Pruned BNDM: Extending the Bit-Parallel Suffix Automaton to Long Strings* **

Simone Faro and Stefano Scafiti

University of Catania, Department of Mathematics and Computer Science, Italy
{simone.faro,stefano.scafiti}@unict.it

Abstract. Automata have always played a very important role in the design of efficient solutions for the exact string matching problem. Among these, a special mention is deserved by those algorithms exploiting the bit-parallelism technique to efficiently simulate the suffix automaton of a string. However, the bit-parallel encoding requires one bit for each character, and thus performance degrade quickly as the length of searched pattern grows beyond the machine word size w. In this paper, we present a novel technique for exploiting the bit-parallel representation of a non-deterministic suffix automaton also in the case of strings of length $m > w$. Our approach consists in searching for a *pruned* version of the original pattern, whose automaton can be represented with a reduced number of bits, thus allowing to retain the performance of the original approach also in the case patterns exceeding the word size. Experimental results show that, in the case of very long patterns, our method scales better than existing approaches.

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. It is a basic problem of computer science that has been studied for more than 40 years [6]. The first linear-time solution to the problem was given by the KnuthMorrisPratt algorithm (KMP) [9], whereas the BoyerMoore (BM) algorithm provided the first sublinear solution on average. Subsequently, the BDM algorithm reached the optimal $\mathcal{O}(n \log_\sigma(m)/m)$ time complexity on the average [3].

Automata play a very important role in the design of efficient string matching algorithms. For instance both the KMP and the BDM algorithms are based on finite automata; in particular, they simulate, respectively, a deterministic automaton for the language $\Sigma^\star P$ and a deterministic suffix automaton for the language of the suffixes of P. The efficiency of such solutions is strictly influenced by the encoding used for simulating the underlying automata.

* This work has been supported by G.N.C.S., Istituto Nazionale di Alta Matematica Francesco Severi and by Programma Ricerca di Ateneo UNICT 2020-22 linea 2

A successful technique which has been extensively used for the efficient simulation of non-deterministic automata is the *bit parallelism* [1]. Such approach is at the base, for instance, of the well known Shift-Or [1] and BNDM [10] algorithms. The first is based on the simulation of the non-deterministic version of the KMP automaton, while the second is a very fast variant of the BDM algorithm, based on the bit-parallel simulation of the non-deterministic suffix automaton.

Specifically, the bit parallel encoding 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 ω, where ω is the number of bits in the computer word. However, one bit per pattern symbol is required for representing the states of the full automaton, for a total of $\lceil m/\omega \rceil$ words. Thus, as long as a pattern fits in a computer word, bit-parallel algorithms are extremely fast, otherwise their performances degrade considerably as $\lceil m/\omega \rceil$ grows. Although such limitation is intrinsic, several techniques have been developed which retain good performance also in the case of long patterns [11,2,4,5].

In this paper, we introduce a novel approach for the the bit-parallel simulation of the suffix automaton in the case of long patterns. The idea is to first search for a *pruned* pattern, and to subsequently verify each occurrence of the original pattern. The pruned pattern is constructed by preserving only the occurrences of a single *pivot* character c; remaining characters are implicitly associated to the special wildcard symbol "$*$" and are allowed to match any character of the alphabet, except from c. It turns out that the suffix automaton of a pruned pattern can be encoded using k bits, where k is the number of occurrences of c in P. Though in the worst case $k = m$, on the average it is much smaller than m, and, when the size of the alphabet is large enough, it is also smaller than ω even for very long patterns.

From our experimental results it turns out that, under suitable conditions, our approach is able to represent patterns which far exceed the word size w.

