A Multiple Sliding Windows Approach to Speed Up String Matching Algorithms

Simone Faro[†] and Thierry Lecroq[‡]

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

Abstract. In this paper we present a general approach to string matching based on multiple sliding text-windows, and show how it can be applied to some among the most efficient algorithms for the problem based on nondeterministic automata and comparison of characters.
From our experimental results it turns out that the new multiple sliding windows approach leads to algorithms which obtain better results than the original ones when searching texts over relatively large alphabets. Best improvements are obtained especially for short patterns.

Keywords. string matching, bit parallelism, text processing, nondeterministic automata, design and analysis of algorithms, natural languages.

1 Introduction

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the *string matching problem* consists in finding *all* occurrences of the pattern p in t. This problem has been extensively studied in computer science because of its direct application to many areas. Moreover string matching algorithms are basic components in many software applications and play an important role in theoretical computer science by providing challenging problems.

Among the most efficient solutions the Boyer-Moore algorithm [1] is a comparison based algorithm which deserves a special mention since it has been particularly successful and has inspired much work. Also automata based solutions have been developed to design algorithms with efficient performance on average. This is done by using factor automata, data structures which identify all factors of a word. Among them the Extended Backward Oracle Matching algorithm [6] (EBOM for short) is one of the most efficient algorithms especially for long patterns. Another algorithm based on the bit-parallel simulation [17] of the nondeterministic factor automaton, and called Backward Nondeterministic Dawg Match algorithm [13] (BNDM), is very efficient for short patterns.

Most string matching algorithms are based on a general framework which works by scanning the text with the help of a substring of the text, called *window*, whose size is equal to m. An *attempt* of the algorithm consists in checking whenever the current window is an occurrence of the pattern. This check is generally carried out by comparing the pattern and the window character by character, or by performing transitions on some kind of automaton. After a whole

match of the pattern (or after a mismatch is detected) the window is shifted to the right by a certain number of positions, according to a shift strategy. This approach is usually called the *sliding window mechanism*.

At the beginning of the search the left end of the window is aligned with the left end of the text, then the sliding window mechanism is repeated until the right end of the window goes beyond the right end of the text.

In this paper we investigate the performance of a straightforward, yet efficient, approach which consists in sliding two or more windows of the same size along the text and in performing comparisons (or transitions) at the same time as long as possible, with the aim of speeding up the searching phase. We call this general method the *multiple sliding windows approach*.

The idea of sliding multiple windows along the text is not original. It was firstly introduced in [11] where the author presented a simple algorithm, called Two Sliding Windows, which divides the text into two parts of size $\lceil n/2 \rceil$ and searches for matches of p in t by sliding two windows of size m. 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. An additional test is performed for searching occurrences in the middle of the text.

More recently Cantone *et al.* proposed in [3] a similar approach for increasing the instruction level parallelism of string matching algorithms based on bit-parallelism. This technique, called *bit-(parallelism)*^{2}, is general enough to be applied to a lot of bit-parallel algorithms.

It includes two different approaches which run two copies of the same automata in parallel and process two adjacent windows simultaneously, sliding them from left to right. In both cases the two automata are encoded by using a single computer word. Thus, due to its representation, the approach turns out to be efficient only for searching very short patterns.

However the two approaches introduced by the bit-parallelism^2 technique are not general enough to be applied to all bit-parallel algorithms. For instance the structure of the BNDM algorithm does not allow it to process in parallel two adjacent (or partially overlapping) windows, since the starting position of the next window alignment depends on the the shift value performed by the leftmost one. Unfortunately such a shift has not a fixed value and it can be computed only at the end of the each attempt.

Moreover the approaches proposed in [3] can be applied only for improving the performance of automata based algorithms, thus they are not applicable for all algorithms based on comparison of characters.

The paper is organized as follows. In Section 2 we introduce some notation and the terminology used along the paper. In Section 3 we describe the new general multiple sliding windows approach and apply it to some of the most efficient algorithms based on the simulation of nondeterministic automata (Section 3.1) and on comparison of characters (Section 3.2). Finally we present an experimental evaluation in Section 4. Conclusions are drawn in Section 5.

2 Notations and Terminology

Throughout the paper we will make use of the following notations and terminology. A string p of length $m > 0$ is represented as a finite array $p[0\,..\,m-1]$ of characters from a finite alphabet Σ of size σ. Thus $p[i]$ will denote the $(i+1)$-st character of p, for $0 \leq i < m$, and $p[i\,..\,j]$ will denote the *factor* (or *substring*) of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, for $0 \leq i \leq j < m$. A factor of the form $p[0\,..\,i]$ is called a *prefix* of p and a factor of the form $p[i\,..\,m-1]$ is called a *suffix* of p for $0 \leqslant i < m$. We denote with $Fact(p)$ the set of all factors of a string p.

The factor automaton of a pattern p, also called the factor DAWG of p (for Directed Acyclic Word Graph), is a Deterministic Finite State Automaton (DFA) which recognizes all the factors of p. Formally its recognized language is defined as the set $\{u \in \Sigma^* \mid \text{ exists } v, w \in \Sigma^* \text{ such that } p = vuw\}$.

We also denote the reverse of the string p by $\bar{p}$, i.e. $\bar{p} = p[m-1]p[m-2]\cdots p[0]$.

Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|" and the `left shift` "$\ll$" operator (which shifts to the left its first argument by a number of bits equal to its second argument).

3 A General Multiple Sliding Windows Approach

In this section we describe a general multiple windows approach which can be used for improving the practical performances of a large class of string matching algorithms, including all comparison based algorithms. The general approach can be seen as a filtering method which consists in processing k different windows of the text at the same time, with $k \geq 2$. Then, specific occurrences of the pattern are tested only when candidate positions have been located.

Suppose p is a pattern of length m and t is a text of length n. Without loss in generality we can suppose that n can be divided by k, otherwise the rightmost $(n \mod k)$ characters of the text could be associated with the last window (as described below). Moreover we assume for simplicity that $m < n/k$ and that the value k is even. Under the above assumptions the new approach can be summarized as follows: if the algorithm searches for the pattern p in t using a text window of size m, then partition the text in $k/2$ partially overlapping substrings, $t_0, t_1, \ldots, t_{k/2-1}$, where t_i is the substring $t[2i\lceil n/k\rceil\,..\,2(i+1)n/k + m - 2]$, for $i = 0, \ldots, (k-1)/2$, and $t_{k/2-1}$ (the last window) is set to $t[n - (2n/k)\,..\,n-1]$.

