A Fast Suffix Automata Based Algorithm for Exact Online String Matching

Simone Faro[†] and Thierry Lecroq[‡]

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

Abstract. Searching for all occurrences of a pattern in a text is a fundamental problem in computer science with applications in many other fields, like natural language processing, information retrieval and computational biology. Automata play a very important role in the design of efficient solutions for the exact string matching problem. In this paper we propose a new very simple solution which turns out to be very efficient in practical cases. It is based on a suitable factorization of the pattern and on a straightforward and light encoding of the suffix automaton. It turns out that on average the new technique leads to longer shift than that proposed by other known solutions which make use of suffix automata.

1 Introduction

The *string matching* problem consists in finding all the occurrences of a pattern P of length m in a text T of length n, both defined over an alphabet Σ of size σ. Automata play a very important role in the design of efficient string matching algorithms. For instance, the Knuth-Morris-Pratt algorithm [6] (KMP) was the first linear-time solution, whereas the Backward-DAWG-Matching algorithm [3] (BDM) reached the optimal $\mathcal{O}(n \log_\sigma(m)/m)$ lower bound time complexity on the average. Both the KMP and the BDM algorithms are based on finite automata; in particular, they respectively simulate a deterministic automaton for the language $\Sigma^* P$ and the deterministic suffix automaton of the reverse of P.

The efficiency of string matching algorithms depends on the underlying automaton used for recognizing the pattern P and on the encoding used for simulating it. The efficient simulation of nondeterministic automata can be performed by using the *bit parallelism* technique [1]. For instance the Shift-Or algorithm, presented in [1], simulates the nondeterministic version of the KMP automaton while a very fast BDM-like algorithm, (BNDM), based on the bit-parallel simulation of the nondeterministic suffix automaton, was presented in [8].

Specifically the bit-parallelism technique takes advantage of the intrinsic parallelism of the bitwise operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to w, where w is the number of bits in the computer word. However the correspondent encoding requires one bit per pattern symbol, for a total of $\lceil m/\omega \rceil$ computer words.

Thus, as long as a pattern fits in a computer word, bit-parallel algorithms are extremely fast, otherwise their performances degrades considerably as $\lceil m/\omega \rceil$ grows. Though there are a few techniques [9, 2, 4] to maintain good performance in the case of long patterns, such limitation is intrinsic.

In this paper we present a new algorithm based on the efficient simulation of a suffix automaton constructed on a substring of the pattern extracted after a suitable factorization. The new algorithm is based on a simple encoding of the underlying automaton and turns out to be very fast in most practical cases, as we show in our experimental results.

The paper is organized as follows. In Section 2 we briefly introduce the basic notions which we use along the paper. In Section 3 we review the previous results known in literature based on the simulation of the suffix automaton of the searched pattern. Then in Section 4 we present the new algorithm and some efficient variants of it. In Section 5 we compare the newly presented solutions with the suffix automata based algorithms known in literature. We draw our conclusions in Section 6.

2 Basic Notions and Definitions

Given a finite alphabet Σ, we denote by Σ^m, with $m \geq 0$, the set of strings of length m over Σ and put $\Sigma^* = \bigcup_{m \in \mathbb{N}} \Sigma^m$. We represent a string $P \in \Sigma^m$, also called an m-gram, as an array $P[0 \,..\, m-1]$ of characters of Σ and write $|P| = m$ (in particular, for $m = 0$ we obtain the empty string ε). Thus, $P[i]$ is the $(i+1)$-st character of P, for $0 \leqslant i < m$, and $P[i \,..\, j]$ is the substring of P contained between its $(i+1)$-st and the $(j+1)$-st characters, for $0 \leqslant i \leqslant j < m$. For any two strings P and P', we say that P' is a suffix of P if $P' = P[i \,..\, m-1]$, for some $0 \leqslant i < m$, and write $\mathit{Suff}(P)$ for the set of all suffixes of P. Similarly, P' is a prefix of P if $P' = P[0 \,..\, i]$, for some $0 \leqslant i < m$. In addition, we write $P \cdot P'$, or more simply PP', for the concatenation of P and P', and P^r for the reverse of the string P, i.e. $P^r = P[m-1]P[m-2] \cdots P[0]$.