The paper is organized as follows. Section 2 presents the basic notions which we use along the paper. In Section 3 we review the previous solutions known in literature which make use of the bit-parallelism approach to efficiently encode the non-deterministic suffix automaton of a string. In Section 4, we present and describe in details the new algorithm. In Section 5 we perform an experimental evaluation comparing the new algorithm with existing solutions. Finally, 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 regard a string $P \in \Sigma^m$ as an array $P[0..m-1]$ of characters and denote its length by $|P| = m$ (in particular, denoting by ϵ the empy string, we have $|\epsilon| = 0$). Thus, $P[i]$ is the i-th character of P, for $0 \leq i < m$, and $P[i..j]$ is the substring of P starting at the i-th position

and ending at the j-th position, for $0 \leq i \leq j < m$. Moreover, for each character $c \in \Sigma$, we denote by $\rho_{_P}(c)$ the number of its occurrences in P (we refer to such value as simply $\rho(c)$ when the pattern P is clear from the context). For any two strings P and P', we say that P' is a suffix of P if $P' = P[i..m-1]$ for some $0 \leq i < m$ 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 $0 \leq i < m$. 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. $P^r = P[m-1]P[m-2]...P[0]$.

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 Related Results

In this section we review existing solutions for the exact online string matching problem which make use of a suffix automaton for searching for all occurrences of a pattern in a text, focusing on those algorithms which implement specific solutions to encode the automata for long patterns. Most of them are filtering based solutions, which means that they use the suffix automaton, constructed over an approximate version of the input string, for finding candidate occurrences of the pattern that must subsequently verified for an exact occurrence by running an additional verification phase.

All algorithms reviewed in this section are variants of the well known Backward-DAWG-Matching algorithm (BDM) algorithm [3], one of the first application of the suffix automaton to get optimal pattern matching algorithms on the average. 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 itself was found.

However, one of the side effects of the BDM algorithm lies in the use of the deterministic variant of the suffix automaton since the workload required to manage the individual transitions may be not negligible and, although its construction is linear in the size of the string, the proportionality factor hidden in the asymptotic notation is particularly high, making its construction prohibitive in the case of long patterns [6].

The BNDM algorithm [10] simulates the suffix automaton for P^r by bit-parallelism. The bit-parallel representation of a suffix automaton 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 $c \in \Sigma$, $0 \leq i < m$. Automaton configurations s_i 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 $D \leftarrow ((D \ll 1) \,\&\, B[c])$. 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).

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.

Peltola and Tarhio presented in [11] an efficient approach for simulating the suffix automaton using bit-parallelism for long patterns. Specifically the algorithm (called LBNDM) works by partitioning the pattern in $\lfloor m/k \rfloor$ consecutive substrings, each of $k = \lfloor (m-1)/\omega \rfloor + 1$ 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.

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 algo-

rithm works as a filter algorithm, thus an additional verification phase is needed when a candidate occurrence has been located.

Cantone *et al.* presented in [2] an alternative technique, still suitable for bit-parallelism, to encode the non-deterministic 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 non-deterministic 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, the resulting Factorized BNDM algorithm (FBNDM) tends to be faster in the case of sufficiently long patterns.

4 The Pruned BNDM Algorithm

As we noticed in the previous section, the efficiency of an algorithm simulating the non-deterministic suffix automaton by bit-parallelism is heavily influenced by the length of the pattern and by the size of the resulting automaton: on the one hand, as we pointed out, the performance of such solutions degrade as m grows; on the other hand, automata constructed over longer patterns lead to larger shifts during the searching phase when a backward scan of the window is performed. Thus the need for efficient bit-parallel encoding able to keep as low as possible the number of words involved in the encoding and able to preserve, at the same time, the length of the pattern.

In this section, we present a new algorithm for the online exact string matching problem based on a suffix automaton constructed over an approximate version of the pattern P, which we simply call *pruned pattern*, where some specifically selected characters are replaced with don't care symbols. We will then show how to efficiently simulate the suffix automaton constructed over the pruned version of the pattern using bit-parallelism.

4.1 The Pruned Version of a Pattern

Let P be a pattern of size m and let T be a text of size n, both strings over a common alphabet Σ of size σ. In addition let $c \in \Sigma$ be a character of the alphabet occurring in P which we refer as the *pivot* character. A *pruned version* of P over the pivot character c is a string P_c obtained by preserving in P all the occurrences of c, while the remaining positions are allowed to match any character belonging to $\Sigma \setminus \{c\}$. In other words the pruned pattern P_c is obtained from P by replacing any character in $\Sigma \setminus \{c\}$ by a don't care symbol.

