{P})$. The BP2WW algorithm uses the following bitwise operations to perform transitions[3] of both automata in parallel:

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

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

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

$$M \leftarrow (1 \ll (m+k-1)) \mid (1 \ll (m-1))\,.$$

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 BPWW algorithm, and $\mathcal{P}_j^*$ is defined as $\mathcal{P}_j^* = \{0 \le i < m \mid P[0\,..\,m-1-i] = T[j-(m-1-i)\,..\,j]\}$, so that $\mathcal{P}_j = \{0 \le i < m \mid (m-1-i) \in \mathcal{P}_j^*\}$.

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

$$PS \leftarrow PS \mid ((D \,\&\, M) \gg \ell)\,.$$

At the end of each iteration, the bit masks SV and PV are retrieved from PS with the following bitwise operations:

$$PV \leftarrow reverse(PS) \gg (\omega - m)\,, \qquad SV \leftarrow PS \gg k\,,$$

In fact, to obtain the correct value of PV we used bit-reversal modulo m, which has been easily achieved by right shifting $reverse(PS)$ by $(\omega - m)$ positions. We recall that the *reverse* function can be implemented efficiently with $\mathcal{O}(\log_2(\omega))$ operations.

Both BPWW2 and BP2WW algorithms need $\lceil m/\omega \rceil$ words to represent all bit masks and have an $\mathcal{O}(n\lceil m/\omega \rceil + \lfloor n/m \rfloor \log_2(\omega))$ worst case time complexity.

3.10 The Bit-Parallel Length Invariant Matcher

The general problem in all bit parallel algorithms is the limitation defined on the length of the input pattern, which does not permit efficient searching of strings longer than the computer word size.

In [49] the author proposed a method, based on bit parallelism, with the aim of searching patterns independently of their lengths. The algorithm was called Bit-Parallel Length Invariant Matcher (BLIM for short). In contrast with the previous bit parallel algorithms, which require the pattern length not to exceed the computer word size, BLIM defines a unique way of handling strings of any length.

Given a pattern P, of length m, the algorithm ideally computes an alignment matrix A which consists of ω rows, where ω is the size of a word in the target machine. Each row row_i , for $0 \le i < \omega$, contains the pattern right shifted by i characters. Thus, A contains $wsize = \omega + m - 1$ columns. During the preprocessing phase the algorithm computes a mask matrix M of size $|\Sigma| \times wsize$, where $M[c,h] = b_{\omega-1} \ldots b_1 b_0$

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

is a bit vector of ω bits, for $c \in \Sigma$ and $0 \leq h < wsize$. Specifically the i-th bit, b_i of $M[c, h]$ is defined as

$$b_i = \begin{cases} 0 & \text{if } (0 \leq h - i < m) \text{ and } (c = P[h - i]) \\ 1 & \text{otherwise .} \end{cases}$$

The main idea of BLIM is to slide the alignment matrix over the text, on windows of size *wsize*, and at each attempt check for any possible placements of the pattern.

The characters of the window are visited in order to perform the minimum number of character accesses. Specifically the algorithm checks the characters at positions $m - i, 2m - i, \ldots, km - i$, where $km - i < wsize$, for $i = 1, 2, \ldots, m$ in order. The main idea behind this ordering is to investigate the window in such a way that at each character access a maximum number of alignments is checked. The scan order is precomputed and stored in a vector S of size *wsize*.

When the window is located at $T[i \,..\, i + wsize - 1]$, a flag variable F is initialized to the mask value $M[T[i+S[0]], S[0]]$. The traversal of other characters of the window continues by performing the following basic bitwise operation

$$F \leftarrow F \& M[T[i + S[j]], S[j]]$$

for $j = 1, \ldots, wsize$, till the flag F becomes 0 or all the characters are visited. If F becomes 0, this implies that the pattern does not exist on the window. Otherwise, one or more occurrences of the pattern are detected at the investigated window. In that case, the 1 bits of the flag F tells the exact positions of occurrences.

At the end of each attempt the BLIM algorithm uses the shift mechanism proposed in the Quick-Search algorithm [42]. The immediate text character following the window determines the shift amount. If that character is included in the pattern then the shift amount is $wsize - k$, where $k = \max\{i \mid P[i] = T[s + wsize]\}$, otherwise the shift value is equal to $wsize + 1$.

The BLIM algorithm has a $\mathcal{O}(\lceil n/\omega \rceil(\omega+m-1))$ overall worst case time complexity and requires $\mathcal{O}(\Sigma \times (\omega + m - 1))$-extra space.

3.11 Bit-Parallel Algorithms for Long Patterns

In [57] the authors introduced an efficient method, based on bit-parallelism, to search for patterns longer than w. Their approach 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 this belongs to an occurrence of the whole pattern. However, besides the costs of the additional verification phase, a drawback of this approach is that, in the case of the BNDM algorithm, the maximum possible shift length cannot exceed w, which could be much smaller than m.

Later in [61] another approach for long patterns was introduced, called LBNDM. In this case the pattern is partitioned in $\lfloor m/k \rfloor$ consecutive substrings, each consisting in $k = \lfloor (m - 1)/\omega \rfloor + 1$ characters. The $m - k\lfloor m/k \rfloor$ remaining characters are left to either end of the pattern. Then the algorithm constructs a superimposed pattern P' of length $\lfloor m/k \rfloor$, where $P'[i]$ is a class of characters including all characters in the i-th substring, for $0 \leq i < \lfloor m/k \rfloor$.

The idea is to search first the superimposed pattern in the text, so that only every k-th character of the text is examined. This filtration phase is done with the standard BNDM algorithm, where only the k-th characters of the text are inspected. When an occurrence of the superimposed pattern is found the occurrence of the original pattern must be verified.

The shifts of the LBNDM algorithm are multiples of k. To get a real advantage of shifts longer than that proposed by the approach of Navarro and Raffinot, the pattern length should be at least about two times ω.

More recently Durian *et al.* presented in [31] another efficient algorithm for simulating the suffix automaton in the case of long patterns. The algorithm is called BNDM with eXtended Shift (BXS). The idea is to cut the pattern into $\lceil m/w \rceil$ consecutive substrings of length w except for the rightmost piece which may be shorter. Then the substrings are superimposed getting a superimposed pattern of length w. In each position of the superimposed pattern a character from any piece (in corresponding position) is accepted. Then a modified version of BNDM is used for searching consecutive occurrences of the superimposed pattern using bit vectors of length w but still shifting the pattern by up to m positions. The main modfication in the automaton simulation consists in moving the rightmost bit, when set, to the

first position of the bit array, thus simulating a circular automaton. Like in the case of the LBNDM, algorithm the BXS algorithm works as a filter algorithm, thus an additional verification phase is needed when a candidate occurrence has been located.

