A Compact Representation of Nondeterministic (Suffix) Automata for the Bit-Parallel Approach

Domenico Cantone, Simone Faro, and Emanuele Giaquinta

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone | faro | giaquinta}@dmi.unict.it

Abstract. We present a novel technique, suitable for bit-parallelism, for representing both the nondeterministic automaton and the nondeterministic suffix automaton of a given string in a more compact way. Our approach is based on a particular factorization of strings which on the average allows to pack in a machine word of w bits automata state configurations for strings of length greater than w. We adapted the Shift-And and BNDM algorithms using our encoding and compared them with the original algorithms. Experimental results show that the new variants are generally faster for long patterns.

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. The Knuth-Morris-Pratt (KMP) algorithm was the first linear-time solution (cf. [5]), whereas the Boyer-Moore (BM) algorithm provided the first sublinear solution on average [3]. Subsequently, the BDM algorithm reached the $\mathcal{O}(n \log_\sigma(m)/m)$ lower bound time complexity on the average (cf. [4]). 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 a deterministic suffix automaton for the language of the suffixes of P.

The bit-parallelism technique, introduced in [2], has been used to simulate efficiently the nondeterministic version of the KMP automaton. The resulting algorithm, named Shift-Or, runs in $\mathcal{O}(n\lceil m/w \rceil)$, where w is the number of bits in a computer word. Later, a variant of the Shift-Or algorithm, called Shift-And, and a very fast BDM-like algorithm (BNDM), based on the bit-parallel simulation of the nondeterministic suffix automaton, were presented in [6].

Bit-parallelism encoding requires one bit per pattern symbol, for a total of $\lceil m/w \rceil$ 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 $\lceil m/w \rceil$ grows. Though there are a few techniques to maintain good performance in the case of long patterns, such limitation is intrinsic.

In this paper we present an alternative technique, still suitable for bit-parallelism, to encode both the nondeterministic automaton and the nondeterministic suffix automaton of a given string in a more compact way. Our encoding is based on factorizations of strings in which no character occurs more

than once in any factor. This is the key towards separating the nondeterministic part from the determinstic one of the corresponding automata. It turns out that the nondeterministic part 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, bit-parallel algorithms based on such approach tend to be faster in the case of sufficiently long patterns. We will illustrate this point by comparing experimentally different implementations of the Shift-And and the BNDM algorithms.

2 Basic notions and definitions

Given a finite alphabet Σ, we denote by Σ^m, with $m \geq 0$, the collection 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 $0 \leqslant i \leqslant j < m$. Also, we put $\mathit{first}(P) = P[0]$ and $\mathit{last}(P) = P[|P| - 1]$. 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 $P.P'$, or more simply PP', for the concatenation of P and P', and P^r for the reverse of the string P, i.e. $P^r = P[m-1]P[m-2]\ldots P[0]$.

Given a string $P \in \Sigma^m$, we indicate with $\mathcal{A}(P) = (Q, \Sigma, \delta, q_0, F)$ the nondeterministic automaton for the language $\Sigma^* P$ of all words in Σ^* ending with an occurrence of P, where:

- $Q = \{q_0, q_1, \ldots, q_m\}$ (q_0 is the initial state)
- the transition function $\delta : Q \times \Sigma \longrightarrow \mathscr{P}(Q)$ is defined by:

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

- $F = \{q_m\}$ (F is the set of final states).

Likewise, for a string $P \in \Sigma^m$, we denote by $\mathcal{S}(P) = (Q, \Sigma, \delta, I, F)$ the nondeterministic suffix automaton with ε-transitions for the language $\mathit{Suff}(P)$ of the suffixes of P, where:

- $Q = \{I, q_0, q_1, \ldots, q_m\}$ (I is the initial state)
- the transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \mathscr{P}(Q)$ is defined by:

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

– $F = \{q_m\}$ (F is the set of final states).

The valid configurations $\delta^*(q_0, S)$ reachable by the automata $\mathcal{A}(P)$ on input $S \in \Sigma^*$ are defined recursively as follows:

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

Much the same definition of reachable configurations holds for the automata $\mathcal{S}(P)$, but in this case one has to use $\delta(I, \varepsilon) = \{q_0, q_1, \ldots, q_m\}$ for the base case.

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 The bit-parallelism technique

Bit-parallelism is a technique introduced by Baeza-Yates and Gonnet in [2] that takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to w, where w is the number of bits in the computer word. Bit-parallelism is particularly suitable for the efficient simulation of nondeterministic (suffix) automata; the first algorithms based on it are the well-known Shift-And [2] and BNDM [6]. The Shift-And algorithm simulates the nondeterministic automaton (NFA, for short) that recognizes the language $\Sigma^* P$, for a given string P of length m. Its 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 $\delta(q_i, c) = q_{i+1}$ or equivalently iff $P[i] = c$, for $c \in \Sigma$, $0 \leqslant i < m$. Automaton configurations $\delta^*(q_0, S)$ on input $S \in \Sigma^*$ are then encoded as a bit-vector D of m bits (the initial state does not need to be represented, as it is always active), where the i-th bit of D is set iff state q_{i+1} is active, i.e. $q_{i+1} \in \delta^*(q_0, S)$, for $i = 0, \ldots, m-1$. For a configuration D of the NFA, a transition on character c can then be implemented by the bitwise operations

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

The bitwise or with 1 (represented as $0^{m-1}1$) is performed to take into account the self-loop labeled with all the characters in Σ on the initial state. When a search starts, the initial configuration D is initialized to 0^m. Then, while the text is read from left to right, the automaton configuration is updated for each text character, as described before.