For a string $P \in \Sigma^m$, the suffix automaton of P is an automaton which recognizes the language $\mathit{Suff}(P)$ of the suffixes of P.

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

3 Previous Efficient Suffix Automaton Based Solutions

In this section we present the known solutions for the online string matching problem which make use of the suffix automaton for searching for all occurrences of the pattern. Most of them are filtering based solutions, thus they use the suffix automaton for finding candidate occurrences of the pattern and then perform an additional verification phase based on a naive algorithm.

The Backward DAWG Matching Algorithm.

One of the first application of the suffix automaton to get optimal pattern matching algorithms on the average was presented in [3]. The algorithm which makes use of the suffix automaton of the reverse pattern is called Backward-DAWG-Matching algorithm (BDM). Such algorithm moves 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 algorithm 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 suffix of length m is recognized then an occurrence of the pattern was found.

The Backward Nondeterministic DAWG Matching Algorithm.

The BNDM algorithm [8] simulates the suffix automaton for P^r with the bit-parallelism technique, for a given string P of length m. The bit-parallel representation uses an array B of $|\Sigma|$ bit-vectors, each of size m, where the i-th bit of $B[c]$ is set iff $P[i] = c$, for $c \in \Sigma$, $0 \leqslant i < m$. Automaton configurations are then encoded as a bit-vector D of m bits, where each bit corresponds to a state of the suffix automaton (the initial state does not need to be represented, as it is always active). In this context the i-th bit of D is set iff the corresponding state is active. D is initialized to 1^m and the first transition on character c is implemented as $D \leftarrow (D \ \& \ B[c])$. Any subsequent transition on character c can be implemented as $D \leftarrow ((D \ll 1) \ \& \ B[c])$.

The BNDM algorithm works by shifting a window of length m over the text. Specifically, for each window alignment, it searches the pattern by scanning the current window backwards and updating the automaton configuration accordingly. Each time a suffix of P^r (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. A search 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.

When the pattern size m is larger than ω, 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 BNDM algorithm degrades considerably as $\lceil m/\omega \rceil$ grows. A common approach to overcome this problem consists in constructing an automaton for a substring of the pattern fitting in a single computer word, to filter possible candidate occurrences of the pattern. When an occurrence of the selected substring is found, a subsequent naive verification phase allows to establish whether this belongs to an occurrence of the whole pattern.

However, besides the costs of the additional verification phase, a drawback of this approach is that, in the case of the BNDM algorithm, the maximum possible shift length cannot exceed ω, which could be much smaller than m.

The Long BNDM Algorithm.

Peltola and Tarhio presented in [9] an efficient approach for simulating the suffix automaton using bit-parallelism in the case of long patterns. Specifically the algorithm (called LBNDM) works by partitioning the pattern 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 time for its verification phase grows proportionally to m/ω, so there is a threshold after which the performance of the algorithm degrades significantly.

The BNDM Algorithm with Extended Shift.

Durian *et al.* presented in [4] another efficient algorithm for simulating the suffix automaton in the case of long patterns. The algorithm is called BNDM with eXtended Shift (BXS). The idea is to cut the pattern into $\lceil m/\omega \rceil$ consecutive substrings of length w except for the rightmost piece which may be shorter. Then the substrings are superimposed getting a superimposed pattern of length ω. In each position of the superimposed pattern a character from any piece (in corresponding position) is accepted. Then a modified version of BNDM is used for searching consecutive occurrences of the superimposed pattern using bit vectors of length ω but still shifting the pattern by up to m positions. The main modification in the automaton simulation consists in moving the rightmost bit, when set, to the first position of the bit array, thus simulating a circular automaton. Like in the case of the LBNDM, algorithm the BXS algorithm works as a filter algorithm, thus an additional verification phase is needed when a candidate occurrence has been located.

