

# The Exact Online String Matching Problem: a Review of the Most Recent Results

SIMONE FARO, Università di Catania  
THIERRY LECROQ, Université de Rouen

This article addresses the *online exact string matching problem* which consists in finding *all* occurrences of a given pattern  $p$  in a text  $t$ . 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, data compression, computational biology and chemistry.

In the last decade more than 50 new algorithms have been proposed for the problem, which add up to a wide set of (almost 40) algorithms presented before 2000. In this article we review the string matching algorithms presented in the last decade and present experimental results in order to bring order among the dozens of articles published in this area.

Categories and Subject Descriptors: I.7.0 [Computing Methodologies]: Document and Text Processing—General; H.3.3 [Information Systems]: Information Search and Retrieval—Search process

General Terms: Algorithms, Design, Experimentation

Additional Key Words and Phrases: string matching, experimental algorithms, text processing, automaton

## 1. INTRODUCTION

Given a text  $t$  of length  $n$  and a pattern  $p$  of length  $m$  over some alphabet  $\Sigma$  of size  $\sigma$ , the *string matching problem* consists in finding *all* occurrences of the pattern  $p$  in the text  $t$ . 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 of linguistic 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.

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

---

Author's address: Simone Faro, Università di Catania, Dipartimento di Matematica e Informatica, Viale Andrea Doria 6, I-95125 Catania, Italy, Tel.: +39-095-7383090, Fax: +39-095-330094, Email: faro@dmf.unict.it  
Thierry Lecroq, Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France, Tel.: +33-235-146581, Fax: +33-235-147140, Email: thierry.lecroq@univ-rouen.fr

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© YYYY ACM 0360-0300/YYYY/01-ARTA \$10.00

DOI 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

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 five classes: algorithms which solve the problem by making use only of comparisons between characters, algorithms which make use of deterministic automata, algorithms which simulate nondeterministic automata, constant-space algorithms and real-time algorithms.

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 [Morris and Pratt 1970] later improved by Knuth, Morris and Pratt [Knuth et al. 1977]. The reader can refer to Section 7 of [Knuth et al. 1977] for historical remarks. Linear algorithms have been developed also based on bit-parallelism [Baeza-Yates and Gonnet 1992; Wu and Manber 1992]. An average lower bound in  $\mathcal{O}(n \log m/m)$  (with equiprobability and independence of letters) has been proved by Yao in [Yao 1979].

Many string matching algorithms have been also developed to obtain sublinear performance in practice (see [Charras and Lecroq 2004] for the most recent survey). Among them the Boyer-Moore algorithm [Boyer and Moore 1977] 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 [Horspool 1980] and Quick-Search [Sunday 1990] 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 [Blumer et al. 1983; Allauzen et al. 1999], data structures which identify all factors of a word. Among them the Backward-Oracle-Matching algorithm [Allauzen et al. 1999] is one of the most efficient algorithms especially for long patterns.

Another algorithm based on the simulation of the non-deterministic factor automaton, and called Backward Nondeterministic Dawg Match algorithm [Navarro and Raffinot 1998] (BNDM), is very efficient for short patterns.

Two of the most important sub-domain of string matching are *constant-space string matching*, which aims to solutions which use only constant extra space, and *real-time string matching* where the time difference between the processing of two consecutive text characters must be bounded by a constant.

In the case of constant-space string matching the two main algorithms are the Galil and Seiferas algorithm [Galil and Seiferas 1983] and the Two Way algorithm by Crochemore and Perrin [Crochemore and Perrin 1991]. The main technique for performing string matching in real time is due to Galil [Galil 1981].

In this article we review the string matching algorithms which have been proposed in the last decade (2000-2010) and present experimental results in order to bring order among the dozens of articles published in recent years. For simplicity we divide the discussion into four main sections: algorithms based on characters comparison, automata based algorithms, algorithms based on bit-parallelism and constant-space algorithms. (no real-time string matching has been designed in the last decade.) Fig. 1 shows a comprehensive list of the algorithms presented in the last decade.

Specifically in Section 2 we introduce basic definitions and the terminology used along the paper. In Section 3 we survey the most classical string matching algorithms designed before year 2000. In Section 4 we survey the most recent algorithms designed after 2000. Experimental data obtained by running under various conditions all the algorithms reviewed are presented and compared in Section 5.

Algorithms based on characters comparison		
Ahmed-Kaykobad-Chowdhury	Ahmed et al. [2003]	Sec. 4.1.1
Fast-Search	Cantone and Faro [2003a]	Sec. 4.1.1
Forward-Fast-Search	Cantone and Faro [2005]	Sec. 4.1.1
Backward-Fast-Search, Fast-Boyer-Moore	Cantone and Faro [2005]	Sec. 4.1.1
Sheik-Sumit-Anindya-Balakrishnan-Sekar	Sheik et al. [2004]	Sec. 4.1.1
Thathoo-Virmani-Sai-Balakrishnan-Sekar	Thathoo et al. [2006]	Sec. 4.1.1
Boyer-Moore-Horspool using Probabilities	Nebel [2006]	Sec. 4.1.1
Franek-Jennings-Smyth	Franek et al. [2007]	Sec. 4.1.1
2-Block Boyer-Moore	Sustik and Moore [2007]	Sec. 4.1.1
Two-Sliding-Window	Hudaib et al. [2008]	Sec. 4.1.2
Wu-Manber for Single Pattern Matching	Lecroq [2007]	Sec. 4.1.3
Boyer-Moore-Horspool with $q$ -grams	Kalsi et al. [2008]	Sec. 4.1.3
Genomic Rapid Algo for String Pm	Deusdado and Carvalho [2009]	Sec. 4.1.3
SSEF	Külekci [2009]	Sec. 4.1.4
Algorithms based on automata		
Extended Backward Oracle Matching	Faro and Lecroq [2008]	Sec. 4.2.1
Simplified Extended Backward Oracle Matching	Fan et al. [2009]	Sec. 4.2.1
Forward Backward Oracle Matching	Faro and Lecroq [2008]	Sec. 4.2.1
Simplified Forward Backward Oracle Matching	Fan et al. [2009]	Sec. 4.2.1
Double Forward DAWG Matching	Allauzen and Raffinot [2000]	Sec. 4.2.2
Wide Window	He et al. [2005]	Sec. 4.2.3
Linear DAWG Matching	He et al. [2005]	Sec. 4.2.3
Improved Linear DAWG Matching	Liu et al. [2006]	Sec. 4.2.3
Improved Linear DAWG Matching 2	Liu et al. [2006]	Sec. 4.2.3
Algorithms based on bit-parallelism		
Simplified BNDM	Peltola and Tarhio [2003]	Sec. 4.3.1
BNDM with loop-unrolling	Holub and Durian [2005]	Sec. 4.3.1
Simplified BNDM with loop-unrolling	Holub and Durian [2005]	Sec. 4.3.1
BNDM with Horspool Shift	Holub and Durian [2005]	Sec. 4.3.1
Horspool with BNDM test	Holub and Durian [2005]	Sec. 4.3.1
Forward SBNDM	Faro and Lecroq [2008]	Sec. 4.3.1
Forward Non-deterministic DAWG Matching	Holub and Durian [2005]	Sec. 4.3.2
Two-Way Non-deterministic DAWG Matching	Peltola and Tarhio [2003]	Sec. 4.3.2
Bit parallel Wide Window	He et al. [2005]	Sec. 4.3.3
Bit-Parallel <sup>2</sup> Wide-Window	Cantone et al. [2010b]	Sec. 4.3.3
Bit-Parallel Wide-Window <sup>2</sup>	Cantone et al. [2010b]	Sec. 4.3.3
Small Alphabet Bit-Parallel	Zhang et al. [2009]	Sec. 4.3.3
Shift Vector Matching	Peltola and Tarhio [2003]	Sec. 4.3.4
Fast Average Optimal Shift-Or	Fredriksson and Grabowski [2005]	Sec. 4.3.5
Average Optimal Shift-Or	Fredriksson and Grabowski [2005]	Sec. 4.3.5
Bit-Parallel Length Invariant Matcher	Külekci [2008]	Sec. 4.3.5
BNDM for Long patterns	Navarro and Raffinot [2000]	Sec. 4.3.5
Long patterns BNDM	Peltola and Tarhio [2003]	Sec. 4.3.5
Factorized BNDM	Cantone et al. [2010a]	Sec. 4.3.5
Factorized Shift-And	Cantone et al. [2010a]	Sec. 4.3.5
BNDM with $q$ -grams	Durian et al. [2009]	Sec. 4.3.6
Simplified BNDM with $q$ -grams	Durian et al. [2009]	Sec. 4.3.6
Shift-Or with $q$ -grams	Durian et al. [2009]	Sec. 4.3.6
Constant-space string matching algorithms		
Tailed-Substring	Cantone and Faro [2004]	Sec. 4.4.1

Fig. 1. The list of the string matching algorithms presented in the last eleven years (2000-2010).

## 2. BASIC DEFINITIONS AND TERMINOLOGY

We consider a text  $t$  of length  $n$  and a pattern  $p$  of length  $m$  over some alphabet  $\Sigma$  of size  $\sigma$ . A string  $p$  of length  $m$  is represented as a finite array  $p[0..m-1]$ , with  $m \geq 0$ . In particular, for  $m = 0$  we obtain the empty string, also denoted by  $\varepsilon$ . By  $p[i]$  we denote the  $(i+1)$ -st character of  $p$ , for  $0 \leq i < m$ . Likewise, by  $p[i..j]$  we denote the substring of  $p$  contained between the  $(i+1)$ -st and the  $(j+1)$ -st characters of  $p$ , for  $0 \leq i \leq j < m$ . Moreover, for any  $i, j \in \mathbb{Z}$ , we put  $p[i..j] = \varepsilon$  if  $i > j$  and  $p[i..j] = p[\max(i, 0), \min(j, m-1)]$  if  $i \leq j$ . A substring of a given string  $p$  is also called a *factor* of  $p$ . We denote by  $\text{Fact}(p)$  the set of all the factors of the string  $p$ . A substring of the form  $p[0..i]$  is called a *prefix* of  $p$  and a substring of the form  $p[i..m-1]$  is called a *suffix* of  $p$  for  $0 \leq i \leq m-1$ . We denote by  $\text{Suff}(p)$  the set of all the suffixes of the string  $p$ . For simplicity in most cases we write  $p_j$  to indicate the prefix of  $p$  of length  $j+1$ , i.e.  $p_j = p[0..j]$ . For any two strings  $u$  and  $w$ , we write  $w \sqsupset u$  to indicate that  $w$  is a suffix of  $u$ . Similarly, we write  $w \sqsubset u$  to indicate that  $w$  is a prefix of  $u$ . A substring  $u$  of  $p$  is a *border* if  $u$  is both a prefix and a suffix of  $p$ . A integer  $k$  is a *period* of a word  $w$  if for  $i, 0 \leq i < m-k$ ,  $w[i] = w[i+k]$ . The smallest period of  $w$  is called *the period* of  $w$ , it is denoted by  $\text{per}(w)$ . A word  $w$  is *basic* if it cannot be written as a power of another word: there exist no word  $z$  and no integer  $k$  such that  $w = z^k$ . The *reverse* of a string  $p[0..m-1]$  is the string, denoted by  $\bar{p}$ , built by the concatenation of its letters from the last to the first:  $p[m-1]p[m-2]\dots p[1]p[0]$ .

For any two strings  $u$  and  $w$  of  $\Sigma^*$ , we define the function  $\text{endpos}_w(u)$  which returns the set of all positions  $j \geq |u| - 1$  such that  $w[j - |u| + 1..j] = u$ . Moreover we define the relation  $\sim_p$  by  $u \sim_p v$  if and only if  $\text{endpos}_p(u) = \text{endpos}_p(v)$ .

A Finite State Automaton is a tuple  $A = \{Q, q_0, F, \Sigma, \delta\}$ , where  $Q$  is the set of states of the automaton,  $q_0 \in Q$  is the initial state,  $F \subseteq Q$  is the set of accepting states,  $\Sigma$  is the alphabet of characters labeling transitions and  $\delta : (Q \times \Sigma) \rightarrow Q$  is the *transition function*. If  $\delta(q, c)$  is not defined for a state  $q \in Q$  and a character  $c \in \Sigma$  we say that  $\delta(q, c)$  is an undefined transition and write  $\delta(q, c) = \perp$ .

The transition function  $\delta$ , naturally induces *final state function*,  $\delta^* : \Sigma^* \rightarrow Q$ , defined recursively by  $\delta^*(w) = \delta(\delta^*(w'), c)$ , for each  $w = w'c$ , with  $w' \in \Sigma^*$ ,  $c \in \Sigma$ .

Let  $t$  be a text of length  $n$  and let  $p$  be a pattern of length  $m$ . When the character  $p[0]$  is aligned with the character  $t[s]$  of the text, so that the character  $p[i]$  is aligned with the character  $t[s+i]$ , for  $i = 0, \dots, m-1$ , we say that the pattern  $p$  has *shift*  $s$  in  $t$ . In this case the substring  $t[s..s+m-1]$  is called the *current window* of the text. If  $t[s..s+m-1] = p$ , we say that the shift  $s$  is *valid*. Thus the string matching problem can be rephrased as the problem of finding *all* valid shifts of a pattern  $p$  relative to a text  $t$ .

Most string matching algorithms generally work as follows. They scan the text with the help of 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.,  $t[s..s+m-1]$ .

### 3. ALGORITHMS DESIGNED BEFORE 2000

In this section we survey the most classical string matching algorithms that have been designed before year 2000. In particular we present in Section 3.1 the algorithms based on comparisons and in Section 3.2 the algorithms based on automata. Then in Section 3.3 the algorithms based on bit-parallelism are presented. Finally, constant-space algorithms and real-time algorithms are presented in Section 3.4 and in Section 3.5, respectively.

#### 3.1. Comparison Based Algorithms

Most of the comparison based algorithms presented in the last ten years are obtained by improving or combining the ideas of previously published algorithms. In the following we briefly review the state of the art until 2000 and the main ideas and the algorithms to which the new solutions refer.

One of the first algorithm to solve the string matching problem in linear time is due to Knuth, Morris and Pratt [Knuth et al. 1977]. The search is done by scanning the current window of the text from left to right character by character and, for each text position  $j$ , remembering the longest prefix of the pattern which is also a suffix of  $t[s..j]$ . Specifically, if a mismatch occurs with the character of position  $i$  in the pattern, the Knuth-Morris-Pratt algorithm advances the shift  $s$  of exactly  $i - \pi_p(i)$  characters to the right, where  $\pi_p(0) = -1$  and

$$\pi_p(i) = \max \left( \{0 \leq k < i \mid p[0..k-1] = p[i-k..i-1] \text{ and } p[k] \neq p[i]\} \cup \{-1\} \right) \quad (1)$$

is the length of the longest border of  $p[0..i-1]$  followed by a character different from  $p[i]$  for  $0 \leq i \leq m-1$ . When an occurrence of the pattern is found, which means that  $i = m$  then the window is shifted using  $\pi_p(m) = \max\{0 \leq k < m \mid p[0..k-1] = p[i-k..m-1]\}$ .

Then the next window alignment starts at position  $s' = s + i - \pi_p(i)$  in the text and the reading resumes from character  $t[s' + \pi_p(i)]$ .

The Boyer-Moore algorithm [Boyer and Moore 1977] is the progenitor of several algorithmic variants which aim at computing close to optimal shift increments very efficiently. Specifically, the Boyer-Moore algorithm checks whether  $s$  is a valid shift by scanning the pattern  $p$  from right to left and, at the end of the matching phase, computes the shift increment as the maximum value suggested by the *good suffix rule* and the *bad character rule* below, using the functions  $gs_p$  and  $bc_p$  respectively, provided that both of them are applicable.

If the first mismatch occurs at position  $i$  of the pattern  $p$ , the good suffix rule suggests to align the substring  $t[s+i+1..s+m-1] = p[i+1..m-1]$  with its rightmost occurrence in  $p$  preceded by a character different from  $p[i]$ . If such an occurrence does not exist, the good suffix rule suggests a shift increment which allows to match the longest suffix of  $t[s+i+1..s+m-1]$  with a prefix of  $p$ . Specifically, if the first mismatch occurs at position  $i$  of the pattern  $p$ , the *good suffix rule* states that the shift can safely be incremented by  $gs_p(i+1)$  positions, where

$$gs_p(j) = \min \{ 0 < k \leq m \mid p[j-k..m-k-1] \supset p \text{ and } (k \leq j-1 \rightarrow p[j-1] \neq p[j-1-k]) \} , \quad (2)$$

for  $j = 0, 1, \dots, m$ . (The situation in which an occurrence of the pattern  $p$  is found can be regarded as a mismatch at position  $-1$ .)

The *bad character rule* states that if  $c = t[s+i] \neq p[i]$  is the first mismatching character, while scanning  $p$  and  $t$  from right to left with shift  $s$ , then  $p$  can safely be shifted in such a way that its rightmost occurrence of  $c$ , if present, is aligned with



position  $(s + i)$  in  $t$ . In the case in which  $c$  does not occur in  $p$ , then  $p$  can safely be shifted just past position  $(s + i)$  in  $t$ . More formally, the shift increment suggested by the bad character rule is given by the expression  $(i - bc_p(t[s + i]))$ , where

$$bc_p(c) = \max(\{0 \leq k < m \mid p[k] = c\} \cup \{-1\}) , \quad (3)$$

for  $c \in \Sigma$ , and where we recall that  $\Sigma$  is the alphabet of the pattern  $p$  and text  $t$ . Notice that there are situations in which the shift increment given by the bad character rule can be negative.