Then process simultaneously the k different text windows, $w_0, w_1, \ldots, w_{k-1}$, where we set $w_{2i} = t[s_{2i} - m + 1\,..\,s_{2i}]$ (and call them *left windows*) and $w_{2i+1} = t[s_{2i+1}\,..\,s_{2i+1} + m - 1]$ (and call them *right windows*), for $i = 0, \ldots, (k-2)/2$.

The couple of windows $(w_{2i},\ w_{2i+1})$, for $i = 0, \ldots, (k-2)/2$, is used to process the substring of the text t_i. Specifically the window w_{2i} starts from position $s_{2i} = (2n/k)i + m - 1$ of t and slides from left to right, while window w_{2i+1} starts from position $s_{2i+1} = (2n/k)(i+1) - 1$ of t and slides from right to left (the window w_{k-1} starts from position $s_{k-1} = n - m$). For each couple

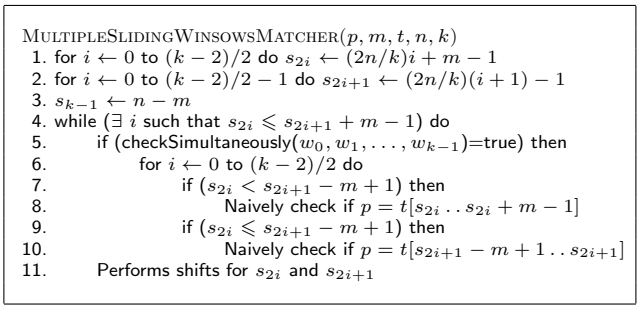

Fig. 1. A General Multiple Sliding Windows Matcher

of windows (w_{2i}, w_{2i+1}) the sliding process ends when the window w_{2i} slides over the window w_{2i+1}, i.e. when $s_{2i} > s_{2i+1} + m - 1$. It is easy to prove that no candidate occurrence is left by the algorithm due to the $m-1$ overlapping characters between adjacent substrings t_i and t_{i+1}, for $i = 0, \ldots, k-2$.

Fig. 1 shows the pseudocode of a general multiple sliding windows matcher which processes k windows in parallel, while Fig. 2 presents a scheme of the search iteration of the multiple sliding windows matcher for $k = 1$, 2 and 4.

Procedure checkSimultaneously is used as a filtering method for locating candidate occurrences of the pattern in the k different text windows. Observe that, when n cannot be divided by k, the assignment of line 3, which set s_{k-1} to $n-m$, implies that the rightmost ($n \mod k$) characters of the text are associated with the last window. Moreover it can be proved that the general matcher algorithm shown in Fig. 1 is correct if the value of checkSimultaneously($w_0, w_1, \ldots, w_{k-1}$) is true whenever $w_i = p$ for some i in the set $\{0, \ldots, k-1\}$.

If $s_{2i} = s_{2i+1} + m - 1$ (which implies $w_{2i} = w_{2i+1}$) and a candidate position is found, only the right window is checked for a real occurrence, in order to avoid to report a duplicate occurrence of the pattern. This is done in lines 7 and 9.

This general approach can be applied to all string matching algorithms, including comparison based and BMDN based algorithms. Moreover it can be noticed that the worst case time complexity of the original algorithm does not degrade with the application of the multiple sliding windows approach.

From a practical point of view, when computation of the filter can be done in parallel and the alphabet is large enough, the new approach leads to more efficient algorithms. As an additional feature, observe also that the two way sliding approach enables exploitation of the structure of the pattern in both directions, leading on average to larger shift advancements and improving further the performance of the algorithm. On the other hand, when the alphabet is small the performances of the original algorithm degrade by applying the new method, since the probability to find mixed candidate positions increases substantially.

In the following sections we present simple variants, based on the multiple sliding windows approach, of some among the most efficient algorithms based on comparison of characters and bit-parallelism.

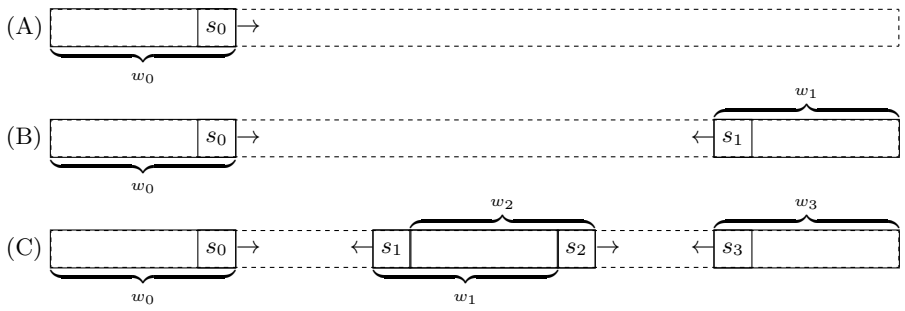

Fig. 2. A general scheme for the multiple sliding windows matcher. The scheme with (A) a single window, (B) two windows and (C) four windows.

3.1 Multiple Windows Variants of Bit-Parallel Algorithms

In the context of string matching, bit-parallelism [17] is used for simulating the behavior of Nondeterministic Finite State Automata (NFA) and allows multiple transactions to perform in parallel with a constant number of operations.

In this section we show the application of the multiple sliding windows approach to a simplified version of the BNDM algorithm [13] which, among the several algorithms based on bit-parallelism, deserves a particular attention as it has inspired a lot of variants and is still considered one of the fastest algorithms. Among the various improvements of the BNDM algorithm we mention the Simplified BNDM algorithm improved with Horspool shift [9], with q-grams [4, 15], and with lookahead characters [6, 15]. The BNDM algorithm has been also modified in order to match long patterns [5] and binary strings [7].

Specifically, the BNDM algorithm simulates the suffix automaton of $\bar{p}$. Its bit-parallel representation of the suffix automaton uses an array, B_p, of σ bit-vectors, each of size m, where the i-th bit of $B_p[c]$ is set iff $p[i] = c$, for $c \in \Sigma$ and $0 \leqslant i < m$. The algorithm works by shifting a window of length m over the text. Specifically, for each text window alignment $t[s-m+1\,..\,s]$, it searches the pattern by scanning the current window backwards and updating the automaton configuration accordingly. In the simplified version of BNDM the bit vector D is initialized to $B_p[p[s]]$, i.e. the configuration of the automaton after the first transition. Then any subsequent transition on character c can be implemented as $D \leftarrow ((D \ll 1)\ \&\ B_p[c])$.