3.12 A bit-parallel algorithm for small alphabets

The bit-parallel algorithm for small alphabets (SABP for short) [67] consists in scanning the text with a window of size $\ell = \max\{m+1, \omega\}$. At each attempt, where the window is positioned on $T[j\,..\,j+\ell-1]$ it maintains a vector of ω bits whose bit at position $\omega - 1 - i$ is set to 0 if P cannot be equal to $T[j+i\,..\,j+m-1+i]$.

The preprocessing phase consists in computing:

- an array T of $\max\{m, \omega\} \times \sigma$ vectors of ω bits as follows:

$$T[j, c]_{\omega-1-i} = \begin{cases} 0 & \text{if } 0 \leq j - i < \omega \text{ and } P[j-i] \neq c \\ 1 & \text{if } j - i \notin [0, \omega) \text{ or } P[j-i] = c \end{cases}$$

  for $0 \leq j < \ell$, $0 \leq i < \omega$ and $c \in \Sigma$.
- an array T' of σ vectors of ω bits as follows:

$$T'[c] = (T[m-1, c] \gg 1) \quad | \quad 10^{\omega-1}$$

  for $c \in \Sigma$.
- the Quick Search bad character rule qbc_P [42].

Then during the searching phase the algorithm maintains a vector F of ω bits such that when the window of size ℓ is positioned on $T[j\,..\,j+\ell-1]$ the bit at position $\omega - 1 - i$ of F is equal to 0 if P cannot be equal to $T[j+i\,..\,j+m-1+i]$ for $0 \leq j \leq n - \ell$ and $0 \leq i < \omega$. Initially all the bits of F are set to 1. At each attempt the algorithm first scan $T[j+m-1]$ and $T[j+\omega-1]$ as follows:

$$F = F \quad \& \quad T[m-1, T[j+m-1]] \quad \& \quad T[\omega-1, T[j+\omega-1]]$$

then it scans $T[j+m-2]$ and $T[j+m]$ as follows:

$$F = F \quad \& \quad T[m-2, T[j+m-2]] \quad \& \quad T'[j+m]$$

and finally it scans $T[j+k]$ for $k = m-3, \ldots, 0$ as follows:

$$F = F \quad \& \quad T[k, T[j+k]]$$

while $F_{\omega-1} = 1$. If all the characters have been scanned and $F_{\omega-1} = 1$ then an occurrence of the pattern is reported and $F_{\omega-1}$ is set to 0. In all cases a shift of the window is performed by taking the maximum between the Quick Search bad character rule and the difference between ω and the position of the righmost bit of value 1 in F which can be computed by $\omega - \lfloor \log_2 F \rfloor$. The bit-vector F is shifted accordingly.

The preprocessing phase of the SABP algorithm has an $\mathcal{O}(m\sigma)$ time and space complexity and the searching phase has an $\mathcal{O}(mn)$ time complexity.

3.13 Tighter packing for bit-parallelism

In order to overcome the problem due to handling long patterns with bit-parallelism, in [22] a new encoding of the configurations of non-deterministic automata for a given pattern P of length m was presented, which on the average requires less than m bits and is still suitable to be used within the bit-parallel framework. The effect is that bit-parallel string matching algorithms based on such encoding scale much better as m grows, at the price of a larger space complexity (at most of a factor σ). The authors illustrated the application of the encoding to the Shift-And and the BNDM algorithms, obtaining two variants named Factorized-Shift-And and Factorized-BNDM (F-Shift-And and F-BNDM for short). However the encoding can also be applied to other variants of the BNDM algorithm as well.

The encoding has 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) which 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 and is based on the 1-factorization of the pattern.

More specifically, given a pattern $P \in \Sigma^m$, a 1-factorization of size k of P is a sequence $\langle u_1, u_2, \ldots, u_k \rangle$ of nonempty substrings of P such that $P = u_1 u_2 \cdots u_k$ and each factor u_j contains at most *one* occurrence for any of the characters in the alphabet Σ, for $j = 1, \ldots, k$. A 1-factorization of P is *minimal* if such is its size.

For $x \in \Sigma^*$, let $\mathit{first}(x) = x[0]$ and $\mathit{last}(x) = x[|x|-1]$. It can easily be checked that a 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of P is minimal if $\mathit{first}(u_{i+1})$ occurs in u_i, for $i = 1, \ldots, k-1$.

Observe, also, that $\lceil \frac{m}{\sigma} \rceil \leq k \leq m$ holds, for any 1-factorization of size k of a string $P \in \Sigma^m$, where $\sigma = |\Sigma|$. The worst case occurs when $P = a^m$, in which case P has only the 1-factorization of size m whose factors are all equal to the single character string a.

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 nonstarting states of the canonical automaton $SMA(P) = (Q, \Sigma, \delta, q_0, F)$ for the language $\Sigma^* P$.

Hence, for any alphabet symbol a the set of states Q_i contains at most one state with an incoming arrow labeled a. Indicate with symbol $q_{i,a}$ the unique state q of $SMA(P)$ with $q \in Q_i$, and q has an incoming edge labeled a.

In the F-Shift-And algorithm the configuration $\delta^*(q_0, Sa)$ is 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 $q_{i,a} \in \delta^*(q_0, Sa)$.

For $i = 1, \ldots, k-1$, we put $\overline{u}_i = u_i.first(u_{i+1})$. 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.

In order to encode the 2-grams present in the closure of the factors u_i the algorithm makes use of 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_1c_2 is present in $\overline{u}_i$ or, equivalently, iff

$$\begin{aligned}(last(u_i) \neq c_1 \wedge q_{i,c_2} \in \delta(q_{i,c_1}, c_2)) \vee \\ (i < k \wedge last(u_i) = c_1 \wedge q_{i+1,c_2} \in \delta(q_{i,c_1}, c_2))\,,\end{aligned} \tag{4}$$

for every 2-gram $c_1c_2 \in \Sigma^2$ 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}, the algorithm makes also use of an array L, of size $|\Sigma|$, 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 $last(u_i) = c$, for $i = 1, \ldots, k$.

The matrix B and the array L, which in total require $(|\Sigma|^2 + |\Sigma|)k$ bits, are used to compute the transition $(D, a) \xrightarrow{SMA} (D', c)$ on character c. In particular D' can be computed from D by the following bitwise operations:

$$\text{(i) } D \leftarrow D \;\&\; B[a, c]; \quad \text{(ii) } H \leftarrow D \;\&\; L[a]; \quad \text{(iii) } D \leftarrow (D \;\&\; \sim H) | (H \ll 1)\,.$$

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 test can then be implemented with the test $D \;\&\; M \;\&\; L[a] \neq 0^k$, where $M = (1 \ll (k-1))$.