Horspool in [Horspool 1980] suggested a simplification of the original Boyer-Moore algorithm, defining a new variant which, though quadratic in the worst case, performs better in practical cases. He just dropped the good suffix rule and proposed to compute the shift advancement in such a way that the rightmost character  $t[s + m - 1]$  is aligned with its rightmost occurrence in  $p[0..m - 2]$ , if present; otherwise the pattern is advanced just past the window. This corresponds to advance the shift by  $hbc_p(t[s + m - 1])$  positions where, for all  $c \in \Sigma$

$$hbc_p(c) = \min(\{1 \leq k < m \mid p[m - 1 - k] = c\} \cup \{m\}) . \quad (4)$$

The resulting algorithm performs well in practice and can be immediately translated into programming code.

The Quick-Search algorithm [Sunday 1990], presented by Sunday in 1990, is a very simple variation of the Horspool algorithm. After each attempt the shift is computed according to the character which immediately follows the current window of the text, i.e. the character  $t[s + m]$  (in many subsequent articles referred as the *forward character*). This corresponds to advance the shift by  $qbc_p(t[s + m])$  positions, where

$$qbc_p(c) = \min(\{1 \leq k \leq m \mid p[m - k] = c\} \cup \{m + 1\}) . \quad (5)$$

for all  $c \in \Sigma$ .

Finally in the Berry-Ravindran algorithm [Berry and Ravindran 1999] (BR for short), which is an improvement of the Quick-Search algorithm, after each attempt the pattern is advanced so that the substring of the text  $t[s + m..s + m + 1]$  is aligned with its rightmost occurrence in  $p$ . This corresponds to advance the shift by  $brbc_p(t[s + m], t[s + m + 1])$  positions where, for all  $c \in \Sigma$

$$brbc_p(c_1, c_2) = \min \left( \{1 \leq k < m \mid p[m - k - 1] = c_1 \text{ and } p[m - k] = c_2\} \cup \{k \mid k = m \text{ and } p[0] = c_2\} \cup \{m + 1\} \right) . \quad (6)$$

More effective implementations of the algorithms based on a bad character rule can be obtained by making use of a *fast loop* (or *character unrolling cycle*). This technique has been introduced first in the Tuned-Boyer-Moore algorithm [Hume and Sunday 1991], and then largely used in almost all algorithms based on a bad character rule. It simply consists in applying rounds of several blind shifts (based on the bad character rule) while needed. The fact that the blind shifts require no checks is at the heart of the very good practical behavior of such algorithms. Moreover, in order to compute the last shifts correctly it is necessary to add a copy of  $p$  at the end of the text  $t$ , as a sentinel.

### 3.2. Algorithms based on Deterministic Automata

Automata play a very important role in the design of efficient string matching algorithms. The first linear algorithm based on deterministic automata is the Automaton Matcher [Aho et al. 1974]. Such algorithm makes use of the string matching automaton (SMA for short)  $\mathcal{D}(p)$  for a given pattern  $p$  of length  $m$ . Specifically the SMA  $\mathcal{D}(p)$  is a quintuple  $\mathcal{D}(p) = \{Q, q_0, F, \Sigma, \delta\}$  where  $Q = \{q_0, q_1, \dots, q_m\}$  is the set of states;  $q_0$

is the initial state;  $F = \{q_m\}$  is the set of accepting states and  $\delta : Q \times \Sigma \rightarrow Q$  is the transition function defined as

$$\delta(q_i, c) = \begin{cases} q_{i+1} & \text{if } c = p[i+1] \\ q_{|k|}, \text{ where } k \text{ is the longest border of } p[0..i-1]_c & \text{otherwise} \end{cases}$$

for  $0 \leq i \leq m$  and  $c \in \Sigma$ . The language recognized by  $\mathcal{D}(p)$  is  $\mathcal{L}(\mathcal{D}(p)) = \Sigma^* p$ .

The Automaton Matcher simply scans the text character by character, from left to right, performing transitions on the automaton  $\mathcal{D}(p)$ . If the final state is reached after the character  $t[j]$  is read, then a match is reported at position  $(j - m + 1)$ . Simon [Simon 1993] noticed that this automaton only have  $m - 1$  backward significant transitions not leading to the initial state. Thus it is possible to represent it in linear size independently from the underlying alphabet. The resulting string matching algorithm has been studied by C. Hancart [Hancart 1993].

Automata based solutions have been also developed to design algorithms which have optimal sublinear performance on average. This is done by using factor automata [Blumer et al. 1983; Crochemore 1985; Blumer et al. 1985] and factor oracles [Allauzen et al. 1999].

The factor automaton of a pattern  $p$ ,  $Dawg(p)$ , is also called the factor DAWG of  $p$  (for Directed Acyclic Word Graph). Such an automaton recognizes all the factors of  $p$ . Formally the language recognized by  $Dawg(p)$  is the set  $Fact(p)$ .

In some cases we add to the previous definition of the automaton a failure function,  $Fail : Fact(p) \setminus \{\varepsilon\} \rightarrow Fact(p)$ , defined, for every nonempty factor  $v$  of  $p$ , as the longest  $u \in Suff(v)$  such that  $u \approx_p v$ .

The factor oracle of a pattern  $p$ ,  $Oracle(p)$ , is a very compact automaton which recognizes at least all the factors of  $p$  and slightly more other words. Formally  $Oracle(p)$  is an automaton  $\{Q, q_m, Q, \Sigma, \delta\}$  such that  $Q$  contains exactly  $m + 1$  states, say  $Q = \{q_0, q_1, q_2, q_3, \dots, q_m\}$ ,  $q_m$  is the initial state, all states are final and the language accepted by  $Oracle(p)$  is such that  $\mathcal{L}(Dawg(p)) \subseteq \mathcal{L}(Oracle(p))$ .

Despite the fact that the factor oracle is able to recognize words that are not factors of the pattern [Mancheron and Moan 2005], it can be used to search for a pattern in a text since the only factor of  $p$  of length greater or equal to  $m$  which is recognized by the oracle is the pattern itself. The computation of the factor automaton and factor oracle is linear in time and space in the length of the pattern.

The factor automaton is used in [Crochemore and Rytter 1994; Crochemore 1986] for a forward search of a pattern in a text, by reading the letters of the text one after the other. The algorithm is called Forward-DAWG-Matching (FDM for short). For each position  $j$  in the text, we compute the length of the longest factor of  $p$  that is a suffix of the prefix  $t_j$ . We first build the factor automaton  $Dawg(p)$  and initialize the current length  $\ell$  to 0. We then read the characters one by one, updating the length  $\ell$  for each letter. Assume that the algorithm reads the text until  $j$  and that it arrives in a state  $q$  with some value  $\ell$ . It reads character  $t[j + 1]$  and updates  $\ell$  and  $q$  in the following way: if there is a transition from  $q$  by  $t[j + 1]$  to a state  $f$ , then the current state becomes  $f$  and  $\ell$  is incremented by 1; If not, we go down the failure path of  $q$  until we found a state which has an existing transition by  $t[j + 1]$ . Two cases may occur: If there is such a state  $d$  that has a transition to a state  $r$  by  $t[j + 1]$ , then the current state becomes  $r$  and  $\ell$  is updated to length  $d + 1$ ; if not, the current state is set to the initial state of  $Dawg(p)$  with  $\ell$  set to 0.

The factor automaton and factor oracle are used respectively in [Crochemore et al. 1994; Crochemore and Rytter 1994] and in [Allauzen et al. 1999] to get optimal pattern matching algorithms on the average. The algorithm which makes use of the factor automaton of the reverse pattern is called Backward-DAWG-Matching algorithm

(BDM for short) while the algorithm using the factor oracle is called Backward-Oracle-Matching (BOM for short). Such algorithms move a window of size  $m$  on the text. For each new position of the window, the automaton of the reverse of  $p$  is used to search for a factor of  $p$  from the right to the left of the window. The basic idea of the BDM and BOM algorithms is that if the backward search failed on a letter  $c$  after the reading of a word  $u$  then  $cu$  is not a factor of  $p$  and moving the beginning of the window just after  $c$  is secure. If a factor of length  $m$  is recognized then an occurrence of the pattern was found. The Reverse Factor [Crochemore et al. 1994] and the Backward Suffix Oracle Matching [Allauzen et al. 1999] algorithms work as the BDM and BOM algorithm, respectively, but computes the shifts according to the longest prefix of the pattern that is a suffix of the window at each attempt.

### 3.3. Bit-Parallel Algorithms

*Bit-parallelism* is a technique firstly introduced in [Dömölki 1968], and later revisited in [Baeza-Yates and Gonnet 1992; Wu and Manber 1992]. It 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  $\omega$ , where  $\omega$  is the number of bits in the computer word. Bit-parallelism is particularly suitable for the efficient simulation of non-deterministic automata.

The algorithms reviewed in this section 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;
- “&” represents the bitwise operation AND;
- “^” represents the bitwise operation XOR;
- “~” represents the one’s complement;
- “<<” represents the bitwise left shift; and
- “>>” represents the bitwise right shift.

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;
- “bsf” represents the bit scan forward operation; and
- “popcount” represents population count operation.

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 preceding 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 [Arndt 2009] for the detailed implementation of the operations listed above).

The first solutions based on bit parallelism were the Shift-And [Wu and Manber 1992] (SA for short), Shift-Or [Baeza-Yates and Gonnet 1992] (SO for short) and BNDM [Navarro and Raffinot 1998] algorithms.

The Shift-And [Wu and Manber 1992] 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  $m$ , early described in its deterministic form in Section 3.2.



More precisely, given a string  $p \in \Sigma^m$ , the nondeterministic automaton for the language  $\Sigma^*p$  is the automaton  $NSMA(p) = (Q, \Sigma, \delta, q_0, F)$  which recognizes all words in  $\Sigma^*$  ending with an occurrence of  $p$ , where  $Q = \{q_0, q_1, \dots, q_m\}$ ,  $q_0$  is the initial state,  $F = \{q_m\}$  is the set of final states and the transition function  $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$  is defined by:

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

The valid configurations  $\delta^*(q_0, x)$  reachable by the automata  $NSMA(p)$  on input  $x \in \Sigma^*$  are defined recursively as follows:

$$\delta^*(q_0, x) =_{\text{Def}} \begin{cases} \{q\} & \text{if } x = \varepsilon, \\ \bigcup_{q' \in \delta(q_0, x')} \delta^*(q', c) & \text{if } x = x'c, \text{ for some } c \in \Sigma \text{ and } x' \in \Sigma^*. \end{cases}$$

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  $i = 0, \dots, m-1$ . For a configuration  $D$  of the NFA, a transition on character  $c$  can then be implemented by the bitwise operations

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

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  $\Sigma$  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 a match at position  $j - m + 1$  is reported.

The Shift-Or algorithm [Baeza-Yates and Gonnet 1992] 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 non-deterministic factor automaton ( $NDawg(p)$ ) of a given string  $p$  is an NFA with  $\varepsilon$ -transitions that recognizes the language  $Fact(p)$ . More precisely, for a string  $p \in \Sigma^m$ ,  $NDawg(p) = (Q, \Sigma, \delta, I, F)$ , where  $Q = \{I, q_0, q_1, \dots, q_m\}$ ,  $I$  is the initial state,  $F = \{q_m\}$  is the set of final states and the transition function  $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$

is defined by:

$$\delta(q, c) =_{\text{Def}} \begin{cases} \{q_{i+1}\} & \text{if } q = q_i \text{ and } c = p[i] \ (0 \leq i < m) \\ \{q_0, q_1, \dots, q_m\} & \text{if } q = I \text{ and } c = \varepsilon \\ \emptyset & \text{otherwise.} \end{cases}$$

The BNBM 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  $i = 0, 1, \dots, m-1$ , and  $D$  is initialized to  $1^m$ , since after the  $\varepsilon$ -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 BNBM 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 BNBM algorithm are  $\mathcal{O}(\lceil m^2/\omega \rceil)$  and  $\mathcal{O}(\sigma \lceil m/\omega \rceil)$ , respectively.

Bit-parallel algorithms are extremely fast as long as the pattern fits in a computer word. Specifically, when  $m \leq \omega$ , the Shift-And and BNBM 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  $\omega$ , the configuration bit-vector and all auxiliary bit-vectors need to be split over  $\lceil m/\omega \rceil$  multiple words. For this reason the performance of the Shift-And and BNBM algorithms, and of bit-parallel algorithms more in general, degrades considerably as  $\lceil m/\omega \rceil$  grows.

### 3.4. Constant-space Algorithms

The requirement for constant-space string matching is to use only variables in addition to the inputs  $p$  and  $t$ . The two main constant-space string matching algorithms are the Galil and Seiferas Algorithm [Galil and Seiferas 1983] (GS for short) and the Two-Way String Matching Algorithm by Crochemore and Perrin [Crochemore and Perrin 1991].

The GS algorithm uses a constant  $k$  (Galil and Seiferas suggest that practically this constant could be equal to 4).

They make use of a function *reach* which is defined for  $0 \leq i < m$  as follows:

$$\text{reach}(i) = i + \max\{i' \leq m - i \mid p[0 \dots i'] = p[i + 1 \dots i' + i + 1]\}.$$

Then a prefix  $p[0 \dots i]$  of  $p$  is a *prefix period* if it is basic and  $\text{reach}(i) \geq k \times i$ .

The preprocessing phase of the Galil-Seiferas algorithm consists in finding a decomposition  $uv$  of  $x$  such that  $v$  has at most one prefix period and  $|u| = \mathcal{O}(\text{per}(v))$ . Such a decomposition is called a *perfect factorization*.

Then the searching phase consists in scanning the text  $t$  for every occurrences of  $v$  and when  $v$  occurs to check naively if  $u$  occurs just before in  $t$ .

The aim of the preprocessing phase is to find a perfect factorization  $uv$  of  $x$  where  $u = p[0 \dots s - 1]$  and  $v = p[s \dots m - 1]$ .

At the beginning searching phase one of the following conditions holds:

- (i).  $p[s \dots m - 1]$  has at most one prefix period;
- (ii). if  $p[s \dots m - 1]$  does have a prefix period, then its length is  $p_1$ ;

- (iii).  $p[s \dots s + p_1 + q_1 - 1]$  has shortest period of length  $p_1$ ;
- (iv).  $p[s \dots s + p_1 + q_1]$  does not have period of length  $p_1$ .

The pattern  $p$  is of the form  $p[0 \dots s - 1]p[s \dots m - 1]$  where  $p[s \dots m - 1]$  is of the form  $z^\ell z' a z''$  with  $z$  basic,  $|z| = p_1$ ,  $z'$  prefix of  $z$ ,  $z' a$  not a prefix of  $z$  and  $|z^\ell z'| = p_1 + q_1$ .

It means that when searching for  $p[s \dots m - 1]$  in  $t$ , if  $p[s \dots s + p_1 + q_1 - 1]$  has been matched a shift of length  $p_1$  can be performed and the comparisons are resumed with character  $p[s + q_1]$ ; otherwise if a mismatch occurs with  $p[s + q]$  with  $q \neq p_1 + q_1$  then a shift of length  $q/k + 1$  can be performed and the comparisons are resumed with  $p[0]$ . This gives an overall linear number of text character comparisons.

The preprocessing phase of the Galil-Seiferas algorithm is in  $\mathcal{O}(m)$  time and constant space complexity. The searching phase is in  $\mathcal{O}(n)$  time complexity. At most  $5n$  text character comparisons can be done during this phase.

For the Two Way algorithm the pattern  $p$  is factorized into two parts  $p_\ell$  and  $p_r$  such that  $p = p_\ell p_r$ . Then the searching phase consists in comparing the characters of  $p_r$  from left to right and then, if no mismatch occurs during that first stage, in comparing the characters of  $p_\ell$  from right to left in a second stage.

The preprocessing phase of the algorithm consists then in choosing a good *factorization*  $p_\ell p_r$ . Let  $(u, v)$  be a factorization of  $p$ . A *repetition* in  $(u, v)$  is a word  $w$  such that the two following properties hold:

- (i).  $w$  is a suffix of  $u$  or  $u$  is a suffix of  $w$ ;
- (ii).  $w$  is a prefix of  $v$  or  $v$  is a prefix of  $w$ .

In other words  $w$  occurs at both sides of the cut between  $u$  and  $v$  with a possible overflow on either side. The length of a repetition in  $(u, v)$  is called a *local period* and the length of the smallest repetition in  $(u, v)$  is called *the local period* and is denoted by  $r(u, v)$ .

Each factorization  $(u, v)$  of  $p$  has at least one repetition. It can be easily seen that  $1 \leq r(u, v) \leq m$ .

A factorization  $(u, v)$  of  $p$  such that  $r(u, v) = \text{per}(p)$  is called a *critical factorization* of  $p$ . If  $(u, v)$  is a critical factorization of  $p$  then at the position  $|u|$  in  $p$  the global and the local periods are the same. The Two Way algorithm chooses the critical factorization  $(p_\ell, p_r)$  such that  $|p_\ell| < \text{per}(p)$  and  $|p_\ell|$  is minimal.