The Factorized BNDM Algorithm.

Cantone *et al.* presented in [2] an alternative technique, still suitable for bit-parallelism, to encode the nondeterministic suffix automaton of a given string in a more compact way. Their encoding is based on factorizations of strings in which no character occurs more than once in any factor. It turns out that the nondeterministic automaton can be encoded with k bits, where k is the size of the factorization. Though in the worst case $k = m$, on the average k is much smaller than m, making it possible to encode large automata in a single or few computer words. As a consequence, their bit-parallel variant of the BNDM, called Factorized BNDM algorithm (F-BNDM) based on such approach tends to be faster in the case of sufficiently long patterns.

4 A New Fast Suffix Automaton Based Algorithm

The efficiency of suffix automata based algorithms for the exact string matching problem resides in two main features of the underlying automaton: the efficiency of the adopted encoding and the size of the automaton itself.

Regarding the first point it turns out that automata admitting simpler encoding turns out to be more efficient in practice. This is the case, for instance, of the automata which admit a bit parallel encoding. Moreover longer automata lead to larger shifts during the searching phase when a backward scan of the window is performed.

In this section we present a new algorithm for the online exact string matching problem based on the simulation of a suffix automaton constructed on the pattern P. The basic idea behind the new algorithm is straightforward but efficient. It consists in constructing the suffix automaton of a substring of the pattern in which each character is repeated at most once. This leads to a simple encoding and, by convenient alphabet transformations, to quite long automata.

The resulting algorithm is named Backward-SNR-DAWG-Matching (BSDM), where SNR is the acronym of *substring with no repetitions*. In what follows we describe separately the preprocessing and the searching phase of the algorithm.

The Preprocessing Phase

Given a pattern P, of length m, over an alphabet Σ of size σ, we say that a substring $S = P[i \, .. \, j]$ of P is a *substring with no repetitions* (SNR) if any character $c \in \Sigma$ appears at most once in S. It turns out trivially that $|S| \leq \min\{\sigma, m\}$. Moreover an SNR admits a suffix automaton where do not exist two states having incoming transitions labeled with the same character.

The preprocessing phase of the BSDM algorithm consists in finding the maximal SNR of the pattern, i.e. an SNR with the maximal length. In particular it finds a pair of integers value (s, ℓ), where $0 \leq s < m$ is the starting position of the maximal SNR of P, and $1 \leq \ell \leq m - s$ is the length of such a substring.

For instance, given the pattern $P =$ `abcabdcbabd`, we have that `abc`, `abdc`, `cba` are all SNR of P. The substring `abdc`, of length 4, is a maximal SNR of P.

In many practical cases the length of the maximal SNR is not large enough if compared with the size of the pattern. This happens especially for patterns over small alphabets, as in the case of genome sequences, or for patterns with characters occurring many times, as in the case of a natural language text.

In order to allow longer SNR it is convenient to use a condensed alphabet whose characters are obtained by combining groups of q characters, for a fixed value q. A hash function $hash : \Sigma^q \leftarrow \{0, \ldots, \text{MAX} - 1\}$ can be used for combining the group of characters, for a fixed constant value MAX. Thus a new condensed pattern P_q of length $m - q + 1$, over the alphabet $\{0, \ldots, \text{MAX} - 1\}$, is obtained from P. Specifically we have $P_q[i \, .. \, j] = hash(P[i] \cdots P[i + q - 1]) \cdots hash(P[j] \cdots P[j + q - 1])$ for $0 \leq i, j \leq m - q$, where $P_q = P_q[0 \, .. \, m - q]$. The maximal SNR is then computed on P_q to get a longer suffix automaton.