The same considerations also hold for the encoding of the factor automaton $Dawg(P)$ in the F-BNDM algorithm. The only difference is in the handling of the initial state. In the case of the automaton $SMA(P)$, 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 $Dawg(P)$, as the initial state has an ε-transition to each state, all the bits in D must be set, as in the BNDM algorithm.

The preprocessing procedure which builds the arrays B and L has a time complexity of $\mathcal{O}(|\Sigma|^2 + m)$. The variants of the Shift-And and BNDM algorithms based on the encoding of the configurations of the automata $SMA(P)$ and $Dawg(P)$ (algorithms F-Shift-And and F-BNDM, respectively) have a worst-case time complexities of $\mathcal{O}(n\lceil k/\omega \rceil)$ and $\mathcal{O}(nm\lceil k/\omega \rceil)$, respectively, while their space complexity is $\mathcal{O}(|\Sigma|^2\lceil k/\omega \rceil)$, where k is the size of a minimal 1-factorization of the pattern.

4 Multiple String Matching

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 pattern matching problem* is to determine all the occurrences in T of the patterns in $\mathcal{P}$.

Multiple string matching is an important problem in many application areas of computer science. For example, in computational biology, with the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application and there is an increasing demand for fast computer methods for analysis and data retrieval. Although there are various kinds of comparison tools which provide aligning and approximate matching, most of them are based on exact matching in order to speed up the process. Another important usage of multiple pattern matching algorithms appears in network intrusion detection systems as well as in anti-virus software. The major performance bottleneck of the regarding solutions to these problems is to achieve high-speed matching required to detect malicious patterns of ever growing sets.

The first linear solution for the multiple pattern matching problem based on finite automata is due to Aho and Corasick in [1]. The Aho-Chorasick 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 [48]. The optimal average complexity of the problem is $\mathcal{O}(n \log_\sigma(rl_{min})/l_{min})$ [56], where l_{min} is the length of the shortest pattern in the set $\mathcal{P}$; this bound has been achieved by algorithms based on the suffix automaton induced by the DAWG data structure, namely the Backward-DAWG-Matching (BDM) and Set-Backward-DAWG-Matching (SBDM) algorithms [27, 59].

To simulate efficiently an NFA with the bit-parallelism technique, the states of the automaton must be mapped into the positions of a bit-vector by a suitable topological ordering of the NFA.[4]

In the case of a single pattern, the construction of the topological ordering is quite simple, since it is unique [10]. 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 restriction that all the patterns have the same length [65], or in a sequential fashion [57]. 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 $\mathcal{P}$.

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. In particular when a parallel simulation is carried out the shift is always of $k = r$ bits.

The above techniques used to extend string matching algorithms based on bit-parallelism to the multiple-string matching problem consists, on a conceptual basis,

[4] 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.

in sequentially concatenating the automata for each pattern. The drawback of this method is that it is not possible to exploit the prefix redundancy in the patterns, a property which can be significant in the case of small alphabets. The trie and the DAWG data structures make it possible to factor common prefixes in the patterns. However, because of the lack of regularity in such structures, 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 presented in [20] a bit-parallel simulation of the Aho-Chorasick NFA which is able to encode variable length shifts. Their solution is based on a suitable topological ordering of the NFA, called *weakly safe* topological ordering, and distinguish between 1-bit edges, i.e. edges (p, q) such that $\pi q - \pi(p) = 1$, and long-bit edges, i.e. edges (p, q) such that $\pi q - \pi(p) > 1$. The simulation of long-bit edges is then simulated using the carry property of addition. In particular a topological ordering π is said to be *weakly safe* if, for each $c \in \Sigma$, the π-intervals of any two distinct long-bit edges labeled by a same character and not originating from the root of T are disjoint.

They proposed an algorithm, called Multiple-Trie-Shift-And, with 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. The construction of the safe topological ordering is based on a DFS approach and runs in $O(L)$-time and -space, under suitable hypotheses. However, such topological orderings do not always exist and the problem of finding one is probably even intractable.

More recently Cantone, Faro and Giaquinta presented in [23] a new more general approach to the efficient bit-parallel simulation of the Aho-Chorasick NFAs and suffix NFAs. When the prefix redundancy is nonnegligible, this method yields a representation that requires smaller bit-vectors and, correspondingly, less words. Therefore, if we restrict to single-word bit-vectors, it results that more patterns can be packed into a word. The construction is based on a result for the Glushkov automaton [60], which however requires exponential space in the number of states in the NFA to encode the transition function. They show that, by exploiting the relation between active states of the NFA and its associated failure function, it is possible to represent the transition function in polynomial space using a similar encoding.

Indicate with symbol $B(c)$, for $c \in \Sigma$, the set of states with an incoming transition labeled by c, and with symbol $Follow(q)$, for $q \in Q$, the set of states reachable from state q with one transition over a character in Σ. Their solution is based on the property [60] that, for every $q \in Q$, $D \subseteq Q$, and $c \in \Sigma$, we have $\delta(q, c) = Follow(q) \cap B(c)$ and $\delta(D, c) = \phi(D) \cap B(c)$, which is particularly suitable for bit-parallelism, as set intersection can be readily implemented by the bitwise and operation.

Moreover in order to find an efficient way of storing and accessing the maps $\phi()$ and $B()$ the authors show that 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 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 state that $\delta(D, c) = \dot{\Phi}(lead(D)) \cap B(c)$, which in turn translates readily into the bit-parallel assignment $D \leftarrow \dot{\Phi}[lead(D)]$ & $B[c]$.

They also presented two simple algorithms, based on such a technique, for searching a set $\mathcal{P}$ of patterns in a text T of length n over an alphabet Σ of size σ. The algorithms, named Log-And and Backward-Log-And, require $\mathcal{O}((m+\sigma)\lceil m/w \rceil)$-space, and work in $\mathcal{O}(n\lceil m/w \rceil)$ and $\mathcal{O}(n\lceil m/w \rceil l_{min})$ worst-case searching time, respectively, where w is the number of bits in a computer word, m is the number of states of the automaton, and l_{min} is the length of the shortest pattern in $\mathcal{P}$.

5 Approximate String Matching

Approximate pattern matching is a classic problem in computer science, with applications in various areas, such as spelling correction, bioinformatics, signal processing, musical information retrieval. It has been actively studied since the sixties [54].

Approximate pattern matching consists in general in searching for substrings of a text T that are within a predefined edit distance threshold from a given pattern P.

Let $d(x, y)$ denote the approximate distance between the strings x and y over a common alphabet Σ, and let k be the maximum allowed distance. Using this notation, the task of approximate string matching is to find all positions j in the text such that $d(P, T[i..j]) \leq k$, for some $i \leq j$.