To compute the critical factorization  $(p_\ell, p_r)$  of  $p$  the algorithm first computes the maximal suffix  $z$  of  $p$  for the order  $\leq$  and the maximal suffix  $\tilde{z}$  for the reverse order of  $\leq$ . Then  $(p_\ell, p_r)$  is chosen such that  $|p_\ell| = \max\{|z|, |\tilde{z}|\}$ . The preprocessing phase can be done in  $\mathcal{O}(m)$  time and constant space complexity.

The searching phase of the Two Way algorithm consists in first comparing the character of  $p_r$  from left to right, then the character of  $p_\ell$  from right to left. When a mismatch occurs when scanning the  $k$ -th character of  $p_r$ , then a shift of length  $k$  is performed. When a mismatch occurs when scanning  $p_\ell$  or when an occurrence of the pattern is found, then a shift of length  $\text{per}(p)$  is performed.

Such a scheme leads to a quadratic worst case algorithm, this can be avoided by a prefix memorization: when a shift of length  $\text{per}(p)$  is performed the length of the matching prefix of the pattern at the beginning of the window (namely  $m - \text{per}(p)$ ) after the shift is memorized to avoid to scan it again during the next attempt.

The searching phase of the Two Way algorithm can be done in  $\mathcal{O}(n)$  time complexity. The algorithm performs  $2n - m$  text character comparisons in the worst case. Breslauer [Breslauer 1996] designed a variation of the Two Way algorithm which performs less than  $2n - m$  comparisons in the worst case using constant space.

### 3.5. Real time Algorithms

For real time string matching the time difference between the processing of two consecutive text characters must be bounded by a constant. In [Galil 1981], Galil introduced a predictability condition which is a sufficient condition for an on-line algorithm to be transformable into a real-time algorithm. He then show that the KMP algorithm satisfies the predictability condition. So does, for instance, the Simon algorithm. More recently Gąsieniec and Kolpakov [Gąsieniec and Kolpakov 2004] designed a real-time string matching algorithm using sublinear extra-space.

## 4. ALGORITHMS AFTER 2000

We are now ready to present the most recent string matching algorithms. In addition to the classification used in the previous section (Comparison based, automata based, bit-parallel, constant-space and real time) we also classify these recent algorithms according to the main idea that leads to them. We retain nine main categories: variants of the Boyer-Moore algorithm, variants of the Backward-Oracle-Matching algorithm, variants of the Backward-Nondeterministic-Dawg-Matching algorithm, hashing and  $q$ -grams, partitioning of the pattern, large window, two windows, two automata and SIMD. Note however that these categories are not exclusive but we classify each algorithm into its main category.

Section 4.1 presents the algorithms based on comparisons. Section 4.2 contains the algorithms based on automata. In Section 4.3 the algorithms based on bit-parallelism are presented and Section 4.4 presents constant-space algorithms.

### 4.1. Comparison Based Algorithms

We now present the most recent string matching based on character comparisons. Section 4.1.1 presents variants of the Boyer-Moore algorithm. Section 4.1.2 presents an algorithm using two windows on the text in parallel. Section 4.1.3 presents algorithms using hashing or  $q$ -grams and Section 4.1.4 presents an algorithm using SIMD instructions that proves to be very efficient in practice for long patterns.

*4.1.1. Variants of the Boyer-Moore Algorithm.* In this section we present recent string matching algorithms based on character comparisons that are variants of the classical Boyer-Moore algorithm. Specifically we present the following algorithms:

- The AKC algorithm, a variant of the Apostolico-Giancarlo algorithm [Apostolico and Giancarlo 1986] that remembers all the suffixes of the pattern found in the text and that computes the shifts accordingly at the end of each attempt;
- Algorithms in the Fast-Search family, based on the fact that at the end of each attempt the shift is computed with the bad character rule only if the first comparison of the attempt is a mismatch and the shift is computed using the good suffix rule otherwise;
- The SSABS and TVSBS algorithms, which scan first the rightmost character of the pattern, then the leftmost and finally the remaining characters of the pattern;
- The Boyer-Moore Horspool algorithm using probabilities, which scans the characters of the pattern according to their frequencies;
- The FJS algorithm, an hybrid between KMP and QS algorithms;
- The 2Block algorithm, which keeps track of all the previously matched characters within the current window and does not to move the reading position unconditionally to the end of the pattern when a mismatch occurs.

*The AKC Algorithm.* The AKC algorithm [Ahmed et al. 2003] is a variant of the Apostolico-Giancarlo algorithm [Apostolico and Giancarlo 1986]. At each attempt it scans the characters of the window from right to left and remembers every factors

of the text that matches a suffix of the pattern during previous attempts. So at the end of each attempt when the pattern is shifted, the AKC algorithm guarantees that each text factor that already matched a suffix of the pattern still match a factor of the pattern.

More specifically assume that the pattern is aligned with  $t[s \dots s + m - 1]$ . Let  $t[s \dots s + m - 1] = u_k v_{k-1} b_{k-1} u_{k-1} \dots v_1 b_1 u_1 v_0 b_0 u_0$  be such that  $u_i, v_i \in \Sigma^*$ ,  $b_i \in \Sigma$  for  $0 \leq i < k$  and  $u_k \in \Sigma^*$ . Suppose moreover that, for  $0 \leq i \leq k$ ,  $u_i$  is a suffix (possibly empty) of  $p$  matched during the previous and the current attempts while  $b_i u_i$  is not a suffix of  $p$ . Let also  $spos(u_i) = s + |u_k| + \sum_{\ell=i+1}^{k-1} (|v_\ell b_\ell u_\ell|) + |v_i b_i|$  and  $epos(u_i) = spos(u_i) + |u_i| - 1$  be respectively the starting and ending positions of  $u_i$  in the text, for  $0 \leq i \leq k$ .

Then the AKC algorithm shifts the window by the smallest amount  $\delta$  such that  $p[h] = t[s + h]$  whenever  $0 \leq spos(u_i) - s + \delta \leq h \leq epos(u_i) - s + \delta < m$  for some  $0 \leq i \leq k$ . This kind of information cannot be preprocessed so the actual shift is computed after each attempt. This improves the shift length upon the Apostolico-Giancarlo and ensures that the  $u_i$ 's can be unconditionally jumped over during next attempts thus leading to at most  $n$  comparisons of text characters overall. However the computation of a single shift can be performed in  $\mathcal{O}(m^2)$  in the worst case. This would lead to a shift of length  $m$ . Thus overall the computation of all the shifts can be performed in  $\mathcal{O}(mn)$ . Then the time complexity of the searching phase of the AKC algorithm is  $\mathcal{O}(mn)$  while the preprocessing phase can be performed in  $\mathcal{O}(m + \sigma)$  space and time.

**Fast-Search Algorithms.** The family of Fast-Search algorithms [Cantone and Faro 2005] consists in three different variants of the Boyer-Moore algorithm. The progenitor of this family of algorithms, the Fast-Search algorithm (FS for short), was presented in [Cantone and Faro 2003a] and is a very simple, yet efficient, variant of the Boyer-Moore algorithm. During each attempt the pattern is compared with the current window proceeding from right to left. Then the Fast-Search algorithm computes its shift increments by applying the Horspool bad-character rule (4) if and only if a mismatch occurs during the first character comparison, namely, while comparing characters  $p[m - 1]$  and  $t[s + m - 1]$ , where  $s$  is the current shift. Otherwise it uses the good-suffix rule (2). A more effective implementation of the Fast-Search algorithm is obtained by making use of a fast-loop, based on the Horspool bad-character rule, in order to locate an occurrence of the rightmost character of the pattern. Then the subsequent matching phase can start with the  $(m - 1)$ -st character of the pattern. At the end of the matching phase the algorithm uses the good-suffix rule for shifting.

The Backward-Fast-Search algorithm (BFS for short) is the second algorithm of the Fast-Search family. It has been presented in [Cantone and Faro 2003b] and independently developed in [Crochemore and Lecroq 2008]. The algorithm makes use of the *backward good suffix* rule, which can be obtained by merging the standard good-suffix rule (2) with the bad-character rule (3) as follows. If, for a given shift  $s$ , the first mismatch occurs at position  $i$  of the pattern  $p$ , i.e.  $p[i + 1 \dots m - 1] = t[s + i + 1 \dots s + m - 1]$  and  $p[i] \neq t[s + i]$ , then the backward good suffix rule proposes to align the substring  $t[s + i + 1 \dots s + m - 1]$  with its rightmost occurrence in  $p$  preceded by the *backward* character  $t[s + i]$ . If such an occurrence does not exist, the backward good suffix rule proposes a shift increment which allows to match the longest suffix of  $t[s + i + 1 \dots s + m - 1]$  with a prefix of  $p$ . More formally, this corresponds to increment the shift  $s$  by  $bgs_p(i + 1, t[s + i])$ , where

$$bgs_p(j, c) = \min\{1 \leq k \leq m \mid p[j - k \dots m - k - 1] \sqsupset p \text{ and } (k \leq j - 1 \rightarrow p[j - 1] = c)\} ,$$

for  $j = 0, 1, \dots, m$  and  $c \in \Sigma$ .



Finally the Forward-Fast-Search algorithm [Cantone and Faro 2004] (FFS for short) maintains the same structure as the Fast-Search algorithm, but it is based upon a modified version of the good-suffix rule, called *forward good-suffix* rule, which uses a look-ahead character to determine larger shift advancements. Thus, if the first mismatch occurs at position  $i < m-1$  of the pattern  $p$ , the forward good-suffix rule suggests to align the substring  $t[s+i+1..s+m]$  with its rightmost occurrence in  $p$  preceded by a character different from  $p[i]$ . If such an occurrence does not exist, the forward good-suffix rule proposes a shift increment which allows to match the longest suffix of  $t[s+i+1..s+m]$  with a prefix of  $p$ . This corresponds to advance the shift  $s$  by  $fgs_p(i+1, t[s+m])$  positions, where

$$fgs_p(j, c) =_{\text{Def}} \min(\{1 \leq k \leq m \mid p[j-k..m-k-1] \sqsupset p \text{ and } (k \leq j-1 \rightarrow p[j-1] \neq p[j-1-k]) \text{ and } p[m-k] = c\} \cup \{m+1\}),$$

for  $j = 0, 1, \dots, m$  and  $c \in \Sigma$ .

The backward good-suffix rule and the forward good-suffix rule require tables of size  $m \cdot \sigma$ , which can be constructed in time  $\mathcal{O}(m \cdot \max(m, \sigma))$ . In [Crochemore and Lecroq 2008] the authors proposed an algorithm for computing the backward good suffix rule in  $\mathcal{O}(m)$ -time for a pattern of length  $m$ .

*SSABS and TVSBS Algorithms.* The SSABS algorithm [Sheik et al. 2004] was presented as a mix of the shifting strategy of the Quick-Search algorithm [Sunday 1990] and the testing strategy of the Raita algorithm [Raita 1992].

Specifically the order of comparisons between the pattern and the current window of the text is carried out, in accordance with the Raita algorithm, by comparing first the rightmost character of the window against its counterpart in the pattern, and after a match, by further comparing the leftmost character of the window and the leftmost character of the pattern. By doing so, an initial resemblance can be established between the pattern and the window, and the remaining characters are compared from right to left until a complete match or a mismatch occurs.

After each attempt, the shift of the window is gained by the Quick-Search bad character rule (5) for the character that is placed next to the window.

According to the authors the reason for successively comparing the rightmost and leftmost characters first, and then continuing the comparison of the other characters, is mainly due to the fact that the dependency of the neighboring characters is strong compared to the other characters. Hence, it is always better to postpone the comparisons on the neighboring characters.

The performance of the SSABS algorithm have been later improved in the TVSBS algorithm [Thathoo et al. 2006] which uses the shift proposed by the Berry-Ravindran bad character rule (6), leading to lesser number of characters comparisons and greater shift advancements.

Both SSABS and TVSBS algorithms have an  $\mathcal{O}(nm)$  worst case time complexity and require  $\mathcal{O}(\sigma)$  and  $\mathcal{O}(\sigma^2)$  additional space, respectively, for computing the shift tables.

*Boyer-Moore-Horspool Algorithm Using Probabilities.* A modified version of the Horspool algorithm making use of the probabilities of the symbols within the pattern has been presented in [Nebel 2006]. Provided that we have different probabilities for different symbols, like for natural language texts or biological data, the algorithm tries to decrease the number of comparisons needed for searching the pattern in the text. A similar modification of the Boyer-Moore algorithm has been presented in the Optimal-Mismatch algorithm [Sunday 1990].

Specifically the idea consists in changing the order in which the symbols of the pattern are compared to the symbols of the current window of the text such that the probability of a mismatch is statistically maximized. To proceed this way it is necessary to perform an additional preprocessing step which takes  $\mathcal{O}(m \log m)$ -time and  $\mathcal{O}(m)$ -space. The author presented also an analysis of the algorithm in the average case showing that, in favorable cases, the new solution needs less than 80% of running time of the original Horspool algorithm.

*Franek-Jennings-Smyth Algorithm.* The Franek-Jennings-Smyth string matching algorithm [Franek et al. 2007] (FJS for short) is a simple hybrid algorithm which mixes the linear worst case time complexity of Knuth-Morris-Pratt algorithm and the sub-linear average behavior of Quick-Search algorithm.

Specifically the FJS algorithm searches for matches of  $p$  in  $t$  by shifting a window of size  $m$  from left to right along  $t$ . Each attempt of the algorithm is divided into two steps. During the first step, in accordance with the Quick-Search approach, the FJS algorithm first compares the rightmost character of the pattern,  $p[m-1]$ , with its corresponding character in the text, i.e.  $t[s+m-1]$ . If a mismatch occurs, a Quick-Search shift (5) is implemented, moving  $p$  along  $t$  until the rightmost occurrence in  $p$  of the character  $t[s+m]$  is aligned with position  $s+m$  in the text. At this new location, the rightmost character of  $p$  is again compared with the corresponding text position. Only when a match is found the FJS algorithm invokes the second step. Otherwise another Quick-Search shift occurs.

The second step of the algorithm consists in a Knuth-Morris-Pratt pattern-matching starting from the leftmost character  $p[0]$  and, if no mismatch occurs, extending as far as  $p[m-2]$ . Then whether or not a match of  $p$  is found, a Knuth-Morris-Pratt shift is eventually performed followed by a return to the first step.

The preprocessing time of the algorithm takes  $\mathcal{O}(m)$ -time for computing the failure function of Knuth-Morris-Pratt, and  $\mathcal{O}(m + \sigma)$ -time in order to compute the Quick-Search bad character rule. The authors showed that the worst case number of character comparisons is bounded by  $3n - 2m$ , so that the corresponding searching phase requires  $\mathcal{O}(n)$ -time. The space complexity of the algorithm is  $\mathcal{O}(m + \sigma)$ .

*2Block Algorithm.* In [Sustik and Moore 2007] the authors proposed a new string matching algorithm inspired by the original Boyer-Moore algorithm. The two key ideas of their improved algorithm are to keep track of all the previously matched characters within the current window and not to move the reading position unconditionally to the end of the pattern when a mismatch occurs. The resulting algorithm, named 2Block algorithm, has an increased average shift amounts and guarantees that any character of the text is read at most once.

Specifically the 2Block algorithm remembers two matching blocks of characters, the left and the right block. The left block always starts at the beginning of the pattern and the number of characters contained in it is indicated by  $\ell$ . The right block is described by its endpoints,  $r_s$  (right start) and  $r_e$  (right end). For each alignment at position  $s$  of the text  $t[s \dots s + \ell - 1] = p[0 \dots \ell - 1]$  and  $t[s + r_s \dots r_e - 1] = p[r_s \dots r_e - 1]$ .

The algorithm attempts to extend the right block with every read. If possible, the right block is extended to the right. In case of an empty right block ( $r_s = r_e$ ), the character aligned with the rightmost character of the pattern is read. Both blocks may degenerate into the empty string ( $\ell = 0$  and  $r_s = r_e$ ).

When the text character specified by the current alignment and the relative reading position does not agree with the pattern, the 2Block algorithm shifts with the smallest possible value, such that in the new alignment the pattern agrees with all the previously read characters. For an efficient implementation, the algorithm builds in a pre-

processing phase a state transition table that only implicitly contains the information on the two blocks of already matched characters.

The 2Block algorithm has a linear worst case time complexity but performs especially well when the alphabet size is small. In addition the state transition table requires  $\mathcal{O}(\sigma m^3)$ -extra space to be stored and can be naively computed in polynomial time.

**4.1.2. Two Windows.** In this section we present a recent string matching algorithm, named Two-Sliding-Windows, based on character comparisons and which uses two text windows while searching for all occurrences of the pattern. It scans in parallel the left part and the right part of the text and uses a shift rule of the Berry-Ravindran algorithm.

*Two-Sliding-Windows Algorithm.* The Two-Sliding-Windows string matching algorithm [Hudaib et al. 2008] (TSW for short) is an efficient variation of the Berry-Ravindran algorithm [Berry and Ravindran 1999].

Specifically the TSW algorithm divides the text into two parts of size  $\lceil n/2 \rceil$  and searches for matches of  $p$  in  $t$  by shifting two windows of size  $m$  along  $t$  simultaneously. The first window scans the left part of the text, proceeding from left to right, while the second window slides from right to left scanning the right part of the text. The two windows search the text in parallel, which makes the TSW algorithm suitable for parallel processors structures.