An attempt ends when either D becomes zero (i.e., when no further factors 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 factor.

Fig. 3 shows the code of the Simplified BNDM algorithm [14] (SBNDM for short) and the code of its multiple sliding windows variant with 4 windows.

In general the k-windows variant of a SBNDM algorithm simulates k different copies of two suffix automata, one for the reverse of the pattern and one for the

```
SBNDM(p, m, t, n)
 1. for each c ∈ Σ do B_p(c) ← 0^m
 2. F_p ← 0^{m−1}1
 3. for i ← m − 1 downto 0 do
 4.     B_p(p[i]) ← B_p(p[i]) | F_p
 5.     F_p ← F_p ≪ 1
 6. s ← m − 1
 7. while (s ≤ n − 1) do
 8.     D ← B_p(t[s])
 9.     j ← s − m + 1
10.     while (D ≠ 0) do
11.         s ← s − 1
12.         D ← (D ≪ 1) & B_p(t[s])
13.     if (s < j) then
14.         s ← s + 1
15.         Output(s)
16.     s ← s + m
```

```
SBNDM-W4(p, m, t, n)
 1. for each c ∈ Σ do B_p(c) ← B_p̄ ← 0^m
 2. F_p ← 0^{m−1}1
 3. for i ← m − 1 downto 0 do
 4.     B_p(p[i]) ← B_p(p[i]) | F_p
 5.     B_p̄(p[m − 1 − i]) ← B_p̄(p[m − 1 − i]) | F_p
 6.     F_p ← F_p ≪ 1
 7. s_0 ← m − 1; s_1 ← n/2 − 1; s_2 ← n/2 + m − 1; s_3 ← n − m
 8. while (s_0 ≤ s_1 + m − 1 and s_2 ≤ s_3 + m − 1) do
 9.     D ← B_p(t[s_0]) | B_p̄(t[s_1]) | B_p(t[s_2]) | B_p̄(t[s_3])
10.     j ← s_0 − m + 1
11.     while (D ≠ 0) do
12.         s_0 ← s_0 − 1; s_1 ← s_1 + 1; s_2 ← s_2 − 1; s_3 ← s_3 + 1
13.         D ← D ≪ 1
14.         D ← D & (B_p(t[s_0]) | B_p̄(t[s_1]) | B_p(t[s_2]) | B_p̄(t[s_3]))
15.     if (s_0 < j) then
16.         s_0 ← s_0 + 1; s_1 ← s_1 − 1; s_2 ← s_2 + 1; s_3 ← s_3 − 1
17.         if (s_0 < s_1 + m − 1 and p = t[s_0 − m + 1 .. s_0])
18.             then Output(s_0 − m + 1)
19.         if (s_0 ≤ s_1 + m − 1 and p = t[s_1 .. s_1 + m − 1])
20.             then Output(s_1)
21.         if (s_2 < s_3 + m − 1 and p = t[s_2 − m + 1 .. s_2])
22.             then Output(s_2 − m + 1)
23.         if (s_2 ≤ s_3 + m − 1 and p = t[s_3 .. s_3 + m − 1])
24.             then Output(s_3)
25.     s_0 ← s_0 + m; s_1 ← s_1 − m; s_2 ← s_2 + m; s_3 ← s_3 − m
```

Fig. 3. (On the left) The SBNDM algorithm and (on the right) the multiple sliding windows variant of the SBNDM algorithm with 4 windows.

pattern p itself. Their bit-parallel representations use two arrays $B_{\bar{p}}$ and B_p, respectively, of σ bit-vectors, each of size m.

Specifically the i-th bit of $B_p[c]$ is set iff $p[i] = c$, while the i-th bit of $B_{\bar{p}}[c]$ is set iff $p[m-i-1] = c$, for $c \in \Sigma$, $0 \leqslant i < m$. Then while searching, left windows use vector B_p while right windows use vector $B_{\bar{p}}$, to perform transitions.

A bit-vector D, with m bits, is used for simulating the mixed behavior of the k automata, when running in parallel. Specifically the i-th bit of D is set to 1 iff the i-th state of, at least, one of the k automata is active. For this purpose, at the beginning of each attempt, the bit-vector D is initialized with the assignment

$$D \leftarrow B_p[t[s_0]] \mid B_{\bar{p}}[t[s_1]] \mid B_p[t[s_2]] \mid \cdots \mid B_{\bar{p}}[t[s_{k-1}]].$$

Then the k windows are searched for candidate occurrences of the pattern by scanning the left windows backwards and the right windows forwards, and updating the mixed automaton configurations accordingly. The transitions are performed in parallel for all k windows by applying the following bitwise operations, for $i = 1$ to $m - 1$

$$D \leftarrow D \ll 1$$
$$D \leftarrow D \;\&\; (B_p[t[s_0 - i]] \mid B_{\bar{p}}[t[s_1 + i]] \mid B_p[t[s_2 - i]] \mid \cdots \mid B_{\bar{p}}[t[s_{k-1} + i]]).$$

An attempt ends when either D becomes zero (i.e., when no further candidate factors of p or $\bar{p}$ can be found) or the algorithm inspected m characters of

the windows (i.e., when candidate matches has been found). In the last case a additional naive test is performed in order to check if any of the current windows corresponds in a whole match of the pattern.

At the end of each attempt the left and the right windows are shifted to the right and to the left, respectively, in order to be aligned with the start position of the longest recognized proper candidate factor.

3.2 Multiple Windows Variants of Comparison Based Algorithms

Most efficient comparison based algorithms are variants of the well-known Boyer-Moore algorithm [1]. It compares characters of the pattern and the current window of the text 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*, provided that both of them are applicable (see [1] for more details).

Many variants of the Boyer-Moore algorithm have been proposed over the years, mostly focusing on the bad character rule. The first practical variant was introduced by Horspool in [10], which proposed a simple modification of the original rule and used it as a simple filtering method for locating candidate occurrences of the pattern.