The nondeterministic suffix automaton for a given string P is an NFA with ε-transitions that recognizes the language $\mathit{Suff}(P)$. The BNDM algorithm simulates the suffix automaton for P^r with the bit-parallelism technique, using an encoding similar to the one described before for the Shift-And algorithm. The i-th bit of D is set iff state q_{i+1} is active, for $i = 0, 1, \ldots, m-1$, and D is initialized

to 1^m, since after the ε-closure of the initial state I all states q_i represented in D are active. The first transition on character c is implemented as $D \leftarrow (D \,\&\, B[c])$, while any subsequent transition on character c can be implemented as

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

The BNDM algorithm works by shifting a window of length m over the text. Specifically, for each window alignment, it searches the pattern by scanning the current window backwards and updating the automaton configuration accordingly. Each time a suffix of 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 w, the configuration bit-vector and all auxiliary bit-vectors need to be splitted over $\lceil m/w \rceil$ multiple words. For this reason the performance of the Shift-And and BNDM algorithms, and of bit-parallel algorithms more in general, degrades considerably as $\lceil m/w \rceil$ 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 w, which could be much smaller than m.

In the next section we illustrate an alternative encoding for automata configurations, which in general requires less than one bit per pattern character and still is suitable for bit-parallelism.

4 Tighter packing for bit-parallelism

We present a new encoding of the configurations of the nondeterministic (suffix) automaton for a given pattern P of length m, which on the average requires less than m bits and is still suitable to be used within the bit-parallel framework. The effect is that bit-parallel string matching algorithms based on such encoding scale much better as m grows, at the price of a larger space complexity. We will illustrate such point experimentally with the Shift-And and the BNDM algorithms, but our proposed encoding can also be applied to other variants of the BNDM algorithm as well.

Our encoding will have the form (D, a), where D is a k-bit vector, with $k \leqslant m$ (on the average k is much smaller than m), and a is an alphabet symbol (the last text character read) which will be used as a parameter in the bit-parallel simulation with the vector D.

The encoding (D, a) is obtained by suitably factorizing the simple bit-vector encoding for NFA configurations presented in the previous section. More specifically, it is based on the following pattern factorization.

Definition (1-factorization)**.** *Let $P \in \Sigma^m$. A 1-factorization of size k of P is a sequence $\langle u_1, u_2, \ldots, u_k \rangle$ of nonempty substrings of P such that:*

(a) $P = u_1 u_2 \ldots u_k$ *;*
(b) each factor u_j contains at most one *occurrence for any of the characters in the alphabet Σ, for $j = 1, \ldots, k$.*

A 1-factorization of P is minimal *if such is its size.*

Remark. It can easily be checked that a 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of P is minimal if *first*(u_{i+1}) occurs in u_i, for $i = 1, \ldots, k-1$.

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

A 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of a given pattern $P \in \Sigma^*$ induces naturally a partition $\{Q_1, \ldots, Q_k\}$ of the set $Q \setminus \{q_0\}$ of nonstarting states of the canonical automaton $\mathcal{A}(P) = (Q, \Sigma, \delta, q_0, F)$ for the language $\Sigma^* P$, where

$$Q_i =_{\text{Def}} \left\{ q_{\sum_{j=1}^{i-1} |u_j|+1}, \ldots, q_{\sum_{j=1}^{i} |u_j|} \right\}, \text{ for } i = 1, \ldots, k\,.$$

Notice that the labels of the arrows entering the states

$$q_{\sum_{j=1}^{i-1} |u_j|+1}, \ldots, q_{\sum_{j=1}^{i} |u_j|}\,,$$

in that order, form exactly the factor u_i, for $i = 1, \ldots, k$. Hence, if for any alphabet symbol a we denote by $Q_{i,a}$ the collection of states in Q_i with an incoming arrow labeled a, it follows that $|Q_{i,a}| \leqslant 1$, since by condition (b) of the above definition of 1-factorization no two states in Q_i can have an incoming transition labeled by a same character. When $Q_{i,a}$ is nonempty, we write $q_{i,a}$ to indicate the unique state q of $\mathcal{A}(P)$ for which $q \in Q_{i,a}$, otherwise $q_{i,a}$ is undefined. On using $q_{i,a}$ in any expression, we will also implicitly assert that $q_{i,a}$ is defined.