Perhaps the most common form of edit distance is the Levenshtein edit distance [50], which is defined as the minimum number of single-character insertions, deletions and substitutions needed in order to make x and y equal. Other common forms of edit distances have been studied over the years and solved by using bit-parallelism. In what follows we survey solutions on approximate string matching under the Damerau distance, the Swap distance and allowing for gaps.

5.1 String Matching with Levenshtein Distance

Perhaps the most common form of edit distance between two strings P and T is the Levenshtein edit distance [50], which is defined as the minimum number of single-character insertions, deletions and substitutions needed in order to make P and T equal.

In this case the dynamic programming algorithm fills a $(|P|+1)\times(|T|+1)$ dynamic programming table D, where at the end each cell $D[i, j]$ will hold the edit distance between $P[0..i]$ and $T[0..j]$.

Although the algorithm is not very efficient it is among the most flexible ones to adapt to different distance functions.

$$D[i,j] = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ D[i-1, j-1] & \text{if } i, j > 0 \text{ and } P[i] = T[j] \\ 1 + \min(D[i-1, j], D[i, j-1], D[i-1, j-1]) & \text{otherwise} \end{cases} \tag{5}$$

Instead of computing the edit distance between strings P and T, the dynamic programming algorithm can be changed to find approximate occurrences of P somewhere inside T by changing the boundary condition $D[i, j] = j$ with $D[i, j] = 0$, when $i = 0$, that is the empty pattern matches with zero errors at any text position. It was converted into a search algorithm only in by Seller [62].

During the last decade, algorithms based on bit-parallelism have emerged as the fastest approximate string matching algorithms in practice for the Levenshtein edit distance [54].

The algorithm of Meyers [53] is based on representing the dynamic programming table D with vertical, horizontal and diagonal differences. This is done by using the length-m bit-vectors VP, VN (vertical positive and vertical negative vector), HP and HN (horizontal positive and horizontal negative vector). Specifically $VP[i] = 1$ at text position j iff $D[i,j] - D[i-1,j] = 1$, while $VN[i] = 1$ at text position j iff $D[i,j] - D[i-1,j] = -1$. A similar definition holds for HP and HN. In addition a diagonal delta vector R is maintained, where $R[i] = 1$ at text position j iff $D[i,j] = D[i-1,j-1]$. Initially $VP = 1^m$ and $VN = 0^m$. The complete formula for computing the updated vectors at text position j is

$$\begin{aligned}
R' &= (((B[T[j]] \;\&\; VP) + VP) \;\wedge VP) \mid B[T[j]] \mid VN \\
HP' &= VN \mid \;\sim (R' \mid VP) \\
HN' &= VP \;\&\; R' \\
VP' &= (HN' \ll 1) \mid \;\sim (R' \mid (HP' \ll 1)) \\
VN' &= (HP' \ll 1) \;\&\; R'
\end{aligned}$$

The current value of the dynamic programming cell $D[m,j]$ can be updated at each text position j by using the horizontal delta vectors (the initial value is $D[m,0] = m$). A match of the pattern with at most k errors is found whenever $D[m,j] \leq k$. If $m \leq w$, the run time of this algorithm is $O(n)$ as there is again only a constant number of operations per text character. The general run time is $O(\lceil m/w \rceil n)$ as a vector of length m may be simulated in $O(\lceil m/w \rceil)$ time using $O(\lceil m/w \rceil)$ bit-vectors of length w.

The bit-parallel approximate string matching algorithm of Wu and Manber [64] is based on representing a non-deterministic finite automaton (NFA) by using bit-vectors. The automaton has $(k+1)$ rows, numbered from 0 to k, and each row contains m states. Let us denote the automaton as D, its row d as D_r and the state i on its row r as $D_r[i]$. The state $D_r[i]$ is active after reading the text up to the j-th character if and only if $ed(P[0..i], T[h..j]) \leq d$ for some $h \leq j$. An occurrence of the pattern with at most k errors is found when the state $D_k[m]$ is active. Assume for now that $m \leq w$. Wu and Manber represent each row D_r as a length-m bit-vector, where the i-th bit tells whether the state $D_r[i]$ is active or not. In addition they build a length-m match vector for each character in the alphabet. We denote the match vector for the character c as $B[c]$. The i-th bit of $B[c]$ is set if and only if $P[i] = c$. Initially each vector D_r has the value $0^{m-r}1^r$ (this corresponds to the boundary conditions in Recurrence 1). The formula to compute the updated values D'_r from the row-vectors D_r at text position j is the following

$$\begin{aligned}
D'_0 &= ((D_0 \ll 1) \mid 1) \;\&\; B[T[j]] \\
D'_r &= ((D_r \ll 1) \;\&\; B[T[j]]) \mid D_{r-1} \mid (D_{r-1} \ll 1) \mid (D'_{r-1} \ll 1) \mid 1, \text{ for } 1 \leq r \leq k.
\end{aligned}$$

When $m \leq w$, the running time of this algorithm is $O(kn)$ as there are $O(k)$ operations per text character. The general run time is $O(kn\lceil m/w \rceil)$.

In [11] Baeza-Yates and Navarro found a bit parallel formula for a diagonal parallelization of the NFA. They packed the states of the automaton along diagonals

instead of rows or columns which run in the same direction of the diagonal arrows. There are $m-k+1$ complete diagonals, the others are not really necessary, which are numbered from 0 to $m-k$. To describe the status of the i-th diagonal it sufices to record the position of the first active state in it. Thus the number D_i is the row of the first active state in diagonal i all the subsequent states in the diagonal are active because of the transitions. If the first active state on the i-th diagonal is f_i, then Baeza-Yates and Navarro represent the diagonal as the bit-sequence $D_i = 0^{k+1-f_i}1^{f_i}$. A match with at most k errors is found whenever $f_{m-k} < k+1$. The bit-sequences are stored consecutively with a single separator zero-bit between two consecutive states. Let $\bar{D}$ denote the complete diagonal representation. Then $\bar{D}$ is the length-$(k+2)(m-k)$ bit-sequence $0D_10D_20...0D_{m-k}$. We assume for now that $(k+2)(m-k) \leq w$ so that $\bar{D}$ fits into a single bit-vector. Baeza-Yates and Navarro encode also the pattern match vectors differently. Let $\bar{B[c]}$ be their pattern match vector for the character. first of all the role of the bits is reversed: a 0-bit denotes a match and a 1-bit a mismatch. To align the matches with the diagonals in $\bar{D}$, $\bar{B[c]}$ has the form $0B_10B_20...0B_{m-k}$, where $B_i = \sim B[c][i..i+k]$. Initially no diagonal has active states and so $\bar{D} = (01^{k+1})^{m-k}$. The formula for updating $\bar{D}$ at text position j is:

$$
\begin{aligned}
x &= (\bar{D} \gg (k+2)) \mid \bar{B[T[j]]} \\
\bar{D'} &= ((D \ll 1) \mid (0^{k+1}1)^{m-k} \;\&\; (\bar{D} \ll (k+3)) \mid (0^{k+1}1)^{m-k-1}01^{k+1} \;\& \\
&\quad (((x + (0^{k+1}1)^{m-k}) \;\&\; x) \gg 1) \;\&\; (01^{k+1})^{m-k}
\end{aligned}
$$

If $(k+2)(m-k) \leq w$, the run time of this algorithm is $O(n)$ as there is only a constant number of operations per text character. The general run time is $O(\lceil km/w \rceil n)$ as a vector of length $(k+2)(m-k)$ may be simulated in $O(\lceil km/w \rceil)$ time using $O(\lceil km/w \rceil)$ bit-vectors of length w.