Specifically the Horspool algorithm labels the current window of the text, $t[s-m+1\,..\,s]$, as a candidate occurrence if $p[m-1] = t[s]$. Failing this the shift advancement is computed in such a way that the rightmost character of the current window, $t[s]$, 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 $hbc_p(t[s])$ positions where, for all $c \in \Sigma$, $hbc_p(c) = \min(\{0 < k < m \mid p[m-1-k] = c\} \cup \{m\})$.

Otherwise, when $t[s] = p[m-1]$ a candidate occurrence of the pattern has been located and a naive test is used for checking the whole occurrence (there is no need to test again if $p[m-1] = t[s]$). After an occurrence is found the pattern is advanced of an amount equal to $\min(\{0 < k < m \mid p[m-1-k] = t[s]\} \cup \{m\})$.

Fig. 4 (on the left) shows the code of the Horspool algorithm and (on the right) the code of its multiple sliding windows variant with 4 windows.

In general the k-windows variant of the Horspool algorithm searches for candidate positions by checking if the rightmost (leftmost) character of the pattern is equal to the rightmost (leftmost) character of any of the left (right) windows (line 11 in our example code).

A lot of variants of the Horspool algorithm have been proposed over the years. In this paper we propose the application of the new approach to the Fast-Search algorithm [2] and to the TVSBS algorithm [16], since they turn out to be among the most efficient in practice.

In particular the Fast-Search algorithm computes its shift increments by applying the Horspool bad-character rule when $p[m-1] \neq t[s]$, otherwise, if a candidate occurrence is found, it uses the good-suffix rule for shifting.

Differently the TVSBS algorithm discards the good suffix rule and uses the first and the last characters of the current window of the text, $(t[s-m+1]$,

```
HORSPOOL(p, m, t, n)
1. for each c ∈ Σ do hbc_p(c) ← m
2. for i ← 0 to m − 2 do
3.     hbc_p(p[i]) ← m − 1 − i
4. s ← m − 1
5. c_l ← p[m − 1]
6. while (s ≤ n − 1) do
7.     if (c_l = t[s]) then
8.         if p = t[s − m + 1 .. s]
9.             then Output(s − m + 1)
10.    s ← s + hbc_p(t[s])
```

```
HORSPOOL-W4(p, m, t, n)
1. for each c ∈ Σ do hbc_p(c) ← hbc_p̄(c) ← m
2. for i ← 0 to m − 2 do
3.     hbc_p(p[i]) ← m − 1 − i
4.     hbc_p̄(p[m − 1 − i]) ← i
5. s_0 ← m − 1, s_1 ← n/2 − 1, s_2 ← n/2 + m − 1, s_3 ← n − m
9. c_f = p[0]; c_l ← p[m − 1]
10. while (s_0 ≤ s_1 + m − 1 or s_2 ≤ s_3 + m − 1) do
11.    if (c_l = t[s_0] or c_f = t[s_1] or c_l = t[s_2] or c_f = t[s_3]) then
12.        if (s_0 < s_1 + m − 1 and p = t[s_0 − m + 1 .. s_0])
13.            then Output(s_0 − m + 1)
14.        if (s_0 ≤ s_1 + m − 1 and p = t[s_1 .. s_1 + m − 1])
15.            then Output(s_1)
16.        if (s_2 < s_3 + m − 1 and p = t[s_2 − m + 1 .. s_2])
17.            then Output(s_2 − m + 1)
18.        if (s_2 ≤ s_3 + m − 1 and p = t[s_3 .. s_3 + m − 1])
19.            then Output(s_3)
20.    s_0 ← s_0 + hbc_p(t[s_0]), s_1 ← s_1 − hbc_p̄(t[s_1])
22.    s_2 ← s_2 + hbc_p(t[s_2]), s_3 ← s_3 − hbc_p̄(t[s_3])
```

Fig. 4. (On the left) The Horspool algorithm and (on the right) the multiple sliding windows variant of the Horspool algorithm with 4 windows.

$t[s]$), for locating a candidate occurrence of the pattern, while using the couple of characters ($t[s+1]$, $t[s+2]$) for computing the shift advancement.

4 Experimental Results

In this section we compare, in terms of running times and under various conditions, the performances of the multiple sliding windows variants of some among the most efficient string matching algorithms. Specifically we tested the k-windows variants of the following algorithms:

- Fast-Search (FS-W(k)), with $k \in \{1, 2, 4, 6, 8\}$
- TVSBS (TVSBS-W(k)), with $k \in \{1, 2, 4, 6, 8\}$
- Simplified BNDM (SBNDM-W(k)), with $k \in \{1, 2, 4, 6\}$
- Forward Simplified BNDM (FSBNDM-W(k)), with $k \in \{1, 2, 4, 6\}$

Observe that when k is equal to 1 we refer to the original algorithms.

All algorithms have been implemented in the C programming language and tested using the smart research tool [8]. The code used in our evaluation can be downloaded from the smart tool.

The experiments have been conducted on a MacBook Pro with a 2 GHz Intel Core i7 processor, a 4 GB 1333 MHz DDR3 memory and a 64 bit word size. In particular, all algorithms have been tested on seven 4MB random text buffers (randσ in smart) over alphabets of size σ with a uniform character distribution, where σ ranges in the set $\{16, 32, 64, 128\}$. For each input file, we have searched sets of 1000 patterns of fixed length m randomly extracted from the text, for m ranging from 2 to 512. Then, the mean of the running times has been reported.

(A) m	2	4	8	16	32	64
$\sigma = 16$						
FS-W(1)	15.83	10.79	8.32	7.26	6.97	6.91
FS-W(2)	11.31	8.12	6.56	5.89	5.72	5.69
FS-W(4)	**9.41**	**7.11**	**5.93**	**5.48**	**5.37**	**5.36**
FS-W(6)	9.59	7.20	5.99	5.51	5.41	5.38
FS-W(8)	9.98	7.41	6.14	5.63	5.48	5.48
$\sigma = 32$						
FS-W(1)	15.02	10.18	7.79	6.69	6.44	6.39
FS-W(2)	10.21	7.45	6.06	5.45	5.39	5.41
FS-W(4)	**8.13**	**6.31**	5.46	5.08	5.07	5.14
FS-W(6)	8.17	6.34	**5.45**	**5.07**	**5.06**	**5.10**
FS-W(8)	8.22	6.35	5.47	5.12	5.10	5.13
$\sigma = 128$						
FS-W(1)	14.29	9.69	7.43	6.34	5.88	5.92
FS-W(2)	9.43	6.97	5.76	5.21	5.04	5.10
FS-W(4)	7.16	5.79	5.17	4.90	4.88	4.95
FS-W(6)	7.18	5.79	5.15	**4.87**	**4.84**	**4.90**
FS-W(8)	**7.04**	**5.73**	**5.12**	4.91	4.87	4.91