More formally, for each $c \in \Sigma$, the pruned string P_c is a string of length m defined over the alphabet $\Sigma_c = \{c, \star\}$ where, for $i = 0, 1, .., m-1$, $P_c[i]$ is set to:

$$P_c[i] = \begin{cases} c & \text{if } P[i] = c \\ \star & \text{otherwise} \end{cases}$$

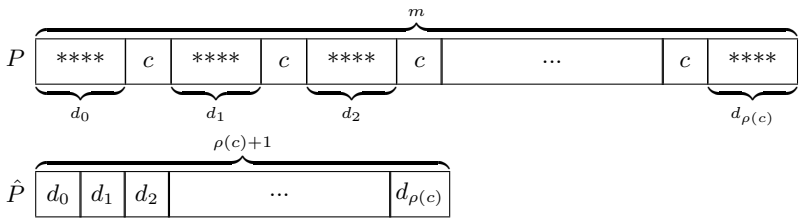

Fig. 1. The pruned version P_c, for a given pattern P, over a generic character c, and its implicit encoding. Here $\rho(c)$ is the absolute frequency of the pivot character c in P. Then the pruned string P_c of a string P is encoded as a sequence, $\langle d_0, d_1, ..., d_{\rho(c)}\rangle$, of length $\rho(c)+1$ over the alphabet $\Sigma_P = \{0, 1, 2, \ldots, m-1\}$, where each element of P_c represents the number of consecutive $\star$ symbols between two successive occurrences of the pivot character c, or located at the two extremities of the string.

For instance, if we assume that $P = abbacbbcac$ is a pattern of length $m = 10$ over the alphabet $\Sigma = \{a, b, c\}$ and that a is the pivot character, then we have that $P_a = a \star\star a \star\star\star\star a\star$, where $\star$ is the don't care symbol. Similarly we have $P_b = \star bb \star\star bb \star\star\star$.

The string matching problem allowing for don't care symbols is a well known approximate variant of the exact matching problem [8,12], also known as string matching on indeterminate strings. It is also well known that the bit parallel simulation of the suffix automaton of an indeterminate pattern can be easily constructed by allowing states corresponding to don't care characters to be activated by any character in the alphabet [1]. However the resulting automaton has a number of states equal to the number of characters in the pattern, inheriting the same problems as any other solution based on this technique. Here we show how the suffix automaton of a pruned pattern can be simulated using a number of bits proportional to the occurrences of the pivot character, leading to a filtration algorithm which may be particularly efficient for very long patterns.

Specifically, let $\rho(c)$ be the absolute frequency of the pivot character c in P. Then the pruned string P_c of a string P can be encoded as a sequence, $\langle d_0, d_1, ..., d_{\rho(c)}\rangle$, of length $\rho(c)+1$ over the alphabet $\Sigma_P = \{0, 1, 2, \ldots, m-1\}$, where each element of P_c represents the number of consecutive $\star$ symbols between two successive occurrences of the pivot character c, or located at the two extremities of the string.

More formally, let $\langle p_0, p_1, \ldots, p_{\rho(c)-1}\rangle$ be the sequence of all positions in P where the pivot character occurs, with $0 \leq p_1$, $p_{\rho(c)-1} < m$ and $p_{i-1} < p_i$, for $0 < i < \rho(c)$. Then we have, for $0 \leq i \leq \rho(c)$:

$$d_i = \begin{cases} p_0 & \text{if } i = 0 \\ p_{i+1} - p_i - 1 & \text{if } 0 < i < \rho(c) \\ m - p_{\rho(c)-1} - 1 & \text{if } i = \rho(c) \end{cases}$$

We refer to such a representation of the pruned string P_c as its *implicit encoding* and we denote it as $\hat{P}_c$ (see Figure 1). It trivially turns out that $d_i \leq m$ for $0 \leq i \leq k$. More precisely we have

$$m = \rho(c) + \sum_{i=0}^{\rho(c)} d_i$$

For instance, given the string $P =$ `banana`, then $P_a =$ `*a*a*a`, $P_b =$ `b*****`, $P_n =$ `**n*n*`, while $\hat{P}_a = \langle 1, 1, 1, 0 \rangle$, $\hat{P}_b = \langle 0, 5 \rangle$ and $\hat{P}_n = \langle 2, 1, 1 \rangle$.