Later Hyyro proposed a new variant [43] of the bit-parallel NFA of Baeza-Yates and Navarro for approximate string matching. The algorithm decreases the original NFA complexity to $(m-k)(k+1)$, and also give a slightly more efficient simulation algorithm for the NFA. In experiments the method by Hyyro turns out to be often noticeably more efficient than the original algorithm under moderate values of k and m.

In [44] Hyyro *et al.* showed how multiple (short) patterns can be packed in a single computer word so as to search for multiple patterns simultaneously obtaining $\mathcal{O}(\lceil r/\lfloor w/m \rfloor \rceil n)$ time to search for r patterns of length $m < w$.

5.2 Damerau Distance

Another common form of edit distance is the Damerau edit distance [28], which is in principle an extension of the Levenshtein distance by permitting also the swap of two adjacent characters.

Navarro [55] has modified the Wu-Manber algorithm [64] described in Section 5.1 to use the Damerau distance by appending the automaton to have a temporary state vector S_r row to keep track of the positions where transposition may occur. In particular T_r is initialized to 0^m, for $0 \leq r \leq k$. Then we have

$$
\begin{aligned}
D'_0 &= ((D_0 \ll 1) \mid 1) \;\&\; B[T[j]] \\
D'_r &= ((D_r \ll 1) \;\&\; B[T[j]]) \mid D_{r-1} \mid (D_{r-1} \ll 1) \mid (D'_{r-1} \ll 1) \mid \\
&\quad (T_r \;\&\; (B[T[j]] \ll 1)) \mid 1, \text{ for } 1 \leq r \leq k. \\
T'_r &= (D_{r-1} \ll 2) \;\&\; B[T[j]], \text{ for } 1 \leq r \leq k.
\end{aligned}
$$

The formula adds $6k$ operations into the basic version for the Levenshtein edit distance

Later Hyyro proposed in [45] a different modification of the Wu-Manber algorithm which adds a total of 6 operations into the basic version for the Levenshtein edit distance. Therefore it makes the same number of operations as Navarro's version when $k = 1$, and wins when $k > 1$.

Moreover in [45] the author gives formulas for extending also the algorithm of Baeza-Yates and Navarro [11] and the algorithm of Myers [53] to use the Damerau distance. The resulting algorithms add 7 extra operations and 6 extra operations, respectively. Thus they do not affect the computational complexity of the original algorithms.

5.3 String Matching Allowing for Swaps

The *Pattern Matching problem with Swaps* (Swap Matching problem, for short) is a well-studied variant of the classic Pattern Matching problem. It consists in finding all occurrences, up to character swaps, of a pattern P of length m in a text T of length n, with P and T sequences of characters drawn from a same finite alphabet Σ of size σ. More precisely, the pattern is said to *swap-match the text at a given location* j if adjacent pattern characters can be swapped, if necessary, so as to make it identical to the substring of the text 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 adjacent equal characters are not allowed to be swapped.

The swap matching problem was introduced in 1995 as one of the open problems in nonstandard string matching [52]. This problem is of relevance in practical applications such as text and music retrieval, data mining, network security, and many others. Following [8], we also mention a particularly important application of the swap matching problem in biological computing, specifically in the process of translation in molecular biology, with the genetic triplets (otherwise called *codons*).

The first nontrivial result was reported by Amir *et al.* [3], who provided an algorithm which achieves $\mathcal{O}(nm^{\frac{1}{3}} \log m)$-time in the case of alphabet sets of size 2, showing also that the case of alphabets of size exceeding 2 can be reduced to that of size 2 with a $\mathcal{O}(\log^2 \sigma)$-time overhead, subsequently reduced to $\mathcal{O}(\log \sigma)$ in the journal version [4]. Amir *et al.* [6] studied some rather restrictive cases in which a $\mathcal{O}(m \log^2 m)$-time algorithm can be obtained. More recently, Amir *et al.* [5] solved the swap matching problem in $\mathcal{O}(n \log m \log \sigma)$-time. We observe that the above solutions are all based on the fast Fourier transform (FFT) technique.

In 2008 the first attempt to provide an efficient solution to the swap matching problem without using the FFT technique has been presented by Iliopoulos and Rahman in [46]. They introduced a new graph-theoretic approach to model the problem and devised an efficient algorithm, based on the bit-parallelism technique [10], which runs in $\mathcal{O}((n + m) \log m)$-time, provided that the pattern size is comparable to the word size in the target machine.

More recently, in 2009, Cantone and Faro [21] presented a first approach for solving the swap matching problem with short patterns in linear time. Their algorithm, named Cross-Sampling, though characterized by a $\mathcal{O}(nm)$ worst-case time complexity, admits an efficient bit-parallel implementation, named BP-Cross-Sampling, which achieves $\mathcal{O}(n)$ worst-case time and $\mathcal{O}(\sigma)$ space complexity in the case of short patterns fitting in few machine words.

Th BPCS algorithm is a natural generalization of the SA algorithm to the swap matching problem. It uses vectors of m bits, D_j and D'_j respectively. The i-th bit of D_j is set to 1 if P_i matches T_j, whereas the i-th bit of D'_j is set to 1 if P_{i-1} matches T_{j-1} and $P[i] = T[j+1]$. All remaining bits in the bit vectors are set to 0. As in the SA algorithm, for each character c of the alphabet Σ, a bit mask $M[c]$ is maintained, where the i-th bit is set to 1 if $P[i] = c$.