(B) m	2	4	8	16	32	64
$\sigma = 16$						
TVSBS-W(1)	12.89	10.32	8.30	7.01	6.28	5.99
TVSBS-W(2)	10.00	8.31	6.98	6.04	5.62	5.49
TVSBS-W(4)	**9.77**	**8.13**	6.84	5.94	5.51	5.37
TVSBS-W(6)	10.30	8.22	**6.81**	**5.92**	**5.48**	**5.33**
TVSBS-W(8)	12.31	8.76	7.02	6.08	5.64	5.52
$\sigma = 32$						
TVSBS-W(1)	12.18	9.90	8.10	6.93	6.21	5.94
TVSBS-W(2)	9.64	8.05	6.83	5.95	5.54	5.42
TVSBS-W(4)	**8.65**	**7.38**	6.36	5.68	5.36	5.29
TVSBS-W(6)	8.80	**7.38**	**6.32**	**5.65**	**5.33**	**5.24**
TVSBS-W(8)	10.26	8.00	6.77	5.92	5.53	5.38
$\sigma = 128$						
TVSBS-W(1)	13.88	10.99	8.76	7.29	6.46	6.07
TVSBS-W(2)	10.15	8.53	7.18	6.27	5.77	5.54
TVSBS-W(4)	8.51	7.38	6.40	**5.78**	**5.53**	**5.44**
TVSBS-W(6)	**8.26**	**7.19**	**6.28**	5.84	5.63	**5.44**
TVSBS-W(8)	9.44	8.07	7.04	6.38	6.02	5.69

(C) m	2	4	8	16	32	64
$\sigma = 16$						
SBNDM-W(1)	12.9	9.73	8.20	6.59	**5.61**	**5.61**
SBNDM-W(2)	**11.2**	8.57	6.66	**5.52**	6.85	6.90
SBNDM-W(4)	11.7	**7.97**	**6.41**	5.70	9.21	9.36
SBNDM-W(6)	12.5	8.75	6.99	5.80	16.8	17.1
$\sigma = 32$						
SBNDM-W(1)	11.4	8.56	7.22	6.62	**5.78**	**5.79**
SBNDM-W(2)	9.03	7.18	6.31	5.46	5.97	5.96
SBNDM-W(4)	**8.48**	**6.88**	**5.84**	**5.22**	6.08	6.08
SBNDM-W(6)	9.41	7.22	5.90	5.31	7.53	7.58
$\sigma = 128$						
SBNDM-W(1)	10.6	7.86	6.57	5.97	5.89	5.85
SBNDM-W(2)	7.76	6.23	5.49	5.19	5.20	5.18
SBNDM-W(4)	**6.54**	**5.61**	**5.22**	**5.06**	**5.11**	**5.09**
SBNDM-W(6)	7.08	5.76	5.32	5.11	5.26	5.28

(D) m	2	4	8	16	32	64
$\sigma = 16$						
FSBNDM-W(1)	11.5	8.26	6.72	5.95	5.53	5.54
FSBNDM-W(2)	**10.4**	**7.34**	6.05	5.71	5.28	5.30
FSBNDM-W(4)	10.9	7.42	**6.01**	**5.43**	**5.08**	**5.11**
FSBNDM-W(6)	12.6	8.31	6.39	5.63	5.27	5.27
$\sigma = 32$						
FSBNDM-W(1)	9.75	7.36	6.22	5.62	5.35	5.34
FSBNDM-W(2)	8.91	6.69	5.60	5.40	5.19	5.20
FSBNDM-W(4)	**8.44**	**6.28**	**5.40**	**5.08**	**5.05**	**5.04**
FSBNDM-W(6)	9.42	6.78	5.62	5.21	5.10	5.10
$\sigma = 128$						
FSBNDM-W(1)	8.84	6.72	5.86	5.45	5.18	5.18
FSBNDM-W(2)	7.94	6.18	5.36	5.00	4.98	4.97
FSBNDM-W(4)	**7.32**	**5.71**	**5.12**	**4.90**	**4.89**	**4.88**
FSBNDM-W(6)	7.45	5.92	5.19	**4.90**	**4.89**	**4.88**

Table 1. Running times of multiple sliding windows variants of the **(A)** Fast-Search, **(B)** TVSBS, **(C)** SBNDM and **(D)** FSBNDM algorithms.

Table 1 shows experimental results obtained by comparing multiple sliding windows variants of the above algorithms. Running times are expressed in hundredths of seconds and best results have been boldfaced and underlined.

Best running times are obtained when a good compromise between the values of k, m and σ is reached. In the case of comparison based algorithms it turns out that the best results are obtained for $k = 4$ and $k = 6$ while, for large alphabets and short patterns, the Fast-Search algorithm performs better with $k = 8$. In this latter case the variant is up to 50% faster than the original algorithm.

In the case of bit parallel algorithm we obtain in most cases best results for multiple windows variants with $k = 4$. For small alphabets and long patterns the better results are obtained with $k = 2$. In the case of the SBNDM algorithm, when the value of m is small enough and the alphabet is large, we obtain results up to 40% better than that obtained by the original algorithm. However the performance of the new variants degrades when the length of the pattern increases,