The TSW algorithm uses a shift rule which is based on the Berry-Ravindran bad character rule (6) to get better shift values during the searching phase. However the TSW algorithm computes the shift values only for the pattern characters using, for each text window, an array of size  $m - 1$ . This is in contrast with the original Berry-Ravindran algorithm which uses a two-dimensional array of dimension  $\mathcal{O}(\sigma^2)$ . This reduces the memory requirements needed to store the shift values but postpones a bit of work to the next searching phase.

More formally during the preprocessing phase the algorithm computes two tables of size  $m - 1$ ,  $nextl$  and  $nextl$ , to be used for shifting in the left and the right part of the text, respectively. If we indicate with  $\bar{p}$  the reverse of the pattern, then we have for  $0 \leq i < m - 1$

$$nextl[i] = brbc_p(p[i], p[i + 1]), \quad \text{and} \quad nextl[m - 1 - i] = brbc_{\bar{p}}(\bar{p}[i], \bar{p}[i + 1]).$$

The two tables can be computed in  $\mathcal{O}(m)$ -time and -space.

Then the searching phase can be divided into two steps. During the first step the two windows are naively compared with the pattern in order to report an occurrence, proceeding from right to left (for the leftmost window) and from left to right (for the rightmost window). Let  $s$  and  $s'$  be the two shift alignments at the end of the first step, so that the pattern is aligned with the windows  $t[s \dots s + m - 1]$  and  $t[s' \dots s' + m - 1]$ , with  $s < s'$ . Then the leftmost window is shifted to the right by an amount equal to  $shiftl[t[s + m], t[s + m + 1]]$  where, for  $a, b \in \Sigma$ , we define

$$shiftl[a, b] = \min \left\{ \begin{array}{ll} 1 & \text{if } p[m - 1] = a \\ nextl[i] & \text{if } p[i] = a \text{ and } p[i + 1] = b \\ m + 1 & \text{if } p[0] = b \\ m + 2 & \text{otherwise} \end{array} \right\}.$$

Similarly the rightmost window is shifted to the left by an amount equal to  $shiftr[t[s' - 1], t[s' - 2]]$  where the symmetrical function,  $shiftr$ , can be obtained from the function defined above, by substituting  $nextl$ , in place of  $nextl$ , and  $\bar{p}$  in place of  $p$ . The TSW algorithm requires  $\mathcal{O}(m)$  worst case time for computing the shift advancement in each step of the searching phase.

Both steps are repeated until both windows are positioned beyond position  $\lceil n/2 \rceil$ . An additional comparison is performed in the case an occurrence of the pattern is found in the middle of the text. The TSW algorithm has a  $\mathcal{O}(nm^2)$  worst case time complexity and requires  $\mathcal{O}(m)$  additional space.

**4.1.3. Hashing and  $q$ -grams.** In this section we present the following string matching algorithms based on character comparisons that use hashing or  $q$ -grams:

- the Hash $_q$  family of algorithms, an adaptation of the Wu-Manber algorithm to single pattern matching;
- the Boyer-Moore Horspool algorithm with  $q$ -grams, which enhances the Horspool algorithm by using a hash function on the rightmost  $q$  characters of the window at each attempt;
- the Genomic Oriented Rapid Algorithm, which hashes every 2-grams on the underlying pattern.

*Hashing Algorithms.* Algorithms in the Hash $_q$  family have been introduced in [Lecroq 2007] where the author presented an adaptation of the Wu and Manber multiple string matching algorithm [Wu and Manber 1994] to single string matching problem.

The idea of the Hash $_q$  algorithm is to consider factors of the pattern of length  $q$ . Each substring  $w$  of such a length  $q$  is hashed using a function  $h$  into integer values within 0 and  $K - 1$ , where  $K$  is a parameter of the algorithm which strongly affects the performance and the resulting complexity.

Then the algorithm computes in a preprocessing phase a function

$$\text{shift} : \{0, 1, \dots, K - 1\} \rightarrow \{0, 1, \dots, m - q\}$$

where for each  $0 \leq c < K$  the value  $\text{shift}(c)$  is defined by

$$\text{shift}(c) = \min \left( \{0 \leq k < m - q \mid h(p[m - k - q .. m - k - 1]) = c\} \cup \{m - q\} \right).$$

The time complexity of the preprocessing phase is equal to  $\mathcal{O}(K + m \times T(h))$ , where  $T(h)$  is the time complexity required for computing the hash value for a string of length  $q$ . The space complexity required for storing the  $\text{shift}$  table is  $\mathcal{O}(K)$ .

Then the Hash $_q$  algorithm searches for matches of  $p$  in  $t$  by shifting a window of size  $m$  from left to right along  $t$ . For each shift  $s$  of the pattern in the text, the algorithm reads the substring  $w = t[s + m - q .. s + m - 1]$  of length  $q$  and computes the hash value  $h(w)$ . If  $\text{shift}[h(w)] > 0$  then a shift of length  $\text{shift}[h(w)]$  is applied. Otherwise, when  $\text{shift}[h(w)] = 0$  the pattern  $p$  is naively checked in the text. In this case, at the end of the test, a shift of length  $(m - 1 - \alpha)$  is applied where

$$\alpha = \max\{0 \leq j \leq m - q \mid h(p[j .. j + q - 1]) = h(p[m - q + 1 .. m - 1])\}.$$

The search of the Hash $_q$  algorithm has a  $\mathcal{O}(n(T(h) + m))$  time complexity.

In [Lecroq 2007] the author succeeded to obtain algorithms with very good performance by computing the hash value with  $K = 256$ , thus using a single byte to store each hash entry. Moreover the hash function  $h$  is computed in a very simple, yet efficient, way by using the following formula, for a string  $w$  of length  $q$

$$h(w) = 2^{q-1}w[0] + 2^{q-2}w[1] + \dots + 2w[q-2] + w[q-1]$$

which turns out to be easily implemented into C-programming code.

*Boyer-Moore Horspool with  $q$ -grams.* The BMH $_q$  algorithm [Kalsi et al. 2008] is an efficient modification of the Horspool algorithm for the DNA alphabet. Instead of inspecting a single character at each alignment of the pattern, their algorithm reads a  $q$ -gram

and computes an integer called *fingerprint* from it. The idea is simple and consists in mapping the ASCII codes of characters in the DNA alphabet, i.e.  $a, c, g$ , and  $t$  to a range of 4 characters. Specifically the algorithm uses a mapping  $r : \{a, c, g, t\} \rightarrow \{0, 1, 2, 3\}$ , in order to limit the computation to the effective alphabet of four characters.

The fingerprint computed for a  $q$ -gram is simply a reversed number of base 4. A separate transformation table  $h_i$  is used for each position  $i$  in the  $q$ -gram and multiplications are incorporated during preprocessing into the tables:  $h_i(c) = r(c)4^i$ , for all  $c \in \{a, c, g, t\}$ . For  $q = 4$ , the fingerprint of  $c_0c_1c_2c_3$  is computed as  $h_0(c_0) + h_1(c_1) + h_2(c_2) + h_3(c_3)$ .

At each alignment of the pattern the last  $q$ -gram of the pattern is compared with the corresponding  $q$ -gram in the current window of the text by testing the equality of their fingerprints. If the fingerprints match, a potential occurrence has been found, which has to be naively checked. However observe that we need to check only the first  $m - q$  characters of the pattern because our fingerprint method does not cause hash collisions (if the searched text contains only DNA characters).

The  $q$ -gram approach is valid also for larger alphabets, although with a larger alphabet the optimal value of  $q$  is smaller. A new variation BMH2 was created based on BMH $_q$ . The range of the ASCII code mapping  $r$  was increased from 4 to 20, to cover the amino acid alphabet instead of the DNA alphabet.

The BMH $_q$  algorithm needs  $\mathcal{O}(m + \sigma)$  extra space and has a  $\mathcal{O}(nm)$  worst case time complexity, although sublinear in the average.

*The Genomic Oriented Rapid Algorithm.* The Genomic-oriented Rapid Algorithm for String Pattern-match [Deusdado and Carvalho 2009] (GRASPm for short), is an algorithm which uses a shifting method based on the Horspool bad-character rule (4) and a filtering method based on an hash function computed on 2-grams in the pattern.

During the preprocessing phase the GRASPm algorithm stores, for each character  $c$  of the alphabet, the list of all positions in  $p$  where the rightmost character of  $p$  is preceded by the character  $c$ . Formally the algorithm computes a function  $z : \Sigma \rightarrow \text{Pow}\{0, 1, \dots, m - 1\}$  where, for all  $c \in \Sigma$

$$z(c) = \{0 \leq i < m \mid (p[i - 1] = c \text{ or } i = 0) \text{ and } p[i] = p[m - 1]\}.$$

Then, during the searching phase, the algorithm uses a fast loop based on the bad character rule for locating in the text  $t$  an occurrence of the rightmost character of the pattern. When an occurrence of  $p[m - 1]$  is located at position  $j$  in  $t$  the GRASPm algorithm aligns  $p$  according to the positions in the list  $z(t[j - 1])$ , reporting the occurrences of  $p$ , if any. Specifically for each position  $k$  in the list  $z(t[j - 1])$  the algorithm naively checks if  $p$  is equal to  $t[j - k .. j - k + m - 1]$ . Then it shifts the pattern by  $m$  positions to the right.

The resulting algorithm has an  $\mathcal{O}(nm)$  worst case time complexity and requires  $\mathcal{O}(\sigma + m)$  space for computing the hash table  $z$  and the bad character shift table.

**4.1.4. SIMD.** In this section we present a recent string matching algorithm based on character comparisons that use SIMD instructions. The resulting algorithm, named SSEF, turns out to be very fast in practice, especially for long patterns. However, the SSEF algorithm, does not run for pattern with a length less than 32 characters.

*SSEF Algorithm.* SIMD instructions exist in many recent microprocessors supporting parallel execution of some operations on multiple data simultaneously via a set of special instructions working on limited number of special registers. The algorithm introduced in [Külekci 2009] is referred as SSEF algorithm and uses the Intel streaming SIMD extensions technology (SEEF is for Streaming SIMD Extension Filtering).



The SSEF algorithm is designed to be effective on long patterns, where the lower limit for  $m$  is 32 ( $m \geq 32$ ). Although it is possible to adapt the algorithm for lesser lengths, the performance gets worse under 32.

Given a text  $t$  and a pattern  $p$  of lengths  $n$  and  $m$  (bytes) respectively, the number of 16-byte blocks in  $t$  and  $p$  are denoted by  $N = \lceil n/16 \rceil$  and  $M = \lceil m/16 \rceil$  respectively. Let  $T$  and  $P$  be the 16-byte sequence relative to  $t$  and  $p$ , respectively, with  $|T| = N$  and  $|P| = M$ . In addition let  $L = \lfloor m/16 \rfloor - 1$  be the zero-based address of the last 16-byte block of  $P$  whose individual bytes are totally composed of pattern bytes without any padding.

The corresponding filter of a 16-byte sequence is the 16 bits formed by concatenating the sign bits (i.e. the leftmost bits) of each byte after shifting left each block of  $K$  bits, where the value  $K$  depends on the alphabet size and character distribution of the text. For instance, while searching on a DNA sequence, the author suggested that setting  $K = 5$  would be a good choice.

The preprocessing stage of the SSEF algorithm consists of computing all possible filter values of the pattern according to the all possible alignments of the pattern with the text. Such values are stored in a linked list, named *FList*.

In particular the computation of the filter  $f$  of a 16-byte block  $B$  is performed by 2 SSE intrinsic functions as  $f = \text{mm\_movemask\_epi8}(\text{mm\_slli\_epi64}(B, K))$ , where the instruction  $\text{mm\_slli\_epi64}(B, K)$  left shifts the corresponding 16 bytes of  $B$  by  $K$  bits and stores the result in a temporary 128-bit register. The second instruction  $\text{mm\_movemask\_epi8}(\text{tmp}_{128})$  returns a 16-bit mask composed of the sign bits of the individual 16 bytes forming the 128-bit value.

The main loop of the algorithm investigates 16-byte blocks of the text  $T$  in steps of length  $L$ . If the filter  $f$  computed on  $D[i]$ , where  $i = zL + L$ , and  $0 \leq z < \lfloor N/L \rfloor$ , is not empty, then the appropriate positions listed in  $FList[f]$  are verified accordingly. In particular  $FList[f]$  contains a linked list of integers marking the beginning of the pattern. If a value  $j$  is contained in  $FList[f]$ , with  $0 \leq j < 16L$ , the pattern potentially begins at  $t[(i - L)16 + j]$ . In that case, a naive verification is to be performed between  $p$  and  $t[(i - L)16 + j \dots (i - L)16 + j + m - 1]$ .

The SSEF algorithm has an  $\mathcal{O}(nm)$  worst case time complexity but performs well in practical cases. In particular the author showed that the algorithm has a  $\mathcal{O}(n/m + nm/2^{16})$  average case time complexity. In addition the algorithm requires an extra space to store the  $2^{16} = 65536$  items of the *FList* linked list.

#### 4.2. Algorithms based on Deterministic Automata

We now present the most recent string matching based on automata. In particular in Section 4.2.1 we present variants of the Backward-Oracle-Matching algorithm. Then in Section 4.2.2 we present an algorithm using two automata. Finally in Section 4.2.3 we present algorithms using wide windows.

**4.2.1. Variants of the BOM Algorithm.** In this section we present the following string matching algorithms based on automata that are variants of the Backward-Oracle-Matching algorithm:

- the Extended BOM and the Simplified Extended BOM algorithms, which compute in one step the transitions for the two rightmost characters of the pattern;
- the Forward BOM and the Simplified Forward BOM algorithms, which compute in one step the transition for the rightmost character of the pattern and the character next to its right.

*The Extended-Backward-Oracle-Matching.* The Extended-BOM string matching algorithm [Faro and Lecroq 2008] is a very fast and flexible variation of the Backward-

Oracle-Matching algorithm. It introduces a fast-loop with the aim of obtaining better results on the average. In the original paper the authors discussed the application of different variations of the fast-loop and presented experimental results in order to identify the best choice.

Specifically the variation of the algorithm they proposed tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition. Moreover, in order to improve the performances, they proposed to encapsulate the two first transitions of the oracle in a function  $\lambda : (\Sigma \times \Sigma) \rightarrow Q$  defined, for each  $a, b \in \Sigma$ , by

$$\lambda(a, b) = \begin{cases} \perp & \text{if } \delta(m, a) = \perp \\ \delta(\delta(m, a), b) & \text{otherwise.} \end{cases}$$

At the end of the fast-loop the algorithm could start standard transitions with the Oracle from state  $q = \lambda(t[s+m-1], t[s+m-2])$  and character  $t[s+m-3]$ . The function  $\lambda$  can be implemented with a two-dimensional table in  $\mathcal{O}(\sigma^2)$  time and space.

In [Fan et al. 2009] a simplified version of the Extended-BOM algorithm was presented where the two-dimensional table  $\lambda$  is implemented by a one-dimensional array using the formula:  $\lambda(a, b) = \lambda(a\sigma + b)$  for  $a, b \in \Sigma$ . Especially on ASCII code, where  $\sigma = 256$ , the formula can be efficiently accomplished as  $\lambda(a, b) = \lambda(a \ll 8 + b)$ . The resulting algorithm is called Simplified-EBOM (SEBOM).

*The Forward-Backward-Oracle-Matching.* The Forward-BOM string matching algorithm [Faro and Lecroq 2008] extends further the improvements introduced in the Extended-BOM algorithm by looking for the character which follows the current window (the forward character) while computing the shift advancement. Thus it mixes the ideas of the Extended-BOM algorithm with those of the Quick-Search algorithm.

In order to take into account the forward character of the current window of the text without skipping safe alignment the authors introduced the *forward factor oracle* of the reverse pattern. The forward factor oracle of a word  $p$ ,  $FOracle(p)$ , is defined as an automaton which recognizes at least all the factors of  $p$ , possibly preceded by a word  $x \in \Sigma \cup \{\varepsilon\}$ . More formally the language recognized by  $FOracle(p)$  is defined by

$$\mathcal{L}(FOracle(p)) = \{xw \mid x \in \Sigma \cup \{\varepsilon\} \text{ and } w \in \mathcal{L}(Oracle(p))\}.$$

Observe that in the previous definition the prefix  $x$  could be the empty string. Thus if  $w$  is a word recognized by the factor oracle of  $p$  then the word  $cw$  is recognized by the forward factor oracle, for all  $c \in \Sigma \cup \{\varepsilon\}$ .

The forward factor oracle of a word  $p$  can be constructed, in time  $\mathcal{O}(m + \Sigma)$ , by simply extending the factor oracle of  $p$  with a new initial state which allows to perform transitions starting at the text character of position  $s + m$  of the text, avoiding to skip valid shift alignments.

Suppose  $Oracle(p) = \{Q, q_m, Q, \Sigma, \delta\}$ , for a pattern  $p$  of length  $m$ . It is possible to construct  $FOracle(p)$  by adding a new initial state  $q_{m+1}$  and introducing transitions from state  $q_{m+1}$ . More formally, given a pattern  $p$  of length  $m$ ,  $FOracle(p)$  is an automaton  $\{Q', q_{m+1}, Q', \Sigma, \delta'\}$ , where (i)  $Q' = Q \cup \{q_{m+1}\}$ , (ii)  $q_{m+1}$  is the initial state, (iii) all states are final, (iv)  $\delta'(q, c) = \delta(q, c)$  for all  $c \in \Sigma$ , if  $q \neq q_{m+1}$  and (v)  $\delta'(q_{m+1}, c) = \{q_m, \delta(q_m, c)\}$  for all  $c \in \Sigma$ .