The bit vectors D_0 and D'_0 are initialized to 0^m. Then the algorithm scans the text from the first character to the last one and, for each position $j \geq 0$, it computes the bit vector D_j in terms of D_{j-1} and D'_{j-1}, by performing the following bitwise operations:

$$D_j \leftarrow (((D_{j-1} \ll 1) \mid 1) \,\&\, M[T[j]]) \mid ((D'_{j-1} \ll 1) \,\&\, M[T[j-1]])$$
$$D'_j \leftarrow ((D_{j-1} \ll 1) \mid 1) \,\&\, M[T[j+1]]$$

During the j-th iteration, we report a swap match at position j, provided that the leftmost bit of D_j is set to 1, i.e., if $(D_j \,\&\, 10^{m-1}) \neq 0^m$.

In practice, we can use only two vectors to maintain D_j and D'_j, for $j = 0, \ldots, n-1$. Thus during iteration j of the algorithm, vector D_{j-1} is transformed into vector D_j, whereas vector D'_{j-1} is transformed into vector D'_j.

In a subsequent paper[15] a more efficient algorithm, named Backward-Cross-Sampling (BCS) and based on a similar structure as the one of the Cross-Sampling algorithm, has been proposed. The BCS algorithm scans the text from right to left and has a $\mathcal{O}(nm^2)$-time complexity, whereas its bit-parallel implementation, named Bit-Parallel Backward-Cross-Sampling (BPBCS), works in $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity.

The BPCS and BPBCS algorithms could be also modified in order to solve the more general *Approximate Pattern Matching problem with Swaps*. Such a problem seeks to compute, for each text location j, the number of swaps necessary to convert the pattern to the substring of length m ending at j, provided there is a swapped matching at j.

A straightforward solution to the approximate swap matching problem consists in searching for all occurrences (with swap) of the input pattern P, using any algorithm for the standard swap matching problem. Once a swap match is found, to get the number of swaps, it is sufficient to count the number of mismatches between the pattern and its swap occurrence in the text and then divide it by 2.

In [7], Amir *et al.* presented an algorithm that counts in time $\mathcal{O}(\log m \log \sigma)$ the number of swaps at every location containing a swapped matching, thus solving the approximate pattern matching problem with swaps in $\mathcal{O}(n \log m \log \sigma)$-time.

In [16], the authors extended the BPCS and BPBCS algorithms designing two algorithms for the Approximate Swap Matching problem, which achieve $\mathcal{O}(n)$ and $\mathcal{O}(nm)$ worst-case time, respectively, and $\mathcal{O}(\sigma)$-space complexity for patterns having length similar to the word-size of the target machine.

In this case the two vectors $\bar{D}_j$ and $\bar{D}'_j$ are maintained as a list of q bits, where $q = \log(\lfloor m/2 \rfloor + 1) + 1$ and m is the length of the pattern. If P_i has a swap occurrence ending at position j of the text, with k swaps, then 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 are set so as to contain the value k (notice that we need exactly $\log(\lfloor m/2 \rfloor + 1)$ bits to represent a value between 0 and $\lfloor m/2 \rfloor$). Otherwise the rightmost bit of the i-th block of $\bar{D}_j$ is set to 0. If $m \log(\lfloor m/2 \rfloor + 1) + m \leq w$, then the entire list fits in a single computer word, otherwise we need $\lceil m(\log(\lfloor m/2 \rfloor + 1)/w \rceil$ computer words to represent the sets $\bar{D}_j$ and $\bar{D}'_j$.

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 $P[i] = c$. Moreover, for each character $c \in \Sigma$, the algorithm maintains, a bit mask $B[c]$ whose i-th block have all bits set to 1 if $P[i] = c$, whereas all remaining bits are set to 0.

The generalization of the BPBCS algorithm to the approximate swap matching problem requires only $\log(\lfloor m/2 \rfloor + 1)$ bits to implement the counter for keeping track of the number of swaps. This compares favorably with the BPACS algorithm which uses instead m counters of $\log(\lfloor m/2 \rfloor + 1)$ bits, one for each prefix of the pattern.

The resulting BPABCS algorithm achieves a $\mathcal{O}(\lceil nm^2/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma \lceil m/w \rceil + \log(\lfloor m/2 \rfloor + 1))$ extra-space. If the pattern fits in few machine words, then the algorithm finds all swapped matches and their corresponding counts in $\mathcal{O}(nm)$-time and $\mathcal{O}(\sigma)$ extra-space.

5.4 String Matching with class of characters and general gaps

The δ-approximate string matching problem with α-bounded gaps (or (δ, α)-matching) [26, 25, 17] arises in many questions in music information retrieval and music analysis. This is particularly true, for instance, in the context of monophonic music, when one wants to retrieve occurrences of a given melody from a complex musical score. It finds also large applications in computational biology.

More formally, let Σ be a finite alphabet of *integer numbers* and let δ and α be nonnegative integers. Two symbols a and b of Σ are said to be δ-approximate, in which case we write $a =_\delta b$, if $|a - b| \leq \delta$. Given a pattern P of length m and a text T of length n over the alphabet Σ, by a δ-approximate occurrence with α bounded gaps of P in T, or simply a (δ, α)-occurrence of P in T, we mean a sequence $(i_0, i_1, \ldots, i_{m-1})$ of indices such that $0 \leq i_0 < i_1 < \cdots < i_{m-1} < n$, $T[i_j] =_\delta P[j]$, for $0 \leq j < m$, and $i_h - i_{h-1} \leq \alpha + 1$, for $0 < h < m$, provided that $m > 1$.

Given an index i, with $0 \leq i < n$, a (δ, α)-occurrence of P at position i in T is a (δ, α)-occurrence $(i_0, i_1, \ldots, i_{m-1})$ of P in T such that $i_{m-1} = i$. We write $P \trianglelefteq^i_{\delta,\alpha} T$ to mean that there is a (δ, α)-occurrence of P at position i in T (in fact, when the bounds δ and α are well understood from the context, one can simply write $P \trianglelefteq^i T$).

The δ-approximate string matching problem with α-bounded gaps has been first formally defined in [26], where the δ-Bounded-Gaps algorithm has been proposed (see also [25, 17]). The δ-Bounded-Gaps algorithm, whose time and space complexity is $\mathcal{O}(nm)$, with n and m the lengths of the text T and of the pattern P respectively, is presented as an incremental procedure, based on the dynamic programming approach. Scanning the pattern P from left to right, the δ-Bounded-Gaps algorithm looks for the (δ, α)-occurrences of each prefix P_j of the pattern P in the whole text T, for

$0 \leq j < m$. Specifically, the δ-Bounded-Gaps algorithm proceeds by filling in a table D of dimensions $m \times n$ such that

$$D[j,i] = \max(\{k \geq 0 : i - \alpha \leq k \leq i \text{ and } P_j \trianglelefteq^k T\} \cup \{-1\})$$

for $0 \leq j < m$ and $0 \leq i < n$. Notice that $P_j \trianglelefteq^i T$ if and only if $D[j,i] = i$.