In the next sections we describe the preprocessing and the searching phases of our new algorithm, which we call Pruned BNDM (PBNDM) algorithm, and which solves the exact string matching problem by making use of the suffix automaton of the pruned pattern.

4.2 The Preprocessing Phase

During the preprocessing phase of the PBNDM algorithm, a character c occurring in P is elected to be the pivot character. Since such choice is arbitrary, the pivot character is selected as the character with maximum absolute frequency not exceeding the word size w, if any. If such choice is not possible we truncate the pattern at its longest prefix that contains at least one character with an absolute frequency not exceeding the word size w. It is easy to observe that the selection of the pivot character can be performed in $\mathcal{O}(m)$ time, by computing the frequencies of all the characters appearing in P. Without loss in generality we can assume that such pivot character can be selected on the pattern P.

Let $\hat{P}_c = \langle d_0, d_1, ..., d_{\rho(c)} \rangle$ the implicit encoding of P_c. It is a string of length $\rho(c) + 1$ over the alphabet $\hat{\Sigma} = \{1, 2, 3, \ldots, m\}$ of size $m + 1$.

The bit-parallel representation of the suffix automaton of $\hat{P}_c^r$ is computed by means of an array B of $m + 1$ bit-vectors, each of size $\rho(c) + 1$, where the i-th bit of $B[d]$ is set iff $\hat{P}[i] = d$, for $0 \leq d \leq m$ and $0 \leq i \leq \rho(c)$. However, since the last transition of the automaton is allowed for any value greater than or equal to d_0, the first bit of each bit-vector in B is set for any value $d \leq d_0$.

More formally, for $0 \leq i \leq \rho(c)$ and $0 \leq d \leq m$, $B[d][i]$ is defined as follows:

$$B[d][i] = \begin{cases} 1 & \text{if } (i = 0 \text{ and } d \geq d_0) \text{ or } (i > 0 \text{ and } d = d_i), \\ 0 & \text{otherwise} \end{cases}$$

A separate discussion should be made for the first transition made on the automaton. Since it is admitted that the first transition can start from any position of the pattern it is necessary to allow that at the first transition each i-th state can be activated by values lower than or equal to d_i. For this purpose an auxiliary set of $m+1$ bit-vectors is defined, called S, which is used for the simulation of the first transition on the automaton.

More formally, for $0 \leq i \leq \rho(c)$ and $0 \leq d \leq m$, $S[d][i]$ is defined as follows:

$$S[d][i] = \begin{cases} 1 & \text{if } (i = 0 \text{ and } d \geq d_0) \text{ or } (i > 0 \text{ and } d \leq d_i), \\ 0 & \text{otherwise} \end{cases}$$

```
READ(S, i, l, c)
1. j ← i
2. while j ≥ l and S[j] ≠ c do
4.       j ← j − 1
6. return (i − j, j)

PREPROCESS(P, m, c, k)
   Δ Initialize bit-vectors B and S
1. for i ← 0 to m do
2.       B[i] ← 0
3.       S[i] ← 0
   Δ Compute B, d_0 and d_max
4. s ← 1
5. d_0 ← −1
6. d_max ← −1
7. i ← m − 1
8. while i ≥ 0 do
9.       (d, i) ← READ(P, i, 0, c)
10.      B[d] ← B[d] | s
11.      s ← s ≪ 1
12.      i ← i − 1
13.      if d_0 = −1 then d_0 ← d
14.      d_max ← MAX(d, d_max)
   Δ Compute the bit-vectors S
15. for i ← m − 2 downto 0 do
16.      S[i] ← B[i] | S[i + 1]
   Δ Set first bits for d ≥ d_0
17. for i ← d_0 to d_max do
18.      B[i] ← B[i] | 10^{k−1}
19.      S[i] ← S[i] | 10^{k−1}
20. return (B, S, d_0, d_max)

PBNDM(P, m, T, n)
1. (c, k) ← SELECTCHAR(P, m)
2. (B, S, d_0, d_max) ← PREPROCESS(P, m, c, k)
3. j ← 0
4. while j ≤ n − m do
5.       prfx ← 0
6.       D ← 1^k
7.       i ← m − 1
8.       l ← MAX(i − d_max, 0)
9.       (w, i) ← READ(P, i, l, c)
10.      if w > d_max then
11.            j ← j + m − d_0
12.            continue
13.      D ← D & S[w]
14.      if D & 1^{k−1} do
15.            prfx ← w + 1
16.            if prfx = m then
17.                  if P = T[j..j + m − 1] then
18.                        output j
19.                  prfx ← m − 1
20.      D ← D ≪ 1
21.      s ← w + 1
22.      while i ≥ 0 and D ≠ 0^k do
23.            i ← i − 1
24.            l ← MAX(i − d_max, 0)
25.            (w, i) ← READ(P, i, l, c)
26.            D ← D & B[w]
27.            if D & 1^{k−1} do
28.                  prfx ← d_0 + s
29.                  if prfx = m then
30.                        if P = T[j..j + m − 1] then
31.                              output j
32.                        prfx ← m − 1
33.            s ← s + w + 1
34.            D ← D ≪ 1
35.      j ← j + m − prfx
```