q/m	2	4	8	16	32	64	128	256	512	1024	2048	4096
1	1.72	2.62	3.20	3.64	3.89	3.99	4.00	4.00	4.00	4.00	4.00	4.00
2	1.00	2.86	5.45	7.61	9.19	10.41	11.23	11.97	12.64	13.09	13.21	13.27
4	-	1.00	4.94	12.33	22.91	32.75	39.89	45.12	50.84	54.29	57.17	59.82
6	-	-	3.00	10.81	24.56	42.19	55.69	66.31	74.35	82.48	88.50	97.82
8	-	-	1.00	8.98	24.50	51.55	88.03	116.33	140.82	163.20	175.24	183.42

q/m	2	4	8	16	32	64	128	256	512	1024	2048	4096
1	1.91	3.46	5.43	6.98	8.20	9.27	10.08	10.95	11.70	12.27	12.91	13.69
2	1.00	2.96	6.53	12.03	17.28	21.04	24.24	27.32	30.02	32.30	34.84	36.61
4	-	1.00	4.99	12.84	27.29	49.85	71.44	88.44	99.13	111.73	125.03	132.34
6	-	-	2.99	10.85	25.03	45.34	62.62	73.88	82.87	90.78	99.09	106.52
8	-	-	1.00	8.99	24.62	53.25	92.54	126.52	152.86	172.15	195.92	217.41

q/m	2	4	8	16	32	64	128	256	512	1024	2048	4096
1	1.99	3.81	6.25	7.83	8.96	9.83	10.46	11.07	11.51	12.18	12.91	14.42
2	1.00	2.99	6.84	12.98	19.01	23.30	26.77	29.68	32.79	35.38	37.80	40.03
4	-	1.00	5.00	12.94	26.86	43.01	55.50	64.67	72.94	79.03	87.22	97.85
6	-	-	3.00	10.99	26.42	50.79	73.93	92.90	104.13	115.86	132.07	148.79
8	-	-	1.00	9.00	24.83	53.98	96.14	128.96	152.85	175.50	189.52	203.14

Table 1. The average length of the maximal SNR in patterns randomly extracted from a genome sequence (on top), a protein sequence (in the middle) and a natural language text (on bottom). The SNR have been computed using condensed alphabets on q characters, where q ranges from 1 to 8.

For instance if $q = 3$ the pattern $P =$ `abcabdcb` is condensed in a new pattern $P_3 = hash(\texttt{abc}) \cdot hash(\texttt{bca}) \cdot hash(\texttt{cab}) \cdot hash(\texttt{abd}) \cdot hash(\texttt{bdc}) \cdot hash(\texttt{dcb})$.

The size MAX of the new condensed alphabet depends on the available memory and on the size of the original alphabet Σ. An efficient method for computing a condensed alphabet was introduced by Wu and Manber [10], and then adopted also in [7]. It computes the shift value by using a shift-and-addition procedure and in particular $hash(c_1, c_2, \ldots, c_q) = \left(\sum_{i=1}^{q}(c_i \ll sh^{q-i})\right) \bmod \text{MAX}$ where $c_i \in \Sigma$ for $i = 1, \ldots, q$. The value of the shift sh depends on MAX and q.

Table 1 shows the average length of the maximal SNR in patterns randomly extracted from a genome sequence, a protein sequence and a natural language text, for different values of q and m, and with $\text{MAX} = 2^{16}$. When $1 \leq q \leq 4$ we use the value $sh = 2$ for computing the hash value, while we use $sh = 1$ when $q > 4$. It turns out that the length of the maximal SNR, though quite less than m in most cases, is quite larger than the size of a computer word (which typically is 32 or 64). This leads to larger shift in a suffix automata based algorithm.

The procedure which computes the maximal SNR of P using a condensed alphabet is shown in Fig. 1. The procedure iterates two indices, i and j along the pattern, starting from the leftmost character. At each iteration the value of i is incremented by one position, and the value of j is incremented in order to make the substring $P_q[j \,..\, i]$ an SNR of P_q. At the end of each iteration, if the substring $P_q[j \,..\, i]$ is longer than the temporary maximal SNR found in the previous iterations, then the values of s and ℓ are updated accordingly.