Note that, according to (v), the forward factor oracle of the reverse pattern  $p$  is a non-deterministic finite automaton (NFA for short).

The simulation of the forward factor oracle can be done by simply changing the computation of the  $\lambda$  table, introduced in the Extended-BOM algorithm, in the following

way

$$\lambda(a, b) = \begin{cases} \delta(m, b) & \text{if } \delta(m, a) = \perp \vee b = p[m-1] \\ \delta(\delta(m, a), b) & \text{otherwise.} \end{cases}$$

In [Fan et al. 2009] a simplified version of the Forward-BOM algorithm (SFBOM) was presented along the same lines of the Simplified-Extended-BOM algorithm.

**4.2.2. Two Automata.** In this section we present a recent string matching algorithm, named Double-Forward-DAWG-Matching, based on the use of two automata. Actually it divides the window into two parts and scan each of them with the factor automaton of  $p$ .

*The Double-Forward-DAWG-Matching.* In [Allauzen and Raffinot 2000] the authors presented an efficient automata based algorithm, called Double-Forward-DAWG-Matching (DFDM), which makes use of two FDM procedures. The algorithm is linear in the worst case and optimal in the average case under a model of independence and equiprobability of letters.

Specifically the algorithm moves a window of size  $m$  along the text, and searches the pattern  $p$  in that window. It uses only the factor automaton of  $p$ .

For each text window,  $t[s..s+m-1]$ , two positions,  $\alpha$  and  $\beta$ , are defined such that  $s < \alpha < \beta < s+m-1$ . In order to obtain optimal average performances the authors suggest to take  $\alpha$  such that the distance  $d_\alpha$  from the beginning of the search window to  $\alpha$  is at most  $\lfloor m/2 \rfloor$ , and to take  $\beta$  such that the distance  $d_\beta$  from  $\beta$  to the end of the search window is  $d_\beta = \max\{1, \min\{m/5, 3\lceil \log_\sigma(m) \rceil\}\}$ .

During each attempt the algorithm reads forward the text in the current window with the factor automaton, starting at position  $\beta$ . Moreover it maintains a value  $\ell$  which is the length of the prefix of the current window recognized as a factor of  $p$ . At the beginning of the algorithm the value of  $\ell$  is set to 0.

If the automaton fails on a letter at position  $j < s+m-1$ , before reaching the end of the window, then  $t[\beta..j]$  is not a factor of the searched pattern  $p$ . In this case the algorithm performs a failure transition and continues to scan text characters with the FDM until it passes the right end of the window. This forward search stops once the computed value  $\ell$  is less than  $\alpha$ . After that, the new window is aligned with the factor of length  $\ell$  and the algorithm begins another forward search starting at position  $\beta$  of the new window.

Otherwise, if the algorithm reaches the right end of the window without doing any failure transition, then the part of the window located after  $\beta$  is a factor of  $p$ . In this case the algorithm resumes the reading with the previous FDM, that had stopped in  $s+\ell$ , in order to read again all the characters up to the right end of the window. Once the right end is passed, as in the previous case, the algorithm stops this FDM once the computed value  $\ell$  is less than  $\alpha$ . The occurrences are marked during the reading with this last FDM.

**4.2.3. Large Window.** In this section we present recent string matching algorithms based on automata that use a large window of length  $2m-1$ . Specifically we present the following algorithms:

- the Wide Window algorithm, which scans the right part of the window from left to right and the left part of the window from right to left;
- the family of Linear DAWG Matching algorithms, that scan the right part of the window from right to left and the left part of the window from left to right.

*The Wide Window Algorithm.* The Wide Window algorithm [He et al. 2005] (WW for short) considers  $n/m$  consecutive windows of length  $2m-1$ . Each attempt is divided

into two steps. The first step consists in scanning the  $m$  rightmost characters of the window from left to right with  $Dawg(p)$  starting with the initial state until a full match or a lack of transition. Doing so, it computes the length  $\ell$  of the longest suffix of  $p$  in this right part of the window. The second phase consists, if  $\ell > 0$ , in scanning the  $m - 1$  leftmost characters of the window from right to left with  $SMA(\bar{p})$  starting with state  $q_\ell$  until the length of the remembered suffix of  $p$ , given by  $\ell$ , is too small for finding an occurrence of  $p$ . Occurrences are only reported during the second phase.

The Wide Window algorithm inspects  $\mathcal{O}(n)$  text characters in the worst case and  $\mathcal{O}(n \log m/m)$  in the average case. Moreover the authors also designed a bit-parallel version referred to as Bit-Parallel Wide-Window algorithm (see Section 4.3.3).

*The Linear DAWG Matching Algorithm.* The Linear DAWG Matching algorithm [He et al. 2005] (LDM for short) considers  $n/m$  consecutive windows of length  $2m - 1$ . Each attempt is divided into two steps. The first step consists in scanning the  $m$  leftmost characters of the window from right to left with  $Dawg(\bar{p})$  starting with the initial state until a full match or a lack of transition. Doing so, it computes the length  $R$  of the longest prefix of  $p$  in this left part of the window. The second phase consists, if  $R > 0$ , in scanning the  $m$  rightmost characters of the window from left to right with  $SMA(p)$  starting with state  $q_\ell$  until the length of the remembered prefix of  $p$ , given by  $R$ , is too small for finding an occurrence of  $p$ . Occurrences are only reported during the second phase.

The Linear DAWG Matching Window algorithm inspects  $\mathcal{O}(n)$  text characters in the worst case and  $\mathcal{O}(n \log m/m)$  in the average case.

In [Liu et al. 2006] the authors propose two improvements of the LDM algorithm by performing the second phase of the algorithm while the length of the remembered prefix (stored in  $R$ ) is larger than a given value: 0 for the first improvement and  $m/2$  for the second one. Actually this could be given as a parameter to the algorithm.

As in the case of the WW algorithm the LDM could use the Shift-Or algorithm instead of  $SMA(p)$  for short patterns.

### 4.3. Bit-Parallel Algorithms

We now present the most recent string matching algorithms based on bit-parallelism. Section 4.3.1 presents variants of the Backward-Nondeterministic-Dawg-Matching algorithm. Section 4.3.2 presents algorithms using two automata. Section 4.3.3 presents algorithms using a large window. Section 4.3.4 presents an algorithm that is a variant of the Boyer-Moore algorithm. Section 4.3.5 presents algorithms that partition the pattern. Section 4.3.6 presents algorithms that use hashing or  $q$ -grams.

*4.3.1. Variants of the BNNDM Algorithm.* In this section we present recent string matching algorithms based on bit-parallelism that are variants of the Backward-Nondeterministic-Dawg-Matching algorithm. In particular:

- the Simplified BNNDM algorithm, which not memorize the longest prefix of the pattern during the scan of the window;
- the BNNDM2 and the Simplified BNNDM2 algorithms, that process the two rightmost characters of the window in one step;
- the BNNDM-BMH and BMH-BNNDM algorithms, hybrid solutions which alternately use the strategy of the BNNDM and Horspool algorithms.
- the Forward Simplified BNNDM algorithm, a bit parallel version of the Forward BOM algorithm (see Section 4.2.1).

*Fast Variants of the BNNDM Algorithm.* A simplified version of the classical BNNDM algorithm (SBNNDM for short) has been presented in [Peltola and Tarhio 2003]. Independently, Navarro has adopted a similar approach earlier in the code of his NR-

grep [Navarro 2001]. 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  $per(p)$  positions to the right. The value of  $per(p)$  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 [Holub and Durian 2005]. 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 [Holub and Durian 2005] the authors presented also two improvements of the BNDM algorithm by combining it with the Horspool algorithm (see Section 4.1) 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 defined in (4). 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  $d : \Sigma \rightarrow \{m + 1, \dots, 2m\}$  is defined as  $d(c) = m + hbc_p(c)$ , for  $c \in \Sigma$ . If  $D$  is 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  $\mathcal{O}(n \lceil m/\omega \rceil)$ -time and  $\mathcal{O}(\sigma \lceil m/\omega \rceil)$ -space complexity of the original algorithm.

*Forward SBNDM Algorithm.* The Forward-SBNDM algorithm [Faro and Lecroq 2008] (FSBNDM for short) is the bit-parallel version of the Forward-BOM algorithm described in Section 4.2.1.

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* 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  $\omega$  is the size 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  $j = s + m - 1$  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  $\mathcal{O}(n \lceil m/\omega \rceil)$ -time complexity as the BNDM algorithm but turns out to be faster in practical cases, especially for small alphabets and short patterns.

**4.3.2. Two Automata.** In this section we present a recent string matching algorithm, named Two-Way-Non-deterministic-DAWG-Matching, based on the bit-parallel technique that makes use of two automata for searching all occurrences of the pattern  $p$ . In particular it alternately simulates the nondeterministic factor automaton of  $p$  and the nondeterministic factor automaton of  $\bar{p}$ .

*Two-Way-Non-deterministic-DAWG-Matching Algorithm.* The recent Two-Way-Non-deterministic-DAWG-Matching algorithm (TNDM for short) was introduced in [Peltola and Tarhio 2003]. 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  $t[j - m + 1 \dots j]$ , 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  $t[j \dots k]$  does not occur in  $p$  or  $t[j \dots k]$  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 [Peltola and Tarhio 2003] 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 [Holub and Durian 2005] 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 TNNDM algorithm.

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

**4.3.3. Large Window.** In this section we present the following recent string matching algorithms based on bit-parallelism that uses a large window:

- the Bit Parallel Wide Window algorithm, a bit-parallel version of the Wide Window algorithm (Section 4.2.3);
- the Bit Parallel<sup>2</sup> Wide Window algorithm, which uses simultaneously two copies of the same automaton;
- the Bit Parallel Wide Window<sup>2</sup> algorithm, which uses simultaneously two different automata;
- the Bit Parallel algorithm for small alphabets, which uses a window of size  $\max\{m + 1, \omega\}$ .

*Bit Parallel Wide Window Algorithm.* The Bit Parallel Wide Window algorithm [He et al. 2005] (BPWW for short) is the bit parallel version of the Wide-Window algorithm described in Section 4.2.3.

The BPWW algorithm divides the text in  $\lceil n/m \rceil$  consecutive windows of length  $2m - 1$ . Each search attempt, on the text window  $t[j - m + 1 \dots j + m - 1]$  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  $t[j \dots j + m - 1]$ , from left to right, using the automaton  $NDawg(p)$ , 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  $\ell$  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  $t[j - m + 1 \dots j - 1]$ , from right to left using the  $NSMA(\bar{p})$  and starting with state  $\ell$  of the automaton. This is done until the length of the remembered suffix of  $p$ , given by  $\ell$ , 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  $\mathcal{O}(\sigma\lceil m/\omega \rceil)$  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.

*Bit-(Parallelism)<sup>2</sup> Algorithms for Short Patterns.* *Bit-parallelism<sup>2</sup>* is a technique recently introduced in [Cantone et al. 2010a] 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 (see Section 4.3.3), but their approaches can be applied as well to other (more efficient) solutions based on bit-parallelism.

In both variants of the BPWW algorithm, a word of  $\omega$  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

$$\begin{aligned}\mathcal{S}_j &= \{0 \leq i < m \mid p[i..m-1] = t[j..j+m-1-i]\}; \\ \mathcal{P}_j &= \{0 \leq i < m \mid p[0..i] = t[j-i..j]\}.\end{aligned}$$

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  $k = 0, \dots, m - 1$ , 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  $\ell = 0, 1, \dots, m - 1$ ), the bit  $SV[m - 1 - \ell]$  (resp., the bit  $PV[\ell]$ ) is set by one of the following bitwise operations:

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

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  $\omega$  is the size of the computer word in bits (see [Arndt 2009]). 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 [Arndt 2009]).

In the first variant (based on the first approach), named Bit-Parallel Wide-Window<sup>2</sup> (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

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

and transitions are performed in parallel with the following bitwise operations

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

for  $\ell = 1, \dots, m-1$  (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<sup>2</sup> 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<sup>1</sup> of both automata in parallel:

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

for  $\ell = 1, \dots, m-1$ . 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

$$M \leftarrow (1 \ll (m+k-1)) \quad | \quad (1 \ll (m-1)).$$

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

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

$$PS \leftarrow PS \quad | \quad ((D \& M) \gg \ell).$$

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

$$PV \leftarrow \text{reverse}(PS) \gg (\omega - m), \quad SV \leftarrow PS \gg k,$$

In fact, to obtain the correct value of  $PV$  we used bit-reversal modulo  $m$ , which has been easily achieved by right shifting  $\text{reverse}(PS)$  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.

*A bit-parallel algorithm for small alphabets.* The bit-parallel algorithm for small alphabets (SABP for short) [Zhang et al. 2009] 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  $\omega$  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  $\omega$  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  $c \in \Sigma$ .

<sup>1</sup>For  $\ell = 0$ ,  $D$  is simply updated by  $D \leftarrow D \quad \& \quad ((B[t[j+\ell]] \ll k) \mid C[t[j-\ell]])$ .

— an array  $T'$  of  $\sigma$  vectors of  $\omega$  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  $qbc_p$  according to equation 5.

Then during the searching phase the algorithm maintains a vector  $F$  of  $\omega$  bits such that when the window of size  $\ell$  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  $k = m-3, \dots, 0$  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  $\omega$  and the position of the rightmost 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.

**4.3.4. Variant of the Boyer-Moore Algorithm.** In this section we present a recent string matching algorithm based on bit-parallelism that uses a large window. The SVM algorithm memorizes information for all the matched characters of the window.

**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 [Peltola and Tarhio 2003] 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 rightmost 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  $\ell$  positions to the right, where  $\ell$  is the number of ones which precede the rightmost zero in the shift-vector  $S$ . In addition the SVM algorithm also shifts the shift-vector of  $\ell$  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  $\ell$  is efficiently computed by using the bitwise operations  $\ell = \text{bsf}(\sim(S \gg 1)) + 1$ , where we recall that the  $\text{bsf}$  function returns the number of zero bits before the leftmost bit set to 1.

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

**4.3.5. Partition the Pattern.** In this section we present the following recent string matching algorithms based on bit-parallelism that partition the pattern:

- the Average Optimal Shift-Or and Fast Average Optimal Shift-Or algorithms, which create several subpatterns consisting of non-adjacent characters of the pattern then they merge these subpatterns and search for them in the text detecting thus candidate positions.
- the Bit-Parallel Length Invariant Matcher, which overcomes the limitation of bit-parallel techniques on the pattern length  $m$  with respect to the size of the computer word  $\omega$  by creating  $\omega$  shifted copies of the patterns.
- the Long Pattern BNDM algorithm, which considers substring of the pattern of length  $\lfloor (m-1)/\omega \rfloor + 1$ .
- the Factorized Shift-And and the Factorized BNDM algorithms, based on a 1-factorization of the pattern.

*Average Optimal Shift Or Algorithms.* Fredriksson and Grabowski presented in [Fredriksson and Grabowski 2005] 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,  $\mathcal{P} = \{p^0, p^1, \dots, p^{q-1}\}$ , where each  $p^j$  has length  $m' = \lfloor m/q \rfloor$  and is defined as

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

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  $p' = p^0 p^1 \dots p^{q-1}$ . Moreover the algorithm initializes a mask  $M$  which has the bits of position  $(j+1)m'$  set to 1, for all  $j = 0, \dots, q-1$ .

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 operations

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

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.

*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 [Külekci 2008] 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  $\omega$  rows, where  $\omega$  is the size of a word in the target machine. Each row  $row_i$ , for  $0 \leq i < \omega$ , contains the pattern right shifted by  $i$  characters. Thus,  $A$  contains  $wsiz = \omega + m - 1$  columns. During the preprocessing phase the algorithm computes a mask matrix  $M$  of size  $\sigma \times wsiz$ , where  $M[c, h] = b_{\omega-1} \dots b_1 b_0$  is a bit vector of  $\omega$  bits, for  $c \in \Sigma$  and  $0 \leq h < wsiz$ . 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  $wsiz$ , and at each attempt check if any possible placements of the input pattern exist.

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  $m - i, 2m - i, \dots, km - i$ , where  $km - i < wsiz$ , for  $i = 1, 2, \dots, m$  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  $wsiz$ .

When the window is located at  $t[i \dots i + wsiz - 1]$ , 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  $j = 1, \dots, wsiz$ , 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 (see Section 4.1, equation (5)). The immediate text character following the window determines the shift amount. If that character is included

in the pattern then the shift amount is  $wsiz - k$ , where  $k = \max\{i \mid p[i] = t[s + wsiz]\}$ , otherwise the shift value is equal to  $wsiz + 1$ .

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

*Bit-Parallel Algorithms for Long Patterns.* In [Navarro and Raffinot 2000] the authors introduced an efficient method, based on bit-parallelism, to search for patterns longer than  $\omega$ . 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  $\omega$ , which could be much smaller than  $m$ .

Later in [Peltola and Tarhio 2003] 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  $\omega$ .

*Tighter packing for bit-parallelism.* In order to overcome the problem due to handling long patterns with bit-parallelism, in [Cantone et al. 2010a] 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  $\sigma$ ). 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  $k \leq m$  (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  $\langle u_1, u_2, \dots, u_k \rangle$  of nonempty substrings of  $p$  such that  $p = u_1 u_2 \dots u_k$  and each factor  $u_j$  contains at most one occurrence for any of the characters in the alphabet  $\Sigma$ , for  $j = 1, \dots, k$ . A 1-factorization of  $p$  is *minimal* if such is its size.