$(\sigma = 16)$ / m	2	4	8	16	32	64	128	256	512
EBOM	**9.14**	**6.77**	6.07	5.75	5.61	5.58	5.54	5.53	5.49
HASH(q)	$17.4^{(1)}$	$11.52^{(1)}$	$8.46^{(2)}$	$6.72^{(2)}$	$5.81^{(5)}$	$5.41^{(5)}$	$5.32^{(5)}$	$5.30^{(4)}$	$5.26^{(5)}$
FSBNDM(q, f)	$10.6^{(2,1)}$	$7.6^{(2,0)}$	$6.29^{(3,1)}$	$5.63^{(3,1)}$	$5.37^{(3,1)}$	$5.40^{(3,1)}$	$5.38^{(3,1)}$	$5.39^{(3,1)}$	$5.39^{(3,1)}$
QF(q, s)	-	$7.5^{(2,4)}$	$6.16^{(2,4)}$	$5.63^{(3,4)}$	$5.37^{(3,4)}$	$5.24^{(3,4)}$	$5.18^{(3,4)}$	**4.97**$^{(3,4)}$	**4.70**$^{(3,4)}$
FSBNDM-W(k)	$10.4^{(2)}$	$7.34^{(2)}$	$6.01^{(4)}$	**5.43**$^{(4)}$	**5.08**$^{(4)}$	**5.11**$^{(4)}$	**5.11**$^{(4)}$	$5.10^{(4)}$	$5.11^{(4)}$
SBNDM-W(k)	$11.2^{(2)}$	$7.97^{(4)}$	$6.41^{(4)}$	$5.52^{(2)}$	$5.38^{(1)}$	$5.37^{(1)}$	$5.37^{(1)}$	$5.37^{(1)}$	$5.38^{(1)}$
FS-W(k)	$9.41^{(4)}$	$7.11^{(4)}$	**5.93**$^{(4)}$	$5.48^{(4)}$	$5.37^{(4)}$	$5.36^{(4)}$	$5.36^{(4)}$	$5.33^{(4)}$	$5.31^{(4)}$
TVSBS-W(k)	$9.77^{(4)}$	$8.13^{(4)}$	$6.81^{(6)}$	$5.92^{(6)}$	$5.48^{(6)}$	$5.33^{(6)}$	$5.19^{(6)}$	$5.06^{(6)}$	$4.94^{(6)}$

$(\sigma = 32)$ / m	2	4	8	16	32	64	128	256	512
EBOM	9.05	6.61	5.90	5.64	5.51	5.48	5.43	5.32	5.37
HASH(q)	$15.88^{(1)}$	$10.79^{(1)}$	$7.95^{(1)}$	$6.54^{(2)}$	$5.80^{(3)}$	$5.45^{(5)}$	$5.36^{(5)}$	$5.30^{(4)}$	$5.26^{(4)}$
FSBNDM(q, f)	$9.28^{(2,1)}$	$7.09^{(2,1)}$	$6.06^{(2,0)}$	$5.54^{(2,0)}$	$5.32^{(2,0)}$	$5.33^{(2,0)}$	$5.33^{(2,0)}$	$5.33^{(2,0)}$	$5.32^{(2,0)}$
QF(q, s)	-	$7.21^{(2,6)}$	$5.97^{(2,6)}$	$5.50^{(2,6)}$	$5.31^{(2,6)}$	$5.22^{(2,6)}$	$5.12^{(2,6)}$	$4.92^{(2,6)}$	$4.71^{(4,3)}$
FS-W(k)	**8.13**$^{(4)}$	$6.31^{(4)}$	$5.45^{(6)}$	**5.07**$^{(6)}$	$5.06^{(6)}$	$5.10^{(6)}$	$5.06^{(6)}$	$5.06^{(6)}$	$5.05^{(6)}$
FSBNDM-W(k)	$8.44^{(4)}$	**6.28**$^{(2)}$	**5.40**$^{(4)}$	$5.08^{(4)}$	**5.05**$^{(4)}$	**5.04**$^{(4)}$	**5.04**$^{(4)}$	$5.03^{(4)}$	$5.04^{(4)}$
SBNDM-W(k)	$9.03^{(2)}$	$6.88^{(4)}$	$5.84^{(4)}$	$5.22^{(4)}$	$5.57^{(1)}$	$5.55^{(1)}$	$5.57^{(1)}$	$5.57^{(1)}$	$5.56^{(1)}$
TVSBS-W(k)	$8.65^{(4)}$	$7.38^{(4)}$	$6.32^{(6)}$	$5.65^{(6)}$	$5.33^{(6)}$	$5.24^{(6)}$	$5.07^{(6)}$	**4.84**$^{(6)}$	**4.66**$^{(6)}$

$(\sigma = 64)$ / m	2	4	8	16	32	64	128	256	512
EBOM	9.75	6.81	5.95	5.65	5.51	5.47	5.25	5.29	5.29
HASH(q)	$15.18^{(1)}$	$10.11^{(1)}$	$7.60^{(1)}$	$6.46^{(2)}$	$5.70^{(2)}$	$5.44^{(5)}$	$5.35^{(4)}$	$5.31^{(4)}$	$5.27^{(4)}$
QF(q, s)	-	$7.13^{(2,6)}$	$5.92^{(2,6)}$	$5.47^{(2,6)}$	$5.28^{(2,6)}$	$5.03^{(2,6)}$	$5.11^{(2,6)}$	$4.85^{(2,6)}$	$4.61^{(2,6)}$
FSBNDM(q, f)	$8.59^{(2,1)}$	$6.80^{(2,1)}$	$5.89^{(2,1)}$	$5.44^{(2,1)}$	$5.25^{(2,1)}$	$5.27^{(2,1)}$	$5.10^{(2,1)}$	$5.27^{(2,0)}$	$5.28^{(2,0)}$
FS-W(k)	$7.37^{(8)}$	$5.91^{(8)}$	$5.20^{(8)}$	**4.92**$^{(6)}$	**4.90**$^{(6)}$	$4.94^{(8)}$	$4.97^{(6)}$	$4.92^{(6)}$	$4.91^{(6)}$
FSBNDM-W(k)	$8.02^{(6)}$	**5.87**$^{(4)}$	**5.17**$^{(4)}$	**4.92**$^{(4)}$	$4.92^{(4)}$	**4.92**$^{(4)}$	**4.95**$^{(4)}$	$4.93^{(4)}$	$4.92^{(4)}$
SBNDM-W(k)	**7.11**$^{(4)}$	$6.01^{(4)}$	$5.50^{(4)}$	$5.09^{(6)}$	$5.33^{(2)}$	$5.32^{(2)}$	$5.34^{(2)}$	$5.32^{(2)}$	$5.32^{(2)}$
TVSBS-W(k)	$8.19^{(6)}$	$7.07^{(6)}$	$6.14^{(6)}$	$5.57^{(6)}$	$5.28^{(6)}$	$5.12^{(6)}$	$5.03^{(6)}$	**4.75**$^{(6)}$	**4.54**$^{(6)}$