The time complexity of the resulting procedure is $\mathcal{O}(m)$ while the space required is $\mathcal{O}(\text{MAX})$.

```
HASH($P, i, q, b$)
    $c \leftarrow P[i]$
    for $j \leftarrow i+1$ to $i+q-1$ do
        $c \leftarrow (c \ll b) + P[j]$
    return $c$

MAXSNR($P, m, q, b$)
    for $c \leftarrow 0$ to MAX $- 1$ do $\delta(c) \leftarrow$ FALSE
    $s \leftarrow \ell \leftarrow 0$
    $j \leftarrow 0$
    for $i \leftarrow 0$ to $m - q$ do
        $c \leftarrow$ HASH($P, i, q, b$)
        if $\delta(c)$ then
            $d \leftarrow$ HASH($P, j, q, b$)
            while $d \neq c$ do
                $\delta(d) \leftarrow$ FALSE
                $j \leftarrow j + 1$
                $d \leftarrow$ HASH($P, j, q, b$)
            $\delta(d) \leftarrow$ FALSE
            $j \leftarrow j + 1$
        $\delta(c) \leftarrow$ TRUE
        if $\ell < i - j + 1$ then
            $\ell \leftarrow i - j + 1$
            $s \leftarrow j$
    return $(s, \ell)$

POSITIONS($P, s, \ell, q, b$)
    for $c \leftarrow 0$ to MAX $- 1$ do $pos(c) = -1$
    for $i \leftarrow 0$ to $\ell - 1$ do
        $c \leftarrow$ HASH($P, s+i, q, b$)
        $pos(c) \leftarrow i$
    return $pos$

BSDM($P, m, T, n, q, b$)
    $(s, l) \leftarrow$ MAXSNR($P, m, q, b$)
    $pos \leftarrow$ POSITIONS($P, s, \ell, q, b$)
    $j \leftarrow \ell - 1$
    $r \leftarrow s + \ell$
    while $j < n$ do
        $c \leftarrow$ HASH($T, j, q, b$)
        $i \leftarrow pos(c)$
        if $i \geq 0$ then
            $k \leftarrow 1$
            while $k \leq i$ and $P[s+i-k] = T[j-k]$ do
                $k \leftarrow k + 1$
            if $k > i$ then
                if $k = \ell$ then
                    if $P = T[j-r+1 \,..\, j-r+m]$
                    then output $(j - s - \ell + 1)$
                else $j \leftarrow j - k$
        $j \leftarrow j + \ell$
```

Fig. 1. The pseudocode of the algorithm BSDM and its auxiliary procedures. The input parameters q and b represent, respectively, the size of the group of characters used in the condensed alphabet and the value sh used for computing the hash function.

The Searching Phase

Let P be a pattern of length m over an alphabet Σ of size σ, and let P_q be the corresponding pattern, of length $m-q+1$, obtained from P by using a condensed alphabet. Let s and ℓ be, respectively, the starting position and the length of the maximal SNR in the P_q. During the searching phase the BSDM algorithm works using a filtering method. Specifically it first searches for all occurrences of the substring $P_q[s \,..\, s+\ell-1]$ in the text. For this purpose the text is also scanned by using a condensed alphabet. When an occurrence is found, ending at position j of the text, the algorithm naively checks for the whole occurrence of the pattern, i.e. if $P = T[j-s-\ell+1 \,..\, j-s-\ell+m]$.