Fig. 2. The pseudocode of the PBNDM algorithm and its auxiliary procedures.

The pseudo-code of the preprocessing phase of the algorithm is shown in Figure 2. It makes use of an auxiliary procedure Read which performs a scan of the string S starting at position $j = i$ and proceeds from right to left until a given position $l \leq j$ is reached or an occurrence of the pivot character c is found. It returns the couple of integers $(i - j, j)$.

The implicit encoding of P_c is computed gradually, during the initialization of table B through procedure READ, which computes the next element of the implicit encoding of P_c. Table S is then computed from table B in a single-pass for loop. The time complexity of the preprocessing phase is $\mathcal{O}(m)$. Since $d_i \leq m$, then the space overhead to store B and S is $\mathcal{O}(m)$ too. Apart from the tables encoding the transitions of the suffix automaton, B and S, the preprocessing phase returns two additional integers, d_0 and $d_{max} = \max\{d_i \ : \ 0 \leq i \leq k\}$, which are used during the searching phase.

4.3 The Searching Phase

The searching phase of the PBNDM algorithm acts using a filtering method. Specifically, it first searches for all the occurrences of the pruned pattern P_c

in the text. When an occurrence of P_c is found, starting at position j of the text, the algorithm naively checks for the whole occurrence of the pattern, i.e. it checks if $P = T[j..j+m-1]$.

As in the original BNDM algorithm, a window W of length m is shifted over the text, starting from the left end of the text and sliding from left to right. At each iteration of the algorithm a position of the window W is attempted performing a scanning of its characters proceeding from right to left and performing the transitions over the automaton accordingly.

During the backward scanning, $\hat{W}_c = \langle w_0, w_1, ..., w_l \rangle$ is computed on the fly by procedure READ, and the automaton configurations s_i, represented as a bit-vector D of $\rho(c)+1$ bits, are updated accordingly.

If $w_l > d_{max}$, then the prefix of P_c of size d_0 has been recognized, so the window is shifted without performing any transition. Otherwise, the first transition is performed by setting $D \leftarrow D \,\&\, S[w_l]$, so that all states s_i such that $d_i \geq w_l$ are kept active. Transition on any subsequent w_i, for $0 < i \leq l$, is implemented as $D \leftarrow D \,\&\, B[w_i]$. Moreover, by the definition of B and S, s_0 is kept active during the i-th transition if $w_i \geq d_0$. When, after performing the i-th transition, state s_0 is active, then a prefix of P_c of size $d_0 + \sum_{j=i+1}^{k} w_j$ has been recognized.

Apart from the case where P_c is recognized, each attempt ends when either D becomes zero or it is established that $w_i > d_{max}$ while proceeding in the backward scan.

As the original BNDM algorithm, the PBNDM algorithm has a $\mathcal{O}(nm)$ worst case time complexity and a $\mathcal{O}(\sigma + m)$ space complexity.