$(\sigma = 128)$ / m	2	4	8	16	32	64	128	256	512
EBOM	9.99	6.93	6.02	5.74	5.57	5.52	5.42	5.28	5.30
HASH(q)	$14.83^{(1)}$	$9.88^{(1)}$	$7.42^{(1)}$	$6.27^{(1)}$	$5.63^{(2)}$	$5.43^{(5)}$	$5.33^{(3)}$	$5.28^{(4)}$	$5.24^{(4)}$
QF(q, s)	-	$7.15^{(2,6)}$	$5.95^{(2,6)}$	$5.50^{(2,6)}$	$5.29^{(2,6)}$	$5.19^{(2,6)}$	$5.10^{(2,6)}$	$4.87^{(2,6)}$	$4.61^{(2,6)}$
FSBNDM(q, f)	$8.28^{(2,1)}$	$6.65^{(2,1)}$	$5.83^{(2,1)}$	$5.43^{(2,1)}$	$5.25^{(2,1)}$	$5.25^{(2,1)}$	$5.25^{(2,1)}$	$5.26^{(2,0)}$	$5.25^{(2,1)}$
FS-W(k)	$7.04^{(8)}$	$5.73^{(8)}$	**5.12**$^{(8)}$	**4.90**$^{(4)}$	**4.84**$^{(6)}$	$4.90^{(6)}$	**4.81**$^{(8)}$	**4.66**$^{(8)}$	$4.57^{(8)}$
FSBNDM-W(k)	$7.32^{(4)}$	$5.71^{(4)}$	**5.12**$^{(4)}$	**4.90**$^{(4)}$	$4.89^{(4)}$	**4.88**$^{(4)}$	$4.88^{(4)}$	$4.89^{(4)}$	$4.89^{(6)}$
SBNDM-W(k)	**6.54**$^{(4)}$	**5.61**$^{(4)}$	$5.22^{(4)}$	$5.06^{(4)}$	$5.11^{(4)}$	$5.09^{(4)}$	$5.09^{(4)}$	$5.09^{(4)}$	$5.10^{(4)}$
TVSBS-W(k)	$8.26^{(6)}$	$7.19^{(6)}$	$6.28^{(6)}$	$5.78^{(4)}$	$5.53^{(4)}$	$5.44^{(4)}$	$4.99^{(6)}$	$4.70^{(6)}$	**4.53**$^{(6)}$

Table 2. Experimental results on random text buffers over alphabets of size 16, 32, 64 and 128, respectively. Best results are boldfaced and underlined.

especially for small alphabets. In all cases best improvements are obtained in the case of short patterns.

We performed an additional comparison where we tested our proposed variants against the following efficient algorithms, which turn out to be very efficient in practical cases:

- the HASHq algorithm [12] (HASH(q)), with $q \in \{1, \ldots, 8\}$;
- the EBOM algorithm [6] (EBOM);
- the Forward SBNDM algorithm [6] enhanced with q-grams and f forward characters [15] (FSBNDM(q, f)), with $q \in \{2, \ldots, 8\}$ and $f \in \{0, \ldots, 6\}$;
- the Q-gram Filtering algorithm [5] (QF(q, s)), with $q \in \{2, \ldots, 6\}$, $s \in \{2, \ldots, 8\}$.

protein / m	2	4	8	16	32	64	128	256	512
EBOM	**9.21**	**6.83**	6.05	5.76	5.61	5.57	5.54	5.51	5.47
HASH(q)	$17.29^{(1)}$	$11.45^{(1)}$	$8.35^{(2)}$	$6.68^{(2)}$	$5.90^{(3)}$	$5.49^{(5)}$	$5.41^{(5)}$	$5.36^{(4)}$	$5.31^{(5)}$
QF(q,s)	-	$7.65^{(2,6)}$	$6.21^{(2,6)}$	$5.64^{(3,4)}$	$5.39^{(3,4)}$	$5.26^{(3,4)}$	$5.19^{(4,3)}$	$\mathbf{4.9}^{(4,3)}$	$\mathbf{4.73}^{(4,3)}$
FSBNDM(q,f)	$10.5^{(2,1)}$	$7.61^{(2,0)}$	$6.32^{(3,1)}$	$5.63^{(3,1)}$	$5.38^{(3,1)}$	$5.39^{(3,1)}$	$5.37^{(3,1)}$	$5.37^{(3,1)}$	$5.37^{(3,1)}$
FS-W(k)	$9.41^{(4)}$	$7.12^{(4)}$	$\mathbf{5.92}^{(4)}$	$5.47^{(4)}$	$5.35^{(4)}$	$5.30^{(6)}$	$5.24^{(4)}$	$5.21^{(6)}$	$5.24^{(4)}$
FSBNDM-W(k)	$10.4^{(2)}$	$7.35^{(4)}$	$5.96^{(4)}$	$\mathbf{5.44}^{(4)}$	$\mathbf{5.13}^{(4)}$	$\mathbf{5.15}^{(4)}$	$\mathbf{5.16}^{(4)}$	$5.13^{(4)}$	$5.12^{(4)}$
SBNDM-W(k)	$11.1^{(2)}$	$7.93^{(4)}$	$6.39^{(4)}$	$5.61^{(2)}$	$5.64^{(1)}$	$5.54^{(1)}$	$5.54^{(1)}$	$5.53^{(1)}$	$5.53^{(1)}$
TVSBS-W(k)	$9.85^{(4)}$	$8.13^{(4)}$	$6.79^{(6)}$	$5.94^{(6)}$	$5.50^{(6)}$	$5.36^{(6)}$	$5.21^{(4)}$	$5.02^{(6)}$	$4.89^{(6)}$