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

Observe, also, that  $\lceil \frac{m}{\sigma} \rceil \leq k \leq m$  holds, for any 1-factorization of size  $k$  of a string  $p \in \Sigma^m$ . 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  $\langle u_1, u_2, \dots, u_k \rangle$  of a given pattern  $p \in \Sigma^*$  induces naturally a partition  $\{Q_1, \dots, Q_k\}$  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  $i = 1, \dots, k-1$ , we put  $\bar{u}_i = u_i \cdot \text{first}(u_{i+1})$ . We also put  $\bar{u}_k = u_k$  and call each set  $\bar{u}_i$  the *closure* of  $u_i$ . Plainly, any 2-gram can occur at most once in the closure  $\bar{u}_i$  of any factor of our 1-factorization  $\langle u_1, u_2, \dots, u_k \rangle$  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  $\bar{u}_i$  or, equivalently, iff

$$\begin{aligned} & (\text{last}(u_i) \neq c_1 \wedge q_{i,c_2} \in \delta(q_{i,c_1}, c_2)) \vee \\ & (i < k \wedge \text{last}(u_i) = c_1 \wedge q_{i+1,c_2} \in \delta(q_{i,c_1}, c_2)), \end{aligned} \quad (7)$$

for every 2-gram  $c_1c_2 \in \Sigma^2$  and  $i = 1, \dots, k$ .

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  $\text{last}(u_i) = c$ , for  $i = 1, \dots, k$ .

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:

$$(i) \ D \leftarrow D \& B[a, c] \quad (ii) \ H \leftarrow D \& L[a] \quad (iii) \ D \leftarrow (D \& \sim H) \mid (H \ll 1).$$

To check whether the final state  $q_m$  belongs to a configuration encoded as  $(D, a)$ , one has 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 by  $D \& M \& L[a] \neq 0^k$ , 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  $\varepsilon$ -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.

**4.3.6. Hashing and  $q$ -grams.** In this section we present recent string matching algorithms based on bit-parallelism that use  $q$ -grams. The  $\text{BNDM}_q$  and  $\text{UFNDM}_q$  algorithm process the  $q$  rightmost characters of the window in one step.

*Bit-Parallel Algorithms with  $q$ -grams.* The idea of using  $q$ -grams for shifting (see Section 4.1.3 and Section 4.1.3) was applied also to bit parallel algorithms in [Durian et al. 2009].

First, the authors presented a variation of the BNDM algorithm called  $\text{BNDM}_q$ . Specifically, at each alignment of the pattern with the current window of the text ending at position  $j$ , the  $\text{BNDM}_q$  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  $\text{BNDM}_q$  algorithm (called  $\text{SBNDM}_q$ ) along the same line of the SBNDM algorithm [Peltola and Tarhio 2003; Navarro 2001] (see Section 4.3.1).

Finally the authors presented also an efficient  $q$ -grams variant of the Forward-Non-deterministic-DAWG-Matching algorithm [Holub and Durian 2005]. The resulting algorithm is called  $\text{UFNDM}_q$ , 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  $\text{BNDM}_q$ , the  $\text{UFNDM}_q$  algorithm reads  $q$  characters before testing the state vector  $D$ . Formally the state vector  $D$  is initialized as follows:

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

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

Finally we notice that a similar approach adopted in [Kalsi et al. 2008] can be used for  $\text{BNDM}_q$  and  $\text{SBNDM}_q$  (see Section 4.1.3). In particular in [Durian et al. 2009] the authors developed three versions for both  $\text{BNDM}_q$  and  $\text{SBNDM}_q$ .

#### 4.4. Constant-space Algorithms

We now present the most recent constant-space string matching algorithm. Section 4.4.1 presents an algorithm that uses a partition of the pattern.

**4.4.1. Partition the Pattern.** In this section we present a recent constant-space string matching algorithm that partitions the pattern. The Tailed-Substring algorithm uses the longest suffix of  $p$  that contains only one occurrence of the last character of  $p$ .

*Tailed-Substring Algorithm.* In [Cantone and Faro 2004] the authors proposed a constant-space algorithm for the string-matching problem which, though quadratic in the worst case, is very efficient in practice.

The proposed algorithm, called Tailed-Substring, performs its preprocessing in parallel with the searching phase. The idea is based on the following notion of maximal



tailed-substring of  $p$ . We say that a substring  $r$  of  $p$  is a *tailed-substring* if its last character is not repeated elsewhere in  $r$ . Then a *maximal tailed-substring* of  $p$  is a tailed-substring of  $p$  of maximal length. In the following, given a maximal tailed-substring  $r$  of a pattern  $p$ , we associate to  $r$  its length  $\alpha$  and a natural number  $k$  (with  $\alpha - 1 \leq k < m$ ) such that  $r = p[k - \alpha + 1 .. k]$ .

The Tailed-Substring algorithm searches for a pattern  $p$  in a text  $t$  in two subsequent phases. During the first phase, while it searches for occurrences of  $p$  in  $t$ , the Tailed-Substring algorithm also computes values of  $\alpha$  and  $k$  such that  $r = p[k - \alpha + 1 .. k]$  is a maximal tailed-substring of  $p$ . During the second phase, it just uses the values for  $\alpha$  and  $k$  computed in the first phase to speed up the search of the remaining occurrences of  $p$  in  $t$ .

The first searching phase works as follows. Initially, the value of  $\alpha$  is set to 1 and the values of  $i$  and  $k$  are set to  $m - 1$ . Next, the following steps are repeated until  $\alpha \geq i$ . The first value of the shift  $s$  such that  $p[i] = t[s + i]$  is looked for and then it is checked whether  $p = t[s .. s + m - 1]$ , proceeding from left to right. At this point, the rightmost occurrence  $h$  of  $p[i]$  in  $p[0 .. i - 1]$  is searched for. If such an occurrence is found (i.e.,  $h \geq 0$ ), the algorithm aligns it with character  $s + i$  of the text; otherwise, the shift is advanced by  $i + 1$  positions (in this case  $h = -1$ ). Then, if the condition  $i - h \geq \alpha$  holds,  $\alpha$  is set to  $i - h$ ,  $k$  is set to  $i$ , and the value of  $i$  is decreased by 1. It can be shown that at the end of the first searching phase  $p[k - \alpha + 1 .. k]$  is a maximal tailed-substring of  $p$ .

In the second searching phase the algorithm looks for an occurrence of character  $p[k]$  in the text. When a value  $s$  such that  $p[k] = t[s + k]$  is found, it is checked whether  $p = t[s .. s + m - 1]$ , proceeding from left to right, and then the shift is advanced by  $\alpha$  positions to the right. The preceding steps are repeated until all occurrences of  $p$  in  $t$  have been found.

The resulting algorithm runs in  $\mathcal{O}(nm)$  worst-case time complexity.

## 5. EXPERIMENTAL EVALUATION

We present next experimental data which allow to compare in terms of running time the best performance of all string matching algorithms, included those reviewed in this paper.

All algorithms have been implemented in the **C** programming language and were used to search for the same strings in large fixed text buffers on a PC with Intel Core2 processor of 1.66GHz with 2GB of RAM and 2MB of L2-cache. Running times have been measured with a hardware cycle counter, available on modern CPUs. The codes have been compiled with the GNU C Compiler, using the optimization option `-O2`.

In particular, the algorithms have been tested on the following text buffers:

- (i) eight  $\text{Rand}\sigma$  text buffers, for  $\sigma = 2, 4, 8, 16, 32, 64, 128$  and  $256$ , where each  $\text{Rand}\sigma$  text buffer consists in a 5MB random text over a common alphabet of size  $\sigma$ , with a uniform distribution of characters;
- (ii) the English King James version of the Bible composed of 4,047,392 characters (with  $\sigma = 63$ );
- (iii) the file `world192.txt` (The CIA World Fact Book) composed of 2,473,400 characters (with  $\sigma = 94$ );
- (iv) a genome sequence of 4,638,690 base pairs of *Escherichia coli* (with  $\sigma = 4$ );
- (v) a protein sequence (the `hs` file) from the *Saccharomyces cerevisiae* genome, of length 3,295,751 bytes (with  $\sigma = 20$ );

Files (ii), (iii) and (iv) are from the Large Canterbury Corpus (<http://www.data-compression.info/Corpora/CanterburyCorpus/>), while file (v) is from the Protein Corpus (<http://data-compression.info/Corpora/ProteinCorpus/>).

For each input file, we have generated sets of 400 patterns of fixed length  $m$  randomly extracted from the text, for  $m$  ranging over the values 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024. For each set of patterns we reported in a table the mean over the running times of the 400 runs. Because of space limitation we report only results of the classical BOM and BNDM algorithms and of algorithms which appear, at least once, in the first three best results. The comprehensive set of experimental evaluation of all string matching algorithms can be found in [Faro and Lecroq 2010].

In addition to the algorithms reviewed in this article the following tables report running times also for the following algorithms:

- KR: the Karp-Rabin algorithm [Karp and Rabin 1987];
- AUT: the Finite-Sate Automaton matcher;
- RCol: the Reverse Colussi algorithm [Colussi 1994];
- ZT: the Zhu-Takaoka algorithm [Zhu and Takaoka 1987].

The set of C codes of all the algorithms which we use in our tests can be found at the web page <http://www.dmi.unict.it/~faro/smart/>.

In the following tables running times are expressed in thousands of seconds. The best results are boldfaced.

#### Experimental Results on Rand2 Problem

$m$	2	4	8	16	32	64	128	256	512	1024
KR	48.17	24.73	16.94	16.36	16.35	16.39	16.38	16.38	16.40	16.39
TVSBS	29.81	34.33	36.90	36.06	34.92	36.56	35.60	36.20	36.27	35.48
HASH3	-	28.24	13.98	9.78	8.64	8.71	8.88	8.71	8.72	8.65
HASH5	-	-	14.44	<b>6.05</b>	3.72	3.07	3.15	3.12	3.12	3.12
HASH8	-	-	-	7.67	<b>3.47</b>	<b>2.47</b>	<b>2.87</b>	1.97	1.44	1.30
SSEF	-	-	-	-	5.38	3.38	3.44	<b>1.79</b>	<b>0.99</b>	<b>0.55</b>
AUT	21.68	21.66	21.70	21.68	21.71	21.77	21.78	21.91	22.60	23.85
BOM	94.08	74.32	47.42	28.89	17.14	9.94	7.52	4.14	2.27	1.27
EBOM	41.11	37.17	25.80	14.44	8.06	4.77	4.61	2.79	1.99	2.92
SO	16.45	16.45	16.44	16.42	16.44	21.74	21.73	21.76	21.75	21.74
SA	<b>16.41</b>	<b>16.40</b>	16.41	16.40	16.40	19.13	19.14	19.13	19.13	19.15
AOSO2	167.2	36.72	<b>11.49</b>	9.66	8.54	8.53	8.55	8.54	8.55	8.55
BNDM	63.5	47.9	25.6	12.6	6.48	8.53	8.52	8.52	8.53	8.50
BNDMq4	-	52.99	18.45	9.49	5.10	6.51	6.49	6.50	6.51	6.49
BNDMq6	-	-	26.32	9.13	5.08	5.11	5.13	5.12	5.14	5.12
SBNDMq8	-	-	97.01	11.94	5.02	4.63	4.63	4.65	4.66	4.64

#### Experimental Results on Rand4 Problem

$m$	2	4	8	16	32	64	128	256	512	1024
TVSBS	22.25	17.43	12.08	8.67	6.90	6.40	6.40	6.30	6.29	6.37
HASH3	-	21.12	<b>8.30</b>	4.74	3.42	3.07	3.13	3.09	3.10	3.08
HASH5	-	-	12.53	5.01	2.93	2.48	2.85	2.49	2.24	2.19
HASH8	-	-	-	7.62	3.45	<b>2.46</b>	2.85	1.96	1.45	1.30
SSEF	-	-	-	-	5.39	3.36	3.43	<b>1.79</b>	<b>0.99</b>	<b>0.54</b>
AUT	21.68	21.68	21.69	21.71	21.70	21.74	21.79	21.91	22.61	23.90
BOM	65.72	44.01	27.71	17.62	11.23	6.84	5.79	3.20	1.78	1.03
EBOM	23.98	<b>14.17</b>	10.18	6.94	4.43	3.06	3.55	2.16	1.61	2.75
SEBOM	24.81	14.34	10.34	7.08	4.58	3.23	3.71	2.27	1.67	2.79
SO	16.45	16.45	16.43	16.42	16.44	21.75	21.75	21.74	21.74	21.75
SA	<b>16.40</b>	16.39	16.41	16.40	16.39	19.14	19.13	19.13	19.15	19.13
BNDM	49.0	27.8	14.7	7.91	4.40	5.85	5.88	5.86	5.87	5.86
FSBNDM	39.73	21.06	11.37	6.23	3.36	3.38	3.37	3.37	3.37	3.37
BNDMq4	-	48.66	10.79	4.89	<b>2.86</b>	3.53	3.54	3.54	3.54	3.53
SBNDMq4	-	45.99	10.22	<b>4.72</b>	2.87	2.68	<b>2.68</b>	2.69	2.68	2.69

## Experimental Results on Rand8 Problem

$m$	2	4	8	16	32	64	128	256	512	1024
BR	16.75	11.60	7.47	4.75	3.24	2.76	3.39	2.97	2.93	2.95
SSABS	16.54	11.63	8.34	6.74	6.33	6.32	6.28	6.29	6.26	6.17
TVSBS	<b>14.92</b>	10.53	6.91	4.50	3.23	2.82	3.17	2.96	2.95	2.92
HASH3	-	19.07	7.25	3.88	2.66	2.46	2.75	2.60	2.45	2.38
HASH5	-	-	12.17	4.79	2.71	<b>2.41</b>	2.78	2.06	1.64	1.47
HASH8	-	-	-	7.61	3.45	2.46	2.85	1.96	1.45	1.30
SSEF	-	-	-	-	5.39	3.36	3.43	<b>1.79</b>	<b>1.00</b>	<b>0.55</b>
BOM	48.58	33.30	22.19	15.08	9.60	5.98	5.11	2.82	1.60	0.94
EBOM	19.62	<b>8.37</b>	<b>5.04</b>	<b>3.70</b>	3.00	2.63	3.13	1.90	1.48	2.65
FBOM	17.39	10.38	6.72	4.63	3.45	2.83	3.30	2.01	1.52	2.69
SEBOM	20.63	8.73	5.22	3.82	3.12	2.76	3.25	2.02	1.56	2.72
SFBOM	17.23	10.44	6.77	4.68	3.49	2.88	3.33	2.05	1.56	2.72
SA	16.40	16.40	16.40	16.41	16.41	18.89	18.89	18.89	18.89	18.89
BNDM	37.3	22.0	11.6	6.10	3.66	4.51	4.51	4.52	4.52	4.51
FSBNDM	28.12	14.25	7.85	4.71	2.74	2.75	2.74	2.74	2.74	2.76
BNDMq2	33.80	12.85	6.58	4.06	2.84	3.41	3.44	3.45	3.45	3.44
BNDMq4	-	48.40	10.44	4.59	2.57	3.16	3.15	3.17	3.16	3.16
SBNDMq2	33.46	12.74	6.72	4.25	2.97	2.79	2.79	2.79	2.81	2.82
SBNDMq4	-	45.76	9.90	4.39	<b>2.56</b>	2.46	<b>2.46</b>	2.46	2.45	2.46

## Experimental Results on Rand16 Problem

$m$	2	4	8	16	32	64	128	256	512	1024
RCol	15.27	8.47	5.08	3.48	2.88	2.77	2.80	2.77	2.72	2.63
FS	15.26	8.47	5.08	3.47	2.86	2.77	2.80	2.78	2.72	2.65
BFS	15.36	8.51	5.08	3.43	2.79	2.65	2.89	2.77	2.56	2.65
SSABS	12.35	8.25	5.51	3.94	3.28	3.12	3.14	3.16	3.15	3.16
TVSBS	<b>11.56</b>	8.09	5.33	3.51	2.74	2.55	2.87	1.96	1.52	1.40
FJS	12.69	8.58	5.75	4.16	3.44	3.26	3.31	3.30	3.28	3.30
HASH3	-	18.34	6.85	3.58	2.49	<b>2.33</b>	2.69	2.31	2.07	1.96
HASH5	-	-	12.07	4.72	2.65	2.39	2.74	1.84	1.37	1.22
HASH8	-	-	-	7.59	3.45	2.46	2.85	1.96	1.44	1.29
SSEF	-	-	-	-	5.38	3.37	3.44	<b>1.79</b>	<b>0.99</b>	<b>0.55</b>
BOM	40.88	28.99	22.62	15.61	9.65	5.86	4.89	2.75	1.55	0.97
EBOM	18.58	<b>7.15</b>	<b>3.88</b>	<b>2.81</b>	2.55	2.44	2.83	1.81	1.42	2.68
FBOM	13.27	8.17	5.10	3.41	2.79	2.66	3.20	1.88	1.45	2.69
SEBOM	19.64	7.57	4.11	2.94	2.68	2.56	2.95	1.93	1.49	2.74
BNDM	30.50	17.60	10.50	5.68	3.11	3.73	3.73	3.75	3.74	3.75
SBNDM2	35.17	12.52	6.10	3.45	2.55	2.44	2.45	2.45	2.45	2.44
BMH-SBNDM	15.48	8.42	5.00	3.35	2.75	2.84	2.84	2.85	2.81	2.84
FSBNDM	23.61	12.05	6.46	3.73	<b>2.38</b>	2.38	<b>2.38</b>	2.39	2.39	2.37
BNDMq2	33.28	11.80	5.61	3.16	2.48	2.68	2.67	2.67	2.68	2.70
SBNDMq2	32.57	11.67	5.70	3.35	2.50	2.45	2.44	2.44	2.44	2.44
SBNDMq4	-	45.72	9.88	4.36	2.54	2.44	2.45	2.45	2.44	2.44