5 Experimental Results

In this section, we compare our new algorithm against other suffix automaton based solutions, focusing on those which make use of bit-parallelism for solving the problem with long strings. In particular we included the following algorithms:

- BNDM: the Backward-Nondeterministic-DAWG-Matching algorithm [10];
- SBNDM: the Simplified BNDM algorithm [11];
- LBNDM: the Long BNDM algorithm [11];
- BSX: the BNDM algorithm [10] with Extended Shift [4];
- FBNDM: the Factorized variant [2] of the BNDM algorithm [10];
- PBNDM: our PBNDM algorithm, presented in Section 4;

All algorithms have been implemented in the C programming language and have been tested using the SMART tool [7]. Experiments have been executed locally on a computer running Linux Ubuntu 20.04.1 with an Intel Core i5 3.40 GHz processor and 8GB RAM. Our tests have been run on a genome sequence, a protein sequence, and an English text, each of size 5MB. Such sequences are provided by the Smart research tool and are available online for download (additional details on the sequences can be found in Faro et al. [7]). In our implementations the value of the word size[1] has been fixed to $w = 32$ and patterns

[1] The value of the word size has been chosen in order to better emphasize scaling problems of the several bit-parallel algorithms.

Average shifts on a genome sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	29	29	29	29	29	29	29	29	29	29	29
SBNDM	30	30	30	30	30	30	30	30	30	30	30
LBNDM	61	112	106	27	32	64	128	**256**	**512**	**1024**	**2048**
BXS	59	117	**219**	1	1	1	1	1	1	1	1
FBNDM	**62**	66	67	67	66	67	67	67	67	67	67
PBNDM	60	**123**	142	**139**	**137**	**130**	**130**	125	125	122	122
Gain	-3%	+5%	-35%	+107%	+107%	+94%	+1%	-51%	-75%	-88%	-94%

Average shifts on a protein sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	31	31	31	31	31	31	31	31	31	31	31
SBNDM	31	31	31	31	31	31	31	31	31	31	31
LBNDM	**63**	**126**	248	470	713	353	206	286	526	1027	2048
BXS	62	125	**252**	**502**	**984**	1710	1635	1	1	1	1
FBNDM	**63**	**126**	146	143	141	145	144	143	144	145	144
PBNDM	56	118	244	492	979	**1928**	**3022**	**3015**	**2942**	**2910**	**2871**
Gain	-11%	-6%	-3%	-2%	-1%	+13%	+85%	+954%	+459%	+183%	+41%

Average shifts on a natural language sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	30	30	30	30	30	30	30	30	30	30	30
SBNDM	31	31	31	31	31	31	31	31	31	31	31
LBNDM	**63**	126	247	471	824	1035	797	702	910	1467	2609
BXS	62	125	**252**	**505**	**1010**	**2008**	3910	7076	10410	11015	8533
FBNDM	**63**	**127**	156	157	156	156	156	155	156	156	157
PBNDM	58	122	245	493	982	1970	**3940**	**7784**	**15438**	**30706**	**60803**
Gain	-7%	-3%	-2%	-2%	-3%	-2%	+1%	+10%	+48%	+178%	+613%

Table 1. Average shifts achieved by bit-parallel algorithms on a genome sequence (on top), a protein sequence (in the middle) and natural language text (on bottom).

of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 6 \leq i \leq 16\}$. For each length, the mean over the running times of 500 runs (expressed in Gigabytes per second) and the average shift advancements has been computed .

Algorithms have been compared in terms of running times and average shift advancements. Table 1 and Table 2 report the average shift achieved by each algorithm during the searching phase and the the search speed (expressed in Gigabytes per second) observed in our experimental evaluations, respectively. In both tables best results have been boldfaced to ease their localization. Table 1 also reports the gain (expressed as a percentage) with respect to the best result obtained by the previous algorithms. If the shift advancement is lower, the gain is expressed with a negative value.

Regarding the average shift advancement our experimental results clearly show that PBNDM achieves the best performance in the most of practical cases and always proposes the largest advancements in the case of long patterns. This suggests that our automaton encoding scales better as the size of the pattern increases and is thus more suitable to handle long strings.