A function $pos : \{0, \ldots, \text{MAX}-1\} \rightarrow \{0, \ldots, \ell-1\}$ is defined for all characters in the condensed alphabet. In particular for each $0 \leqslant c < \text{MAX}$ the value of $pos(c)$ is defined as the relative position in $P_q[s \,..\, s+\ell-1]$ where the character c appears, if such position exists. Otherwise $pos(c)$ is set to -1. More formally $pos(c) = i$ if there exists $i < \ell$ such that $P_q[s+i] = c$ and -1 otherwise, for $0 \leqslant c < \text{MAX}$. Observe that if position i exists such that $i < \ell$ and $P_q[s+i] = c$, then it is unique, since the substring $P_q[s \,..\, s+\ell-1]$ has no repetitions of characters. The function pos is computed in $\mathcal{O}(m)$ time and $\mathcal{O}(\text{MAX})$ space by using the procedure POSITION shown in Fig. 1.

The pos function defined above is then used during the searching phase for simulating the suffix automaton of the maximal SNR of the pattern. Observe that, since there is no repetition of characters, at most a single state could be active at any time. Thus the configuration of the suffix automaton can be encoded by using a single integer value of $\lceil \log \ell \rceil$ bits, which simply indicates the active state of the automaton, if any. Otherwise it is set to -1.

The algorithm works by sliding a window on length $\ell + q - 1$ along the text. At each attempt a condensed character c is computed from the rightmost q characters of the window. If c is not present in the maximal SNR of the pattern, i.e. if $pos(c) = -1$, then the window is advanced ℓ positions to the right. Otherwise (if $pos(c) \geqslant 0$) the position i where character c appears in the maximal SNR is computed by setting $i = pos(c)$. Then the text and the pattern are compared, character by character, from positions $s + i$ and $j - s - \ell + i$, respectively, until a mismatch occurs or until position s in the pattern is passed.

If a mismatch occurs, no prefix of the substring has been recognized and the window is simply advanced ℓ positions to the right.

Otherwise, if position s in the pattern is passed, then a prefix of the substring has been recognized. If we read exactly ℓ characters in T then an occurrence of the substring has been found and a naive verification follows in order to check the occurrence of the whole pattern. If we read less than ℓ characters we recognized a prefix of the substring and the window is advanced in order to align he character of position s in the pattern with the starting position of the recognized prefix in the text. The searching phase of the algorithm is shown in Fig. 1. It has a $\mathcal{O}(nm)$ worst case time complexity and requires $O(\text{MAX})$ space.

5 Experimental Results

In this section we briefly present experimental evaluations in order to understand the performances of the newly presented algorithm and to compare it against the best on suffix (factor) automata based string matching algorithms. In particular we tested the following algorithms: the Backward-DAWG-Matching algorithm [3] (BDM); the Backward-Nondeterministic-DAWG-Matching algorithm [8] (BNDM); the Simplified version of the BNDM algorithm [9] (SBNDM); the BNDM for algorithm long patterns [9] (LBNDM); the Factorized BNDM algorithm [2] (F-BNDM); the BNDM algorithm with Extended Shift [4] (BXS); and the new Backward-SNR-DAWG-Matching algorithm using condensed alphabets with groups of q characters, with $1 \leq q \leq 8$ (BSDM$_q$)

All the algorithms listed above could be enhanced by using fast loops, q-grams and other efficient techniques. However this type of code tuning goes beyond the scope of this paper. Thus we tested only the original versions of the algorithms.

All algorithms have been implemented in the C programming language and have been tested using the SMART tool [5]. The experiments were executed locally on an MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3. Algorithms have been compared in terms of running times, including any preprocessing time.

For the evaluation we use a genome sequence, a protein sequence and a natural language text (English language), all sequences of 4MB. The sequences