An algorithm, slightly more efficient than the δ-Bounded-Gaps, has been presented by the authors in [17], under the name (δ,α)-Sequential-Sampling. As in the case of the δ-Bounded-Gaps algorithm, the (δ,α)-Sequential-Sampling is also based on dynamic programming, but it follows a different computation ordering than the δ-Bounded-Gaps algorithm does; more precisely, it scans the text T from left to right and for each position i of T it looks for the (δ,α)-occurrences at position i of all prefixes of the pattern P. The (δ,α)-Sequential-Sampling algorithm has an $\mathcal{O}(nm)$ running time and requires $\mathcal{O}(m\alpha)$-space. A much more efficient variant of it is the (δ,α)-Tuned-Sequential-Sampling algorithm, which has an average case running time of $\mathcal{O}(n)$, in the case in which α is assumed constant (cf. [18]).

Another algorithm, named (δ,α)-Shift-And, has also been described in [18]. The (δ,α)-Shift-And algorithm is a very simple variant of a forward search algorithm presented in [58] for a pattern matching problem with gaps and character classes, particularly suited for applications to protein searching. It uses bit-parallelism to simulate the behavior of a nondeterministic finite automaton with ε-transitions. The automaton has $\ell = (\alpha+1)(m-1)+2$ states, and the simulation is carried out by representing it as a bit mask B of length $\ell-1$ (the initial state of the automaton need not be represented in the bit mask since it is always active during the computation). When $\ell < w$ (the computer word length), the entire bit mask B fits in a single computer word. In this case the (δ,α)-Shift-And algorithm becomes extremely fast in practice.

Other efficient algorithms for the (δ,α)-matching problem have been presented more recently in [36] and [37]. In particular, [36] presents two algorithms, called DA-bpdb and DA-mloga-bits. The first one inherits the basic idea from the dynamic programming algorithm δ-Bounded-Gaps presented in [25]. It uses bit-parallelism to compute an $m \times n$ bit-matrix $\mathcal{D}$ such that $(\mathcal{D})_{j,i} = 1$ if and only if $P_j \trianglelefteq^i T$, for $0 \leq j < m$ and $0 \leq i < n$. Basically, the algorithm DA-bpdb partitions each row of the matrix $\mathcal{D}$ as a sequence of $\lceil n/w \rceil$ consecutive bit masks, each of which represents a group of w bits on that row. Then, the computation of the j-th bit mask in row i is performed bit-parallely by using the $(j-1)$-st and the j-th bit masks of the $(i-1)$-st row. It turns out that DA-bpdb has an $\mathcal{O}(n\delta + \lceil n/w \rceil m)$ worst-case execution time, which becomes $\mathcal{O}(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$ on the average. The second algorithm presented in [36], namely DA-mloga-bits, is based on a compact representation, in the form of a systolic array, of the nondeterministic automaton used in the algorithm (δ,α)-Shift-And. The systolic array is composed of m building blocks, called *counters* in [36], one for each symbol of the pattern, and is represented as a bit mask of length $(m-1)(\lceil \log_2(\alpha+1) \rceil + 1) + 1$. Notice that this improves the representations used in [58, 18] in which $(\alpha+1)(m-1)+1$ bits are needed to represent the automaton. It turns out that the DA-mloga-bits algorithm has an $\mathcal{O}(n\lceil (m \log_2 \alpha)/w \rceil)$ worst-case searching time.

The algorithms presented in [37], called SDP-rows, SDP-columns, SDP-simple, and SDP-simple-compute-L_0, use different computation orderings, in combination

with sparse dynamic programming techniques, to implement the calculation of the table D above. Specifically, in the case of the SDP-rows algorithm, the computation is performed row-wise, whereas a column-wise computation is used by SDP-columns. The algorithm SDP-simple, which can be considered as a brute force variant of SDP-rows, performs very well in practice, especially for small values of δ and α; SDP-simple-compute-L_0 improves the average case running time of SDP-simple by using a Boyer-Moore-Horspool-like shifting strategy [41], suitably adapted to handle gaps. In particular, the latter two algorithms turn out to be among the most efficient ones, in terms of running time, in many practical cases, especially for small values of α, as shown in [37]. However, although these algorithms are very fast in practice, they require additional $\mathcal{O}(n)$-space, plus $\mathcal{O}(\sigma)$-space in the case of SDP-simple-compute-L_0.

More recently, in [19], the authors presented four new efficient variants of the algorithm (δ, α)-Sequential-Sampling, all based on bit-parallelism. In particular, one of these variants, the (δ, α)-Tuned-Sequential-Sampling-HBP algorithm, is extremely efficient in most practical cases and outperforms both algorithms SDP-simple and SDP-simple-compute-L_0. The variant (δ, α)-Sequential-Sampling-BP$^+$ turns out to be faster than existing algorithms (e.g., (δ, α)-Shift-And) in the case of short patterns and very small values of α.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. 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*, number 1725 in Lecture Notes in Computer Science, pages 291–306, Milovy, Czech Republic, 1999. Springer-Verlag, Berlin.
3. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. In *IEEE Symposium on Foundations of Computer Science*, pages 144–153, 1997.
4. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Journal of Algorithms*, 37(2):247–266, 2000.
5. A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. *Inf. Comput.*, 181(1):57–74, 2003.
6. 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.
7. A. Amir, M. Lewenstein, and Ely Porat. Approximate swapped matching. *Inf. Process. Lett.*, 83(1):33–39, 2002.
8. P. Antoniou, C.S. Iliopoulos, I. Jayasekera, and M.S. Rahman. Implementation of a swap matching algorithm using a graph theoretic model. In *Bioinformatics Research and Development, Second International Conference, BIRD 2008*, volume 13 of *Communications in Computer and Information Science*, pages 446–455. Springer, 2008.
9. Jörg Arndt. *Matters Computational*. Springer, 2011. `http://www.jjj.de/fxt/`.
10. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
11. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

12. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel. Linear size finite automata for the set of all subwords of a word: an outline of results. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 21:12–20, 1983.

13. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.

14. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

15. M. Campanelli, D. Cantone, and S. Faro. A new algorithm for efficient pattern matching with swaps. In *IWOCA 2009: 20th International Workshop on Combinatorial Algorithms*, Lecture Notes in Computer Science. Springer, 2009.

16. M. Campanelli, D. Cantone, S. Faro, and E. Giaquinta. Pattern matching with swaps in practice. *International Journal of Foundation of Computer Science*, 23(2):323–342, 2012.

17. D. Cantone, S. Cristofaro, and S. Faro. An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In S. E. Nikoletseas, editor, *Proceedings of 4-th International Workshop on Experimental and Efficient Algorithms (WEA 2005)*, volume 3503 of *Lecture Notes in Computer Science*, pages 428–439. Springer-Verlag, 2005.