The only exception is the case of very small alphabets where PBNDM is second to LBNDM which is the clear winner in terms of shift advancements. In this case PBNDM proposes shorter shifts than 10 times those proposed by LBNDM. However, it is important to note that LBNDM search simultaneously m/w substrings of the pattern, each of length w (see Section 3), and that the

Experimental results on a genome sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	1.78	1.79	1.78	1.81	1.78	1.80	1.76	1.76	1.76	1.74	1.73
SBNDM	1.86	1.82	1.88	1.88	1.84	1.86	1.86	1.85	1.84	1.84	1.82
LBNDM	1.74	1.91	0.90	0.20	0.20	0.22	0.23	0.24	0.25	0.25	0.25
BXS	1.44	1.67	1.31	-	-	-	-	-	-	-	-
FBNDM	**2.21**	**2.29**	**2.28**	**2.26**	**2.29**	**2.27**	**2.28**	**2.25**	**2.26**	**2.22**	**2.21**
PBNDM	1.26	1.83	1.92	1.90	1.89	1.86	1.88	1.83	1.78	1.82	1.82

Experimental results on a protein sequence

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	2.24	2.23	2.21	2.23	2.17	2.21	2.19	2.19	2.21	2.21	2.16
SBNDM	2.33	2.31	2.36	2.31	2.30	2.34	2.33	2.29	2.34	2.30	2.26
LBNDM	2.34	2.56	**2.70**	**2.81**	2.63	1.32	0.60	0.47	0.45	0.46	0.47
BXS	2.18	2.36	2.61	2.52	2.49	2.16	-	-	-	-	-
FBNDM	**2.48**	**2.71**	2.67	2.71	**2.68**	2.68	2.67	2.67	2.68	2.65	2.38
PBNDM	1.27	1.69	2.18	2.47	2.56	**2.70**	**2.68**	**2.68**	**2.69**	**2.66**	**2.63**

Experimental results on a natural language text

m	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
BNDM	1.87	1.88	1.89	1.89	1.86	1.88	1.86	1.90	1.87	1.88	1.82
SBNDM	1.94	1.95	1.93	1.95	1.92	1.94	1.95	1.92	1.91	1.94	1.90
LBNDM	2.15	2.44	**2.65**	**2.81**	**2.77**	2.52	1.71	1.05	0.77	0.65	0.58
BXS	1.91	2.25	2.54	2.54	2.74	**2.84**	2.70	2.50	0.95	0.29	-
FBNDM	**2.28**	**2.60**	2.57	2.56	2.60	2.61	2.58	2.60	2.53	2.57	2.54
PBNDM	1.07	1.59	2.14	2.38	2.45	2.76	**2.84**	**3.64**	**4.93**	**6.18**	**7.40**

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

overall progress due to the shift is always counterbalanced by the number of checks that must be performed, which corresponds to the number of superimposed substrings. This results in poor performance in the case of very long patterns, especially when the size of the alphabet is small. In other words, the advantage obtained in the advancements is not exploited in the general performance of the algorithm. Less performances are also shown by the BNDM and SBNDM algorithms, due to the fact that the length of the pattern is always limited to the size of the word w.

Also the BXS algorithm uses an encoding based on a superimposed pattern and this causes it to have a degenerative behavior in the case of very long patterns, failing to scale well with respect to the length of the pattern. Specifically it does not work well when the superimposed pattern is not sensitive enough, i.e. too many different characters are accepted at the same position. This happens when the alphabet is too small or the pattern is too long.

We can observe, indeed, that although the size of the pattern increases the length of the superimposed pattern remains fixed at w. As a result the BXS algorithm, although particularly fast for patterns of moderate length (up to 128 or 256), is particularly slow for very long patterns, going beyond the execution time limit set for our tests, especially in the case of small alphabets. Its behavior, on the other hand, improves considerably as the size of the alphabet increases, becoming the best alternative to PBNDM for natural language texts.

The FBNDM algorithm is in general the one that performs best among the previous solutions, managing to scale the length of the shift quite well as the length of the pattern increases. Any factor of the pattern with no repeated

characters (see Section 3) is always limited by the size of the alphabet and, as the pattern length increases, this imposes a relatively limited number of factors which can be represented by a single computer word. As a consequence this limits the shift advancement to the overall length of the factors of the pattern taken into account. But despite this limitation FBNDM is in most cases the best alternative to PBNDM, especially in the case of small alphabets, and this translates into the best running times in most of the cases where PBNDM is not the best alternative.