Experimental results on a genome sequence												
m	2	4	8	16	32	64	128	256	512	1024	2048	4096
BDM	21.04	14.66	9.90	7.40	5.94	5.15	4.79	4.68	4.90	5.42	7.22	10.59
BNDM	19.52	12.56	8.92	6.72	5.50	5.55	5.51	5.47	5.58	5.49	5.50	5.49
SBNDM	12.25	9.13	7.66	6.27	5.14	5.14	5.12	5.13	5.12	5.13	5.13	5.14
BXS	19.57	13.88	9.27	6.88	5.47	5.15	4.99	5.52	523.2	-	-	-
F-BNDM	15.49	10.74	8.71	7.09	5.78	5.10	5.03	5.03	5.02	5.03	5.05	5.05
LBNDM	27.62	15.24	9.79	7.28	5.80	5.38	5.36	8.45	26.93	25.28	22.50	20.67
BSDM	20.94	17.49	14.42	13.05	11.99	11.52	11.55	11.39	11.42	11.38	11.46	11.50
BSDM$_2$	**11.43**	9.26	8.66	8.31	7.82	7.44	7.15	6.89	6.60	6.51	6.37	6.24
BSDM$_3$	-	**7.44**	5.92	5.57	5.43	5.38	5.33	5.31	5.28	5.30	5.28	5.28
BSDM$_4$	-	9.67	**5.61**	**4.99**	4.79	4.73	4.66	4.64	4.63	4.63	4.66	4.66
BSDM$_5$	-	-	5.99	5.00	4.77	4.66	4.61	4.58	4.57	4.56	4.58	4.58
BSDM$_6$	-	-	6.86	5.09	**4.69**	4.58	4.53	4.49	4.50	4.47	4.50	4.50
BSDM$_7$	-	-	8.81	5.25	4.71	**4.55**	**4.51**	**4.47**	**4.45**	**4.47**	**4.47**	**4.49**
BSDM$_8$	-	-	14.88	5.57	4.80	4.56	**4.51**	4.50	4.48	4.48	4.49	4.50

Experimental results on a protein sequence												
m	2	4	8	16	32	64	128	256	512	1024	2048	4096
BDM	9.82	8.20	7.05	5.80	4.91	4.59	4.53	4.57	4.73	5.30	7.14	10.60
BNDM	9.27	7.67	6.74	5.61	4.81	4.83	4.80	4.80	4.81	4.80	4.82	4.83
SBNDM	9.25	**5.93**	**4.96**	**4.59**	**4.41**	4.57	4.57	4.57	4.58	4.57	4.62	4.58
BXS	8.41	7.19	6.41	5.45	4.69	**4.53**	**4.39**	**4.29**	**4.27**	**4.17**	**4.28**	105.3
F-BNDM	11.94	8.06	6.22	5.32	4.91	4.79	4.63	4.64	4.62	4.65	4.64	4.65
LBNDM	19.66	12.60	8.84	6.51	5.79	4.88	4.54	4.40	4.34	4.46	6.20	10.44
BSDM	8.37	7.58	7.15	6.89	6.63	6.37	6.14	5.92	5.71	5.56	5.47	5.37
BSDM$_2$	**8.29**	6.04	5.44	5.15	5.07	4.99	4.99	4.97	4.95	4.93	4.94	4.95
BSDM$_3$	-	6.58	5.25	4.85	4.71	4.64	4.62	4.59	4.59	4.58	4.60	4.60
BSDM$_4$	-	9.71	5.49	4.89	4.68	4.59	4.56	4.53	4.52	4.52	4.50	**4.53**
BSDM$_5$	-	-	6.04	5.07	4.79	4.68	4.65	4.61	4.61	4.61	4.62	4.64
BSDM$_6$	-	-	7.02	5.19	4.79	4.64	4.60	4.58	4.59	4.57	4.58	4.61
BSDM$_7$	-	-	9.02	5.38	4.82	4.64	4.62	4.58	4.58	4.60	4.58	4.59
BSDM$_8$	-	-	15.11	5.68	4.94	4.70	4.64	4.63	4.62	4.61	4.59	4.58