natural lang./m	2	4	8	16	32	64	128	256	512
EBOM	**9.69**	**7.35**	6.63	6.30	6.00	5.87	5.71	5.62	5.62
HASH(q)	$17.72^{(1)}$	$12.02^{(1)}$	$8.38^{(2)}$	$6.72^{(2)}$	$5.92^{(3)}$	$5.55^{(5)}$	$5.46^{(4)}$	$5.38^{(4)}$	$5.32^{(4)}$
QF(q,s)	-	$8.52^{(2,6)}$	$6.66^{(3,4)}$	$5.77^{(4,3)}$	$5.41^{(4,3)}$	$5.24^{(6,2)}$	$5.12^{(4,3)}$	$\mathbf{4.97}^{(4,3)}$	$\mathbf{4.7}^{(6,2)}$
FSBNDM(q,f)	$11.4^{(2,1)}$	$8.37^{(2,0)}$	$6.73^{(3,1)}$	$5.98^{(4,1)}$	$5.46^{(4,1)}$	$5.46^{(4,1)}$	$5.45^{(4,1)}$	$5.46^{(4,1)}$	$5.47^{(4,1)}$
FS-W(k)	$10.3^{(4)}$	$7.80^{(4)}$	$\mathbf{6.41}^{(4)}$	$5.80^{(4)}$	$5.38^{(4)}$	$5.23^{(4)}$	$\mathbf{5.08}^{(4)}$	$5.00^{(6)}$	$4.89^{(6)}$
FSBNDM-W(k)	$11.7^{(2)}$	$8.17^{(2)}$	$6.49^{(4)}$	$\mathbf{5.72}^{(4)}$	$\mathbf{5.21}^{(4)}$	$\mathbf{5.20}^{(4)}$	$5.23^{(4)}$	$5.22^{(4)}$	$5.23^{(4)}$
SBNDM-W(k)	$12.2^{(4)}$	$8.90^{(4)}$	$7.10^{(4)}$	$6.06^{(4)}$	$6.07^{(1)}$	$6.07^{(1)}$	$6.06^{(1)}$	$6.07^{(1)}$	$6.08^{(1)}$
TVSBS-W(k)	$10.6^{(4)}$	$8.66^{(4)}$	$7.04^{(6)}$	$6.02^{(6)}$	$5.52^{(6)}$	$5.36^{(6)}$	$5.18^{(4)}$	$5.02^{(6)}$	$4.82^{(6)}$

Table 3. Experimental results on a protein sequence (on the top) and on a natural language text buffer (on the bottom). Best results are boldfaced and underlined.

Table 2 shows the experimental results obtained on random text buffers over alphabets of size 16, 32, 64 and 128, while Table 3 shows experimental results performed on two real data problems and in particular a protein sequence and on a natural language text buffer. All text buffers are available on smart.

For each different algorithm we have reported only the best result obtained by its variants. Parameters of the variants which obtained the best running times are reported as apices. The reader can find a more detailed discussion about the performances of the different variants in the original papers [12, 15, 5].

From experimental results it tuns out that the new variants obtain very good results, especially in the cases of large alphabets ($\sigma \geq 32$). Specifically, best results are obtained by the FS-W(4) algorithm, in the case of small patterns, and by the FSBNDM-W(2) algorithm, in the case of moderately length patterns ($16 \leq m \leq 128$). Moreover for very long patterns ($m \geq 256$) multiple windows variants of the TVSBS algorithm turn out to be the fastest, even in the case of moderate large alphabets. In the case of small alphabets the EBOM and the QF algorithms still obtain the best results for short and long patterns, respectively.

5 Conclusions

We presented a general multiple sliding windows approach which could be applied to a wide family of comparison based and automata based algorithms. The new approach turns out to be simple to implement and leads to very fast algorithms in practical cases, especially in the case of large alphabets and natural language texts, as shown in our experimental results. It would be interesting to investigate further the application of this approach to other more effective solutions for the string matching problem, or to apply it to other related problems.

References

1. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
2. D. Cantone and S. Faro. Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm. *J. Autom. Lang. Comb.*, 10(5/6):589–608, 2005.
3. D. Cantone, S. Faro, and E. Giaquinta. Bit-(parallelism)^2: Getting to the next level of parallelism. In P. Boldi and L. Gargano, editors, *Fun with Algorithms*, LNCS 6099, 166–177. Springer-Verlag, Berlin, 2010.
4. B. Durian, J. Holub, H. Peltola, and J. Tarhio. Tuning BNDM with q-grams. In I. Finocchi and J. Hershberger, editors, *Workshop on Algorithm Engineering and Experiments*, 29–37, 2009.
5. B. Durian, H. Peltola, L. Salmela, and J. Tarhio. Bit-parallel search algorithms for long patterns. In P. Festa, editor, *Symposium on Experimental Algorithms*, LNCS 6049, 129–140, Springer-Verlag, Berlin, 2010.
6. S. Faro and T. Lecroq. Efficient variants of the Backward-Oracle-Matching algorithm. In J. Holub and J. Žďárek, editors, *Prague Stringology Conference 2008*, 146–160, Czech Technical University in Prague, Czech Republic, 2008.
7. S. Faro and T. Lecroq. An efficient matching algorithm for encoded DNA sequences and binary strings. In G. Kucherov and E. Ukkonen, editors, *Combinatorial Pattern Matching*, LNCS 5577, 106–115, Springer-Verlag, Berlin, 2009.
8. S. Faro and T. Lecroq. Smart: a string matching algorithm research tool. University of Catania and University of Rouen, 2011. `http://www.dmi.unict.it/~faro/smart/`.
9. J. Holub and B. Durian. Talk: Fast variants of bit parallel approach to suffix automata. In *The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation*, 2005. `http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf`.
10. R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
11. A. Hudaib, R. Al-Khalid, D. Suleiman, M. Itriq, and A. Al-Anani. A fast pattern matching algorithm with two sliding windows (TSW). *J. Comput. Sci.*, 4(5):393–401, 2008.
12. T. Lecroq. Fast exact string matching algorithms. *Inf. Process. Lett.*, 102(6):229–235, 2007.
13. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *Combinatorial Pattern Matching*, LNCS 1448, 14–33. Springer-Verlag, Berlin, 1998.
14. H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, editors, *String Processing and Information Retrieval*, LNCS 2857, 80–94, Springer-Verlag, Berlin, 2003.
15. H. Peltola and J. Tarhio. Variations of forward-SBNDM. In J. Holub and J. Žďárek, editors, *Prague Stringology Conference 2011*, 3–14, Czech Technical University in Prague, Czech Republic, 2011.
16. R. Thathoo, A. Virmani, S. S. Lakshmi, N. Balakrishnan, and K. Sekar. TVSBS: A fast exact pattern matching algorithm for biological sequences. *J. Indian Acad. Sci., Current Sci.*, 91(1):47–53, 2006.
17. S. Wu and U. Manber. Fast text searching: allowing errors. *Commun. ACM*, 35:83–91, October 1992.