18. D. Cantone, S. Cristofaro, and S. Faro. On tuning the (δ, α)-sequential-sampling algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In S. D. Reiss and G. A. Wiggins, editors, *Proceedings of 6-th International Conference on Music Information Retrieval (ISMIR 2005)*, pages 454–459, 2005.

19. D. Cantone, S. Cristofaro, and S. Faro. New efficient bit-parallel algorithms for the (δ, α)-matching problem with applications in music information retrieval. *International Journal of Foundation of Computer Science*, 20(6):1087–1108, 2009.

20. D. Cantone and S. Faro. A space efficient bit-parallel algorithm for the multiple string matching problem. *Int. J. Found. Comput. Sci.*, 17(6):1235–1252, 2006.

21. D. Cantone and S. Faro. Pattern matching with swaps for short patterns in linear time. In *SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2009.

22. D. Cantone, S. Faro, and E. Giaquinta. Bit-(parallelism)^2: Getting to the next level of parallelism. In Paolo Boldi and Luisa Gargano, editors, *Fun with Algorithms*, volume 6099 of *Lecture Notes in Computer Science*, pages 166–177. Springer-Verlag, Berlin, 2010.

23. D. Cantone, S. Faro, and E. Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Information and Computation*, 213:3–12, 2012.

24. C. Charras and T. Lecroq. *Handbook of exact string matching algorithms.* King's College Publications, 2004.

25. M. Crochemore, C. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps. *Nordic J. of Computing*, 9(1):54–65, 2002.

26. M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and W. Rytter. Finding motifs with gaps. In Don Byrd and J. Stephen Downie, editors, *Proceedings of International Symposium on Music Information Retrieval: Music IR 2000*, Amherst, MA, 2000. University of Massachusetts at Amherst.

27. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

28. F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, March 1964.

29. B. Dömölki. A universal compiler system based on production rules. *BIT Numerical Mathematics*, 8:262–275, 1968.

30. B. Durian, J. Holub, H. Peltola, and J. Tarhio. Tuning BNDM with q-grams. In I. Finocchi and J. Hershberger, editors, *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2009*, pages 29–37, New York, New York, USA, 2009. SIAM.

31. B. Durian, H. Peltola, L. Salmela, and J. Tarhio. Bit-parallel search algorithms for long patterns. In *SEA*, LNCS 6049, pages 129–140, 2010.

32. S. Faro and T. Lecroq. Efficient variants of the Backward-Oracle-Matching algorithm. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2008*, pages 146–160, Czech Technical University in Prague, Czech Republic, 2008.

33. S. Faro and T. Lecroq. The exact string matching problem: a comprehensive experimental evaluation. Report arXiv:1012.2547, 2010.

34. S. Faro and T. Lecroq. The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys*, 45(2), 2013. to appear.

35. K. Fredriksson and S. Grabowski. Practical and optimal string matching. In M. P. Consens and G. Navarro, editors, *SPIRE*, volume 3772 of *Lecture Notes in Computer Science*, pages 376–387. Springer-Verlag, Berlin, 2005.

36. K. Fredriksson and Sz. Grabowski. Efficient bit-parallel algorithms for (δ, α)-matching. In *Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA'06)*, LNCS 4007, pages 170–181. Springer–Verlag, 2006.

37. K. Fredriksson and Sz. Grabowski. Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. *Information Retrieval*, March 2008. to appear (currently available only online).

38. L. He, B. Fang, and J. Sui. The wide window string matching algorithm. *Theor. Comput. Sci.*, 332(1-3):391–404, 2005.

39. J. Holub and B. Durian. Talk: Fast variants of bit parallel approach to suffix automata. In *The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation, http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf*, 2005.

40. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2001.

41. R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.

42. A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.

43. H. Hyyrö. Tighter packed bit-parallel nfa for approximate string matching. In *CIAA*, pages 287–289, 2006.

44. H. Hyyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate string matching. In *WEA*, pages 285–298, 2004.

45. Heikki Hyyrö. A bit-vector algorithm for computing levenshtein and damerau edit distances. *Nord. J. Comput.*, 10(1):29–39, 2003.

46. C. S. Iliopoulos and M. S. Rahman. A new model to solve the swap matching problem and efficient algorithms for short patterns. In *SOFSEM 2008*, volume 4910 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 2008.

47. P. Kalsi, H. Peltola, and J. Tarhio. Comparison of exact string matching algorithms for biological sequences. In M. Elloumi, J. Küng, M. Linial, R. F. Murphy, K. Schneider, and C. Toma, editors, *Proceedings of the Second International Conference on Bioinformatics Research and Development, BIRD'08*, volume 13 of *Communications in Computer and Information Science*, pages 417–426, Vienna, Austria, 2008. Springer-Verlag, Berlin.

48. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

49. M. Oğuzhan Külekci. A method to overcome computer word size limitation in bit-parallel pattern matching. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation, ISAAC 2008*, volume 5369 of *Lecture Notes in Computer Science*, pages 496–506, Gold Coast, Australia, 2008. Springer-Verlag, Berlin.

50. VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

51. J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.

52. S. Muthukrishnan. New results and open problems related to non-standard stringology. In *Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95*, volume 937 of *Lecture Notes in Computer Science*, pages 298–317. Springer, 1995.

53. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

54. G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

55. G. Navarro. NR-grep: a fast and flexible pattern-matching tool. *Softw. Pract. Exp.*, 31(13):1265–1312, 2001.

56. G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *Theor. Comput. Sci.*, 321(2-3):283–290, 2004.

57. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *J. Exp. Algorithmics*, 5:4, 2000.

58. G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *RECOMB '01: Proceedings of the fifth annual international conference onComputational biology*, pages 231–240, New York, NY, USA, 2001. ACM.

59. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

60. G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2005.

61. H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In M. A. Nascimento, E. Silva de Moura, and A. L. Oliveira, editors, *Proceedings of the 10th International Symposium on String Processing and Information Retrieval SPIRE'03*, volume 2857 of *Lecture Notes in Computer Science*, pages 80–94, Manaus, Brazil, 2003. Springer-Verlag, Berlin.

62. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1(10):359–373, 1980.

63. D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

64. S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

65. S. Wu and U. Manber. Fast text searching: allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

66. A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.

67. G. Zhang, E. Zhu, L. Mao, and M. Yin. A bit-parallel exact string matching algorithm for small alphabet. In X. Deng, J. E. Hopcroft, and J. Xue, editors, *Proceedings of the Third International Workshop on Frontiers in Algorithmics, FAW 2009, Hefei, China*, volume 5598 of *Lecture Notes in Computer Science*, pages 336–345. Springer-Verlag, Berlin, 2009.