Experimental results on a natural language text												
m	2	4	8	16	32	64	128	256	512	1024	2048	4096
BDM	10.50	9.27	7.89	6.29	5.33	4.93	4.73	4.99	4.98	5.51	7.34	10.82
BNDM	10.02	8.74	7.50	6.06	5.20	5.25	5.23	5.25	5.23	5.25	5.25	5.26
SBNDM	9.68	6.39	5.47	5.01	4.76	4.99	4.99	4.99	4.99	4.97	4.98	4.98
BXS	9.12	8.25	7.20	5.91	5.06	4.76	**4.50**	**4.35**	**4.27**	**4.09**	**3.92**	**3.90**
F-BNDM	12.36	8.45	6.64	5.75	5.30	5.04	4.72	4.67	4.66	4.67	4.67	4.67
LBNDM	20.36	13.38	9.38	6.83	5.56	4.99	4.63	4.44	4.35	4.38	4.62	5.69
BSDM	8.90	8.35	7.72	7.15	6.71	6.43	6.16	6.01	5.85	5.79	5.69	5.61
BSDM$_2$	**8.41**	**6.24**	5.62	5.37	5.27	5.23	5.18	5.14	5.11	5.09	5.08	5.08
BSDM$_3$	-	6.76	**5.40**	5.00	4.85	4.79	4.74	4.71	4.69	4.67	4.70	4.71
BSDM$_4$	-	9.88	5.62	**4.95**	4.76	4.65	4.62	4.61	4.61	4.56	4.57	4.62
BSDM$_5$	-	-	6.02	5.00	**4.75**	4.65	4.62	4.59	4.59	4.55	4.60	4.63
BSDM$_6$	-	-	7.05	5.16	4.78	**4.64**	4.59	4.57	4.58	4.54	4.58	4.60
BSDM$_7$	-	-	9.26	5.41	4.84	4.66	4.60	4.56	4.59	4.56	4.57	4.58
BSDM$_8$	-	-	16.07	5.80	4.96	4.72	4.67	4.62	4.60	4.54	4.53	4.55

Table 2. Experimental results on a genome sequence (on top), a protein sequence (in the middle) and natural language text (on bottom).

are provided by the SMART research tool. In all cases the patterns were randomly extracted from the text and the value m was made ranging from 2 to 4096. For each case we reported the mean over the running times of 500 runs.

Table 2 shows experimental results on the three different sequences. Running times are expressed in thousands of seconds. Best times have been boldfaced and underlined. It turns out that the BSDM algorithm is really competitive against the most efficient algorithms which make use of suffix automata. The versions based on condensed characters obtain in many cases the best results, especially in the case of the genome sequence and of the natural language text. Otherwise SBNDM and BXS obtain the best times for short and long patterns, respectively.

6 Conclusions and Future Works

We presented a new simple and efficient algorithm, named Backward-SNR-DAWG-Matching, based on suffix automata. It uses a very simple encoding of the automaton, consisting in a single integer value, but obtains larger shift on average than that obtained by algorithms based on the bit parallel encoding.

In our future works we intend to tune the algorithm in order to make it competitive with the most efficient algorithms in practical cases. This includes the use of fast loops, q-grams and most efficient hash functions for implementing the condensed alphabets. We would also investigate the possibility of tuning the hash function in order to reflect only the size of the set of characters appearing in the pattern. Another possible line for enhancing the performances of the algorithm is to make it recognize factors instead of suffixes.

References

1. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
2. D. Cantone, S. Faro, and E. Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In *CPM*, LNCS 6129, pages 288–298, 2010.
3. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.
4. B. Durian, H. Peltola, L. Salmela, and J. Tarhio. Bit-parallel search algorithms for long patterns. In *SEA*, LNCS 6049, pages 129–140, 2010.
5. S. Faro and T. Lecroq. Smart: a string matching algorithm research tool. Univ. of Catania and Univ. of Rouen, 2011. `http://www.dmi.unict.it/~faro/smart/`.
6. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
7. T. Lecroq. Fast exact string matching algorithms. *Inf. Process. Lett.*, 102(6):229–235, 2007.
8. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *CPM*, LNCS 1448, pages 14–33, 1998.
9. H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In *SPIRE*, LNCS 2857, pages 80–94, 2003.
10. S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Report TR-94-17, Depart. of Computer Science, University of Arizona, Tucson, AZ, 1994.