## Experimental Results on Rand32 Problem

$m$	2	4	8	16	32	64	128	256	512	1024
ZT	29.83	15.41	8.29	4.86	3.08	2.55	2.87	1.67	1.07	0.77
RCol	13.77	7.43	4.27	2.76	2.43	2.33	2.53	2.52	2.48	2.47
BR	11.79	8.20	5.34	3.50	2.64	2.51	2.85	1.67	1.08	0.79
FS	13.77	7.44	4.27	2.76	2.43	2.33	2.53	2.51	2.49	2.47
BFS	13.83	7.47	4.29	2.75	2.44	2.35	2.65	2.61	2.49	2.58
SSABS	10.57	6.92	4.43	3.06	2.58	2.46	2.66	2.64	2.65	2.66
TVSBS	<b>10.19</b>	7.19	4.73	3.17	2.62	2.50	2.68	<b>1.58</b>	1.02	0.74
FJS	10.54	6.97	4.49	3.09	2.61	2.49	2.68	2.66	2.66	2.66
HASH3	-	18.06	6.68	3.45	2.45	2.30	2.63	1.92	1.55	1.38
SSEF	-	-	-	-	5.38	3.38	3.44	1.78	<b>1.00</b>	<b>0.54</b>
BOM	37.8	27.4	24.6	17.4	11.4	6.97	5.31	2.96	1.73	1.15
EBOM	18.33	<b>6.87</b>	<b>3.63</b>	<b>2.67</b>	2.49	2.41	2.72	1.71	1.38	2.69
FBOM	11.78	7.41	4.61	3.05	2.67	2.61	2.91	1.79	1.46	2.72
SEBOM	19.38	7.29	3.85	2.79	2.61	2.52	2.83	1.79	1.45	2.73
BNDM	27.43	15.07	8.84	5.38	3.12	3.71	3.72	3.72	3.72	3.73
SBNDM2	34.97	12.30	5.90	3.27	2.45	2.39	2.40	2.39	2.39	2.39
BMH-SBNDM	13.87	7.42	4.24	2.73	2.41	2.44	2.45	2.44	2.44	2.44
FSBNDM	21.92	11.23	5.97	3.43	<b>2.29</b>	<b>2.28</b>	<b>2.29</b>	2.31	2.29	2.30
SBNDMq2	32.35	11.44	5.46	3.12	2.42	2.38	2.37	2.39	2.39	2.39

## Experimental Results on Rand64 Problem

$m$	2	4	8	16	32	64	128	256	512	1024
ZT	29.43	15.39	8.42	5.01	3.25	2.61	2.86	1.61	0.97	0.61
TunBM	13.69	7.29	4.09	2.60	2.34	2.20	2.60	2.42	2.40	2.38
RCol	13.14	7.00	3.94	2.52	2.32	2.20	2.58	2.41	2.38	2.36
BR	11.37	8.02	5.32	3.60	2.78	2.60	2.82	1.59	0.96	0.63
FS	13.13	7.00	3.94	2.52	2.31	<b>2.19</b>	2.58	2.42	2.39	2.37
BFS	13.16	7.01	3.95	2.53	2.34	2.23	2.68	2.55	2.59	2.78
SSABS	9.75	6.38	4.02	2.69	2.43	2.29	2.66	2.48	2.44	2.44
TVSBS	9.83	7.04	4.75	3.30	2.73	2.56	2.67	<b>1.52</b>	<b>0.92</b>	0.59
FJS	<b>9.59</b>	<b>6.30</b>	3.98	2.69	2.44	2.31	2.66	2.48	2.43	2.44
GRASpm	14.57	7.72	4.31	2.63	2.36	2.21	2.64	2.41	2.26	2.05
SSEF	-	-	-	-	5.39	3.37	3.43	1.80	0.99	<b>0.55</b>
BOM	36.2	26.6	25.6	19.0	13.9	9.40	6.85	3.71	2.10	1.36
EBOM	18.38	6.92	<b>3.72</b>	2.77	2.59	2.51	2.76	1.73	1.41	2.68
SEBOM	19.43	7.32	3.94	2.88	2.70	2.62	2.87	1.83	1.48	2.74
BNDM	25.97	13.83	7.79	4.65	3.01	3.51	3.51	3.53	3.51	3.51
SBNDM2	34.91	12.26	5.85	3.23	2.42	2.38	2.38	2.38	2.38	2.38
SBNDM-BMH	12.36	6.90	4.08	2.83	2.41	2.52	2.52	2.51	2.52	2.52
BMH-SBNDM	13.17	6.98	3.92	<b>2.50</b>	2.30	2.34	2.35	2.36	2.36	2.35
FSBNDM	21.20	10.90	5.80	3.32	<b>2.27</b>	2.27	<b>2.27</b>	2.26	2.27	2.27
KBNDM	40.21	20.48	10.77	5.94	3.54	2.64	3.01	1.63	1.47	1.46

## Experimental Results on Rand128 Problem

$m$	2	4	8	16	32	64	128	256	512	1024
ZT	39.17	20.34	10.89	6.16	3.83	2.77	2.98	1.62	1.07	0.60
TunBM	13.10	6.99	3.85	2.48	2.29	2.14	2.58	2.03	2.06	1.84
RCol	12.83	6.82	3.80	2.46	2.27	2.13	2.58	2.03	2.04	1.83
BR	15.18	10.60	6.87	4.43	3.06	2.74	2.92	1.59	1.08	0.60
FS	12.81	6.82	3.79	2.46	2.29	<b>2.13</b>	2.57	2.02	2.04	1.84
FFS	12.83	6.83	3.81	2.47	2.32	2.17	2.66	2.20	2.44	2.49
TS	11.55	11.44	11.05	10.46	9.47	7.96	6.43	5.01	3.67	2.54
SSABS	9.34	6.14	3.81	2.54	2.39	2.25	2.61	2.07	2.08	1.86
TVSBS	13.64	9.60	6.30	4.15	2.95	2.66	2.77	<b>1.54</b>	<b>1.05</b>	0.59
FJS	<b>9.18</b>	<b>6.04</b>	3.76	2.54	2.38	2.25	2.61	2.04	2.10	1.86
HASH3	-	18.01	6.62	3.41	2.44	2.30	2.57	1.72	1.45	1.18
GRASpm	14.20	7.51	4.13	2.48	2.35	2.15	2.64	2.06	2.06	<b>1.80</b>
SSEF	-	-	-	-	5.40	3.38	3.43	1.79	1.14	<b>0.57</b>
BOM	35.5	26.4	26.6	20.3	16.0	12.2	9.55	5.49	3.45	1.92
BOM2	21.47	11.45	6.43	3.93	2.83	2.54	3.03	1.72	1.41	2.48
BNDM	25.26	13.32	7.23	4.16	2.73	3.12	3.12	3.48	3.56	3.21
SBNDM	48.07	16.59	7.46	3.94	2.42	2.36	2.36	2.66	2.70	2.43
SBNDM-BMH	11.82	6.55	<b>3.71</b>	2.58	2.29	2.36	2.36	2.67	2.69	2.43
BMH-SBNDM	12.83	6.86	3.78	<b>2.45</b>	2.28	2.34	2.33	2.61	2.67	2.40
FSBNDM	20.90	10.76	5.74	3.29	<b>2.26</b>	2.27	<b>2.25</b>	2.54	2.57	2.33

## Experimental Results on Rand256 Problem

$m$	2	4	8	16	32	64	128	256	512	1024
ZT	47.74	24.70	13.29	7.84	5.44	4.32	3.87	2.02	1.08	0.63
TunBM	13.23	6.98	3.87	2.51	2.34	2.17	2.60	1.73	1.30	1.16
RCol	13.08	6.91	3.84	2.53	2.32	2.18	2.58	1.73	1.31	1.18
BR	19.88	13.80	9.06	6.15	4.75	4.21	3.78	2.00	1.10	0.60
FS	13.08	6.90	3.84	2.50	2.35	<b>2.17</b>	2.59	1.72	1.29	1.18
FFS	13.08	6.92	3.84	2.54	2.36	2.23	2.68	1.91	1.69	1.91
BFS	13.09	6.90	3.84	2.52	2.35	2.23	2.69	1.94	1.69	1.92
TS	11.73	11.63	11.46	11.12	10.51	9.48	8.08	6.68	4.51	2.90
SSABS	9.47	6.19	3.85	2.57	2.43	2.29	2.61	1.75	1.31	1.18
TVSBS	18.35	12.77	8.46	5.88	4.64	4.15	3.70	1.98	1.08	0.60
FJS	<b>9.26</b>	<b>6.07</b>	3.76	2.55	2.44	2.29	2.62	1.74	1.32	1.18
GRASpm	14.47	7.60	4.18	<b>2.50</b>	2.38	2.19	2.64	1.75	1.32	1.16
SSEF	-	-	-	-	5.58	3.49	3.55	1.84	1.03	<b>0.55</b>
BNDM	25.75	13.37	7.21	4.03	2.65	2.94	2.92	2.92	2.94	2.93
LBNDM	30.43	15.69	8.33	4.66	2.87	2.46	2.91	<b>1.61</b>	0.96	0.66
SBNDM-BMH	11.93	6.50	<b>3.63</b>	2.55	<b>2.30</b>	2.35	2.35	2.34	2.36	2.36
BMH-SBNDM	13.09	6.90	3.83	2.51	2.32	2.39	2.38	2.38	2.39	2.40
AOSO2	11.63	10.04	10.05	10.04	10.05	8.80	8.79	8.80	8.79	8.80
AOSO4	-	6.81	5.27	5.25	5.26	4.69	4.69	4.70	4.70	4.70
KBNDM	46.62	23.68	12.48	7.16	4.75	3.61	3.42	1.78	<b>0.96</b>	0.87

## Experimental Results on a Natural Language Text (the Bible)

$m$	2	4	8	16	32	64	128	256	512	1024
ZT	26.41	13.76	7.52	4.44	2.87	2.26	2.55	1.77	1.27	0.98
BR	11.29	8.17	5.28	3.41	2.40	2.17	2.63	1.78	1.30	1.01
BFS	13.06	7.62	4.52	3.01	2.29	2.08	2.40	2.14	1.94	1.99
SSABS	10.80	7.17	4.88	3.43	2.66	2.33	2.52	2.25	1.91	1.65
TVSBS	<b>10.14</b>	7.16	4.60	3.12	2.35	2.15	2.42	1.71	1.21	0.94
FJS	11.07	7.53	5.05	3.53	2.77	2.45	2.62	2.32	1.98	1.69
HASH3	-	14.59	5.42	<b>2.79</b>	<b>1.97</b>	<b>1.84</b>	2.09	1.45	1.06	<b>0.85</b>
HASH5	-	-	9.68	3.81	2.14	1.90	2.21	1.49	1.07	0.89
HASH8	-	-	-	6.08	2.77	1.97	2.28	1.58	1.15	0.98
SSEF	-	-	-	-	4.36	2.59	2.72	<b>1.44</b>	<b>0.80</b>	0.45
BOM	34.6	25.2	19.9	13.9	9.36	6.03	4.90	2.90	1.72	1.10
EBOM	15.30	<b>6.53</b>	<b>3.87</b>	2.91	2.47	2.21	2.55	1.68	1.41	2.67
SEBOM	16.21	6.84	4.10	3.07	2.61	2.33	2.67	1.80	1.48	2.71
BNDM	25.60	15.45	9.09	5.11	3.01	3.65	3.64	3.65	3.64	3.65
BMH-SBNDM	13.25	7.41	4.46	2.94	2.28	2.31	2.31	2.31	2.30	2.29
FSBNDM	19.79	10.35	5.74	3.56	2.20	2.18	2.20	2.18	2.18	2.19
BNDMq4	-	38.75	8.48	3.79	2.16	2.66	2.67	2.68	2.67	2.67
SBNDMq4	-	36.54	8.02	3.61	2.14	2.04	<b>2.05</b>	2.05	2.05	2.05

## Experimental Results on a Natural Language Text (World192)

$m$	2	4	8	16	32	64	128	256	512	1024
ZT	15.61	8.13	4.46	2.68	1.77	1.44	1.60	1.08	0.76	0.56
Raita	8.39	4.57	2.73	1.75	1.36	1.22	1.46	1.18	0.91	0.75
RCol	7.47	4.10	2.50	1.62	1.31	1.23	1.40	1.12	0.87	0.74
BR	6.60	4.69	3.13	2.09	1.55	1.42	1.65	1.10	0.78	0.61
FS	7.39	4.10	2.48	1.64	1.32	1.20	1.40	1.13	0.89	0.71
FFS	7.44	4.26	2.50	1.64	1.33	1.25	1.45	1.28	1.19	1.39
BFS	7.43	4.18	2.48	1.63	1.32	1.23	1.46	1.29	1.23	1.46
SSABS	6.04	4.13	2.65	1.83	1.44	1.28	1.46	1.18	0.93	0.77
TVSBS	<b>5.91</b>	4.21	2.88	1.98	1.54	1.43	1.56	1.05	0.74	<b>0.54</b>
FJS	6.07	3.98	2.70	1.86	1.48	1.32	1.51	1.20	0.96	0.82
HASH3	-	8.85	3.31	1.72	<b>1.22</b>	<b>1.15</b>	1.29	<b>0.88</b>	<b>0.63</b>	0.50
HASH5	-	-	5.94	2.32	1.32	1.17	1.36	0.93	0.68	0.57
HASH8	-	-	-	3.74	1.70	1.23	1.43	0.99	0.72	0.63
SSEF	-	-	-	-	2.74	1.69	1.73	0.93	0.48	0.29
EBOM	9.36	<b>3.84</b>	<b>2.20</b>	1.64	1.48	1.44	1.62	1.18	1.11	2.49
SEBOM	9.90	4.10	2.35	1.80	1.62	1.56	1.75	1.28	1.19	2.54
BNDM	14.67	8.33	4.96	2.89	1.73	2.07	2.08	2.07	2.08	2.06
SBNDM2	17.18	6.27	3.07	1.81	1.35	1.29	1.27	1.29	1.30	1.28
BMH-SBNDM	7.60	4.18	2.41	<b>1.58</b>	1.28	1.30	1.30	1.29	1.32	1.32
FSBNDM	11.39	6.04	3.27	1.95	1.26	1.25	<b>1.23</b>	1.24	1.24	1.25
SBNDMq2	15.89	5.84	2.92	1.75	1.34	1.29	1.27	1.30	1.28	1.29

## Experimental Results on a Genome Sequence

$m$	2	4	8	16	32	64	128	256	512	1024
TVSBS	44.63	35.28	24.65	17.49	13.85	12.71	12.58	12.55	12.85	12.48
HASH3	-	39.75	<b>14.97</b>	<b>7.93</b>	<b>5.38</b>	4.96	5.59	5.22	5.06	5.03
HASH5	-	-	25.14	9.84	5.54	<b>4.91</b>	5.66	4.00	3.04	2.58
HASH8	-	-	-	15.78	7.18	5.09	5.86	4.19	3.13	2.70
SSEF	-	-	-	-	11.47	6.22	6.68	<b>3.75</b>	<b>2.26</b>	<b>1.59</b>
AUT	44.75	46.04	44.75	46.06	46.05	44.78	46.15	44.96	45.68	47.06
BOM	136.3	90.25	56.48	36.57	23.00	14.05	11.85	6.52	3.59	2.05
EBOM	49.77	<b>29.06</b>	21.30	14.35	9.03	6.11	7.03	4.08	2.74	3.42
SEBOM	51.36	29.43	21.41	14.49	9.14	6.32	7.24	4.19	2.79	3.47
SFBOM	57.47	37.49	24.06	15.36	9.68	6.30	7.22	4.18	2.80	3.47
SO	35.29	35.29	35.28	35.27	35.27	44.84	44.84	44.82	44.84	44.85
SA	<b>33.84</b>	33.85	33.83	33.82	33.84	38.96	38.95	38.95	38.96	38.94
BNDM	102	57.1	30.5	16.5	9.12	12.1	12.1	12.0	12.1	12.1
BNDMq4	-	101.0	22.44	10.21	5.95	7.34	7.34	7.35	7.35	7.33
SBNDMq4	-	94.90	21.18	9.76	5.93	5.55	<b>5.57</b>	5.56	5.58	5.58



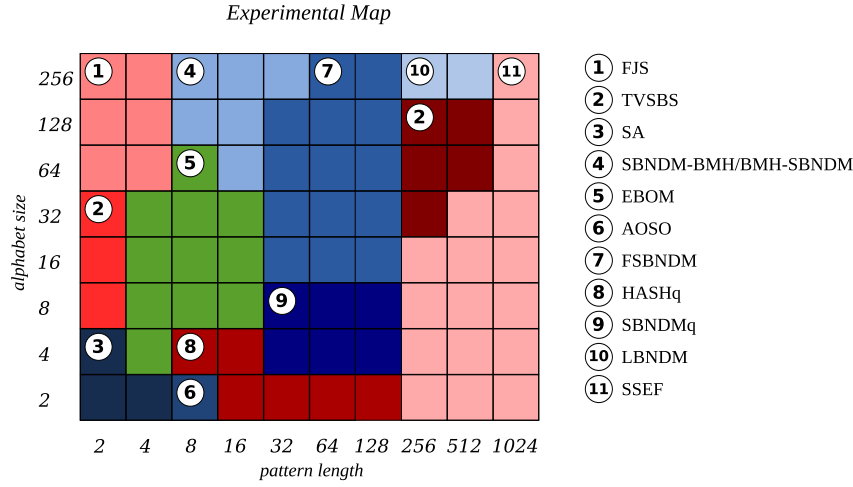


Fig. 2. Experimental map of the best results obtained in our evaluation. Comparison based algorithms are presented in red gradations, automata based algorithms are presented in green gradations and bit parallel algorithms are presented in blue gradations.