On the whole we notice how the loss relative to the advancement of the shift never goes beyond 11% (with the exception of LBNDM as discussed above) while the gain obtained in increasing the shift, in the case of longer patterns, is impressive and reaches up to 954%. This aspect is clearly reflected in the execution times in which PBNDM is always among the first two best alternatives, becoming the fastest among all the algorithms for very long patterns. Although the presets remain moderate for small alphabets, the speed up achieved as the alphabet size increases is such that PBNDM is up to 3 times faster than the best alternative for natural language texts.

6 Conclusions

In this paper, we introduced a new algorithm, called PBNDM, based on a novel encoding of the suffix automaton of a string, suitable for patterns exceeding the word size w. Our algorithm is based on a pruned version of the pattern whose automaton can be encoded in an implicit form using few bits. From our experimental results our solution turns out to be competitive when compared for searching long strings against existing bit-parallel algorithms.

We observe that, although in this paper we focused on the application of the new encoding in the case of the exact pattern matching problem, it turns out to be flexible enough to be applied in all those solutions that make use of such data structure, even in the case of non-standard and approximate pattern matching.

In our future works we intend to tune the algorithm in order to make it competitive with the most efficient algorithms in practical cases, also in the case of small alphabets, a condition in which our algorithm still suffers due to the reduced length of the shifts. This includes the use of fast loops for further improving running times, and the use of condensed alphabets for representing longer and longer patterns inside a single word.

References

1. Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992. URL: `https://doi.org/10.1145/135239.135243`, `doi:10.1145/135239.135243`.
2. Domenico Cantone, Simone Faro, and Emanuele Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In Amihood Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching,*

21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings, volume 6129 of *Lecture Notes in Computer Science*, pages 288–298. Springer, 2010. URL: https://doi.org/10.1007/978-3-642-13509-5_26, doi:10.1007/978-3-642-13509-5_26.

3. Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994. URL: http://www-igm.univ-mlv.fr/%7Emac/REC/B1.html.
4. Branislav Durian, Hannu Peltola, Leena Salmela, and Jorma Tarhio. Bit-parallel search algorithms for long patterns. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 2010. URL: https://doi.org/10.1007/978-3-642-13193-6_12, doi:10.1007/978-3-642-13193-6_12.
5. Simone Faro and Thierry Lecroq. A fast suffix automata based algorithm for exact online string matching. In Nelma Moreira and Rogério Reis, editors, *Implementation and Application of Automata - 17th International Conference, CIAA 2012, Porto, Portugal, July 17-20, 2012. Proceedings*, volume 7381 of *Lecture Notes in Computer Science*, pages 149–158. Springer, 2012. URL: https://doi.org/10.1007/978-3-642-31606-7_13, doi:10.1007/978-3-642-31606-7_13.
6. Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13:1–13:42, 2013. URL: https://doi.org/10.1145/2431211.2431212, doi:10.1145/2431211.2431212.
7. Simone Faro, Thierry Lecroq, Stefano Borzi, Simone Di Mauro, and Alessandro Maggio. The string matching algorithms research tool. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2016*, pages 99–111. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016. URL: http://www.stringology.org/event/2016/p09.html.
8. M. J. Fischer and M. S. Paterson. String matching and other products. In R. Karp, editor, *Complexity of Computation (SYAM-AMS Proceedings 7)*, volume 7, pages 113,125, USA, 1974. Massachusetts Institute of Technology.
9. Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. URL: https://doi.org/10.1137/0206024, doi:10.1137/0206024.
10. Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In Martin Farach-Colton, editor, *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings*, volume 1448 of *Lecture Notes in Computer Science*, pages 14–33. Springer, 1998. URL: https://doi.org/10.1007/BFb0030778, doi:10.1007/BFb0030778.
11. Hannu Peltola and Jorma Tarhio. Alternative algorithms for bit-parallel string matching. In Mario A. Nascimento, Edleno Silva de Moura, and Arlindo L. Oliveira, editors, *String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003, Manaus, Brazil, October 8-10, 2003, Proceedings*, volume 2857 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2003. URL: https://doi.org/10.1007/978-3-540-39984-1_7, doi:10.1007/978-3-540-39984-1_7.
12. Ron Y. Pinter. Efficient string matching with don't-care patterns. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, pages 11–29, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.