#### Experimental Results on a Protein Sequence

$m$	2	4	8	16	32	64	128	256	512	1024
ZT	20.54	10.58	5.69	3.35	2.14	1.76	2.02	1.34	0.99	0.84
RCol	9.86	5.46	3.25	2.19	1.82	1.70	1.81	1.75	1.72	1.68
BFS	9.94	5.48	3.25	2.15	1.78	1.68	1.87	1.85	1.80	2.01
SSABS	8.01	5.28	3.57	2.48	2.02	1.87	1.97	1.93	1.93	1.93
TVSBS	7.50	5.28	3.50	2.34	1.85	1.73	1.92	1.31	0.98	0.82
FJS	8.21	5.52	3.70	2.65	2.13	1.96	2.03	1.99	1.98	1.98
HASH3	-	11.81	4.39	2.29	1.61	<b>1.52</b>	1.74	1.37	1.17	1.07
HASH5	-	-	7.85	3.07	1.74	1.56	1.79	1.22	0.90	0.80
HASH8	-	-	-	4.96	2.24	1.60	1.87	1.29	0.96	0.85
SSEF	-	-	-	-	3.57	2.24	2.29	<b>1.21</b>	<b>0.63</b>	0.36
BOM	26.49	18.39	14.33	9.93	6.36	3.92	3.22	1.84	1.06	<b>0.73</b>
EBOM	12.13	4.71	<b>2.59</b>	<b>1.91</b>	1.75	1.70	1.95	1.34	1.20	2.57
SEBOM	12.83	5.01	2.77	2.03	1.89	1.81	2.07	1.46	1.28	2.61
BNDM	19.63	11.27	6.66	3.71	2.06	2.47	2.48	2.47	2.47	2.47
BMH-SBNDM	10.09	5.43	3.18	2.11	1.72	1.77	1.78	1.80	1.78	1.78
AOSO6	<b>2.33</b>	<b>2.32</b>	16.08	3.29	2.33	2.15	2.15	2.16	2.15	2.16
FSBNDM	15.27	7.77	4.15	2.42	<b>1.56</b>	1.56	<b>1.55</b>	1.54	1.56	1.56
BNDMq2	21.45	7.61	3.63	2.05	1.62	1.74	1.73	1.75	1.73	1.73
SBNDMq2	21.08	7.58	3.70	2.16	1.63	1.59	1.60	1.59	1.61	1.60

We performed comparisons between 85 exact string matching algorithms with 12 text of different types. We divide the patterns into four classes according to their length  $m$ : very short ( $m \leq 4$ ), short ( $4 < m \leq 32$ ), long ( $32 < m \leq 256$ ) and very long ( $m > 256$ ). We proceed in the same way for the alphabets according to their size  $\sigma$ : very small ( $\sigma < 4$ ), small ( $4 \leq \sigma < 32$ ), large ( $32 \leq \sigma < 128$ ) and very large ( $\sigma > 128$ ). According to our experimental results, we conclude that the following algorithms are the most efficient in the following situations (see Fig. 2):

- SA: very short patterns and very small alphabets.
- TVSBS: very short patt. and small alphabets, and long patt. and large alphabets.
- FJS: very short patterns and large and very large alphabets.
- EBOM: short patterns and large and very large alphabets.
- SBNDM-BMH and BMH-SBNDM: short patterns and very large alphabets.
- HASHq: short and large patterns and small alphabets.
- FSBNDM: long patterns and large and very large alphabets.

- SBNDMq: long patterns and small alphabets.
- LBNDM: very long patterns and very large alphabets.
- SSEF: very long patterns.

Among these algorithms all but one (the SA algorithm) have been designed during the last decade, four of them are based on comparison of characters, one of them is based on automata while six of them are bit-parallel algorithms.

## REFERENCES

- AHMED, M., KAYKOBAD, M., AND CHOWDHURY, R. A. 2003. A new string matching algorithm. *Int. J. Comput. Math.* 80, 7, 825–834.
- AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The design and analysis of computer algorithms*. Addison-Wesley.
- ALLAUZEN, C., CROCHEMORE, M., AND RAFFINOT, M. 1999. Factor oracle: a new structure for pattern matching. In *SOFSEM'99, Theory and Practice of Informatics*, J. Pavelka, G. Tel, and M. Bartosek, Eds. Number 1725 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Milovy, Czech Republic, 291–306.
- ALLAUZEN, C. AND RAFFINOT, M. 2000. Simple optimal string matching algorithm. *J. Algorithms* 36, 1, 102–116.
- APOSTOLICO, A. AND GIANCARLO, R. 1986. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.* 15, 1, 98–105.
- ARNDT, J. 2009. *Matters Computational*. <http://www.jjj.de/fxt/>.
- BAEZA-YATES, R. AND GONNET, G. H. 1992. A new approach to text searching. *Commun. ACM* 35, 10, 74–82.
- BERRY, T. AND RAVINDRAN, S. 1999. A fast string matching algorithm and experimental results. In *Proceedings of the Prague Stringology Club Workshop '99*, J. Holub and M. Šimánek, Eds. Czech Technical University, Prague, Czech Republic, 16–28. Collaborative Report DC–99–05.
- BLUMER, A., BLUMER, J., EHRENFEUCHT, A., HAUSSLER, D., CHEN, M. T., AND SEIFERAS, J. 1985. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.* 40, 1, 31–55.
- BLUMER, A., BLUMER, J., EHRENFEUCHT, A., HAUSSLER, D., AND MCCONNELL, R. 1983. 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.
- BOYER, R. S. AND MOORE, J. S. 1977. A fast string searching algorithm. *Commun. ACM* 20, 10, 762–772.
- BRESLAUER, D. 1996. Saving comparisons in the Crochemore-Perrin string matching algorithm. *Theor. Comput. Sci.* 158, 1–2, 177–192.
- CANTONE, D. AND FARO, S. 2003a. Fast-Search: a new efficient variant of the Boyer-Moore string matching algorithm. In *WEA 2003*. Lecture Notes in Computer Science, vol. 2647. Springer-Verlag, Berlin, 247–267.
- CANTONE, D. AND FARO, S. 2003b. Forward-Fast-Search: another fast variant of the Boyer-Moore string matching algorithm. In *Proceedings of the Prague Stringology Conference '03*, M. Šimánek, Ed. Czech Technical University, Prague, Czech Republic, 10–24.
- CANTONE, D. AND FARO, S. 2004. Searching for a substring with constant extra-space complexity. In *Proc. of Third International Conference on Fun with algorithms*, P. Ferragina and R. Grossi, Eds. 118–131.
- CANTONE, D. AND FARO, S. 2005. Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm. *J. Autom. Lang. Comb.* 10, 5/6, 589–608.
- CANTONE, D., FARO, S., AND GIAQUINTA, E. 2010a. Bit-(parallelism)<sup>2</sup>: Getting to the next level of parallelism. In *Fun with Algorithms*, P. Boldi and L. Gargano, Eds. Lecture Notes in Computer Science, vol. 6099. Springer-Verlag, Berlin, 166–177.
- CANTONE, D., FARO, S., AND GIAQUINTA, E. 2010b. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In *Combinatorial Pattern Matching*, A. Amir and L. Parida, Eds. Lecture Notes in Computer Science, vol. 6129. Springer-Verlag, Berlin, 288–298.
- CHARRAS, C. AND LECROQ, T. 2004. *Handbook of exact string matching algorithms*. King's College Publications.
- COLUSSI, L. 1994. Fastest pattern matching in strings. *J. Algorithms* 16, 2, 163–189.
- CROCHEMORE, M. 1985. Optimal factor transducers. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Eds. NATO Advanced Science Institutes, Series F, vol. 12. Springer-Verlag, Berlin, 31–44.

- CROCHEMORE, M. 1986. Transducers and repetitions. *Theor. Comput. Sci.* 45, 1, 63–86.
- CROCHEMORE, M., CZUMAJ, A., GĄSIENIEC, L., JAROMINEK, S., LECROQ, T., PLANDOWSKI, W., AND RYTTER, W. 1994. Speeding up two string matching algorithms. *Algorithmica* 12, 4/5, 247–267.
- CROCHEMORE, M. AND LECROQ, T. 2008. A fast implementation of the Boyer-Moore string matching algorithm. *manuscript*.
- CROCHEMORE, M. AND PERRIN, D. 1991. Two-way string-matching. *J. Assoc. Comput. Mach.* 38, 3, 651–675.
- CROCHEMORE, M. AND RYTTER, W. 1994. *Text algorithms*. Oxford University Press.
- DEUSDADO, S. AND CARVALHO, P. 2009. GRASPM: an efficient algorithm for exact pattern-matching in genomic sequences. *Int. J. Bioinformatics Res. Appl.* 5, 4, 385–401.
- DÖMÖLKI, B. 1968. A universal compiler system based on production rules. *BIT Numerical Mathematics* 8, 262–275.
- DURIAN, B., HOLUB, J., PELTOLA, H., AND TARHIO, J. 2009. Tuning BNDM with q-grams. In *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2009*, I. Finocchi and J. Hershberger, Eds. SIAM, New York, New York, USA, 29–37.
- FAN, H., YAO, N., AND MA, H. 2009. Fast variants of the backward-oracle-marching algorithm. In *Proceedings of the 2009 Fourth International Conference on Internet Computing for Science and Engineering*. ICICSE '09. IEEE Computer Society, Washington, DC, USA, 56–59.
- FARO, S. AND LECROQ, T. 2008. Efficient variants of the Backward-Oracle-Matching algorithm. In *Proceedings of the Prague Stringology Conference 2008*, J. Holub and J. Ždárek, Eds. Czech Technical University in Prague, Czech Republic, 146–160.
- FARO, S. AND LECROQ, T. 2010. The exact string matching problem: a comprehensive experimental evaluation. Report arXiv:1012.2547.
- FRANEK, F., JENNINGS, C. G., AND SMYTH, W. F. 2007. A simple fast hybrid pattern-matching algorithm. *J. Discret. Algorithms* 5, 4, 682–695.
- FREDRIKSSON, K. AND GRABOWSKI, S. 2005. Practical and optimal string matching. In *SPIRE*, M. P. Consens and G. Navarro, Eds. Lecture Notes in Computer Science, vol. 3772. Springer-Verlag, Berlin, 376–387.
- GALIL, Z. 1981. String matching in real time. *J. Assoc. Comput. Mach.* 28, 1, 134–149.
- GALIL, Z. AND SEIFERAS, J. 1983. Time-space optimal string matching. *J. Comput. Syst. Sci.* 26, 3, 280–294.
- GĄSIENIEC, L. AND KOLPAKOV, R. 2004. Real-time string matching in sublinear space. S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, Eds. 117–129.
- HANCART, C. 1993. On Simon's string searching algorithm. *Inf. Process. Lett.* 47, 2, 95–99.
- HE, L., FANG, B., AND SUI, J. 2005. The wide window string matching algorithm. *Theor. Comput. Sci.* 332, 1–3, 391–404.
- HOLUB, J. AND DURIAN, B. 2005. 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>.
- HORSPOOL, R. N. 1980. Practical fast searching in strings. *Softw. Pract. Exp.* 10, 6, 501–506.
- HUDAIB, A., AL-KHALID, R., SULEIMAN, D., ITRI, M., AND AL-ANANI, A. 2008. A fast pattern matching algorithm with two sliding windows (TSW). *J. Comput. Sci.* 4, 5, 393–401.
- HUME, A. AND SUNDAY, D. M. 1991. Fast string searching. *Softw. Pract. Exp.* 21, 11, 1221–1248.
- KALSI, P., PELTOLA, H., AND TARHIO, J. 2008. Comparison of exact string matching algorithms for biological sequences. In *Proceedings of the Second International Conference on Bioinformatics Research and Development, BIRD'08*, M. Elloumi, J. Küng, M. Linial, R. F. Murphy, K. Schneider, and C. Toma, Eds. Communications in Computer and Information Science, vol. 13. Springer-Verlag, Berlin, Vienna, Austria, 417–426.
- KARP, R. M. AND RABIN, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 31, 2, 249–260.
- KNUTH, D. E., MORRIS, JR., J. H., AND PRATT, V. R. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6, 1, 323–350.
- KÜLEKCI, M. O. 2008. A method to overcome computer word size limitation in bit-parallel pattern matching. In *Proceedings of the 19th International Symposium on Algorithms and Computation, ISAAC 2008*, S.-H. Hong, H. Nagamochi, and T. Fukunaga, Eds. Lecture Notes in Computer Science, vol. 5369. Springer-Verlag, Berlin, Gold Coast, Australia, 496–506.

- KÜLEKCI, M. O. 2009. Filter based fast matching of long patterns by using simd instructions. In *Proceedings of the Prague Stringology Conference 2009*, J. Holub and J. Žďárek, Eds. Czech Technical University in Prague, Czech Republic, 118–128.
- LECROQ, T. 2007. Fast exact string matching algorithms. *Inf. Process. Lett.* 102, 6, 229–235.
- LIU, C., WANG, Y., LIU, D., AND LI, D. 2006. Two improved single pattern matching algorithms. In *ICAT Workshops*. IEEE Computer Society, Hangzhou, China, 419–422.
- MANCHERON, A. AND MOAN, C. 2005. Combinatorial characterization of the language recognized by factor and suffix oracles. *Int. J. Found. Comput. Sci.* 16, 6, 1179–1191.
- MORRIS, JR., J. H. AND PRATT, V. R. 1970. A linear pattern-matching algorithm. Report 40, University of California, Berkeley.
- NAVARRO, G. 2001. NR-grep: a fast and flexible pattern-matching tool. *Softw. Pract. Exp.* 31, 13, 1265–1312.
- NAVARRO, G. AND RAFFINOT, M. 1998. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, M. Farach-Colton, Ed. Number 1448 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Piscataway, NJ, 14–33.
- NAVARRO, G. AND RAFFINOT, M. 2000. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. Experimental Algorithmics* 5, 4.
- NEBEL, M. E. 2006. Fast string matching by using probabilities: on an optimal mismatch variant of Horspool’s algorithm. *Theor. Comput. Sci.* 359, 1, 329–343.
- PELTOLA, H. AND TARHIO, J. 2003. Alternative algorithms for bit-parallel string matching. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval SPIRE’03*, M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, Eds. Lecture Notes in Computer Science, vol. 2857. Springer-Verlag, Berlin, Manaus, Brazil, 80–94.
- RAITA, T. 1992. Tuning the Boyer-Moore-Horspool string searching algorithm. *Softw. Pract. Exp.* 22, 10, 879–884.
- SHEIK, S., AGGARWAL, S., PODDAR, A., BALAKRISHNAN, N., AND SEKAR, K. 2004. A fast pattern matching algorithm. *J. Chem. Inf. Comput.* 44, 1251–1256.
- SIMON, I. 1993. String matching algorithms and automata. In *Proceedings of the 1st South American Workshop on String Processing*, R. Baeza-Yates and N. Ziviani, Eds. Universidade Federal de Minas Gerais, Brazil, 151–157.
- SUNDAY, D. M. 1990. A very fast substring search algorithm. *Commun. ACM* 33, 8, 132–142.
- SUSTIK, M. AND MOORE, J. 2007. String searching over small alphabets. In *Technical Report TR-07-62*. Department of Computer Sciences, University of Texas at Austin.
- THATHOO, R., VIRMANI, A., LAKSHMI, S. S., BALAKRISHNAN, N., AND SEKAR, K. 2006. TVSBS: A fast exact pattern matching algorithm for biological sequences. *J. Indian Acad. Sci., Current Sci.* 91, 1, 47–53.
- WU, S. AND MANBER, U. 1992. Fast text searching allowing errors. *Commun. ACM* 35, 10, 83–91.
- WU, S. AND MANBER, U. 1994. A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ.
- YAO, A. C. 1979. The complexity of pattern matching for a random string. *SIAM J. Comput.* 8, 3, 368–387.
- ZHANG, G., ZHU, E., MAO, L., AND YIN, M. 2009. A bit-parallel exact string matching algorithm for small alphabet. In *Proceedings of the Third International Workshop on Frontiers in Algorithmics, FAW 2009, Hefei, China*, X. Deng, J. E. Hopcroft, and J. Xue, Eds. Lecture Notes in Computer Science, vol. 5598. Springer-Verlag, Berlin, 336–345.
- ZHU, R. F. AND TAKAOKA, T. 1987. On improving the average case of the Boyer-Moore string matching algorithm. *J. Inform. Process.* 10, 3, 173–177.

Received R; revised e; accepted c

eived Month Year; revised Month Year; accepted Month Year