For any valid configuration $\delta^*(q_0, Sa)$ of the automaton $\mathcal{A}(P)$ on some input of the form $Sa \in \Sigma^*$, we have that $q \in \delta^*(q_0, Sa)$ only if the state q has an incoming transition labeled a. Therefore, $Q_i \cap \delta^*(q_0, Sa) \subseteq Q_{i,a}$ and, consequently, $|Q_i \cap \delta^*(q_0, Sa)| \leqslant 1$, for each $i = 1, \ldots, k$. The configuration $\delta^*(q_0, Sa)$ can then be encoded by the pair (D, a), where D is the bit-vector of size k such that $D[i]$ is set iff Q_i contains an active state, i.e., $Q_i \cap \delta^*(q_0, Sa) \neq \emptyset$, iff $q_{i,a} \in \delta^*(q_0, Sa)$. Indeed, if $i_1, i_2, \ldots, i_l$ are all the indices i for which $D[i]$ is set, we have that $\delta^*(q_0, Sa) = \{q_{i_1,a}, q_{i_2,a}, \ldots, q_{i_l,a}\}$ holds, showing that the above encoding (D, a) can be inverted.

```
F-PREPROCESS (P, m)

for c ∈ Σ do S[c] ← L[c] ← 0
for c, c' ∈ Σ do B[c][c'] ← 0
b ← 0, e ← 0, k ← 0
while e < m do
    while e < m and S[P[e]] = 0 do
        S[P[e]] ← 1, e ← e + 1
    for i ← b to e − 1 do S[P[i]] ← 0
    for i ← b + 1 to e − 1 do
        B[P[i − 1]][P[i]] ← B[P[i − 1]][P[i]] | (1 ≪ k)
    L[P[e − 1]] ← L[P[e − 1]] | (1 ≪ k)
    if e < m then
        B[P[e − 1]][P[e]] ← B[P[e − 1]][P[e]] | (1 ≪ k)
    b ← e
    k ← k + 1
M ← (1 ≪ (k − 1))
return (B, L, M, k)
```

Fig. 1. Preprocessing procedure for the construction of the arrays B and L relative to a minimal 1-factorization of the pattern.

To show how to compute D' in a transition $(D, a) \xrightarrow{\mathcal{A}} (D', c)$ on character c using bit-parallelism, it is convenient to give some further definitions.

For $i = 1, \ldots, k-1$, we put $\overline{u}_i = u_i.first(u_{i+1})$. We also put $\overline{u}_k = u_k$ and call each set $\overline{u}_i$ the *closure* of u_i.

Plainly, any 2-gram can occur at most once in the closure $\overline{u}_i$ of any factor of our 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of P. We can therefore encode the 2-grams present in the closure of the factors u_i by a $|\Sigma| \times |\Sigma|$ matrix B of k-bit vectors, where the i-th bit of $B[c_1][c_2]$ is set iff the 2-gram c_1c_2 is present in $\overline{u}_i$ or, equivalently, iff

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

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

To properly take care of transitions from the last state in Q_i to the first state in Q_{i+1}, it is also useful to have an array L, of size $|\Sigma|$, of k-bit vectors encoding for each character $c \in \Sigma$ the collection of factors ending with c. More precisely, the i-th bit of $L[c]$ is set iff $last(u_i) = c$, for $i = 1, \ldots, k$.

We show next that the matrix B and the array L, which in total require $(|\Sigma|^2 + |\Sigma|)k$ bits, are all is needed to compute the transition $(D, a) \xrightarrow{\mathcal{A}} (D', c)$ on character c. To this purpose, we first state the following basic property, which can easily be proved by induction.

Transition Lemma. *Let $(D, a) \xrightarrow{\mathcal{A}} (D', c)$, where (D, a) is the encoding of the configuration $\delta^*(q_0, Sa)$ for some string $S \in \Sigma^*$, so that (D', c) is the encoding of the configuration $\delta^*(q_0, Sac)$.*

Then, for each $i = 1, \ldots, k$, $q_{i,c} \in \delta^*(q_0, Sac)$ if and only if either

(i) $last(u_i) \neq a$, $q_{i,a} \in \delta^*(q_0, Sa)$, *and* $q_{i,c} \in \delta(q_{i,a}, c)$, *or*
(ii) $i \geqslant 1$, $last(u_{i-1}) = a$, $q_{i-1,a} \in \delta^*(q_0, Sa)$, *and* $q_{i,c} \in \delta(q_{i-1,a}, c)$. □

Now observe that, by definition, the i-th bit of D' is set iff $q_{i,c} \in \delta^*(q_0, Sac)$ or, equivalently by the Transition Lemma and (1), iff (for $i = 1, \ldots, k$)

$$
\begin{aligned}
&(D[i] = 1 \wedge B[a][c][i] = 1 \wedge {\sim} L[a][i] = 1) \vee \\
&\qquad (i \geqslant 1 \wedge D[i-1] = 1 \wedge B[a][c][i-1] = 1 \wedge L[a][i-1] = 1) && \text{iff} \\
&((D \mathbin{\&} B[a][c] \mathbin{\&} {\sim} L[a])[i] = 1 \vee (i \geqslant 1 \wedge (D \mathbin{\&} B[a][c] \mathbin{\&} L[a])[i-1] = 1)) && \text{iff} \\
&((D \mathbin{\&} B[a][c] \mathbin{\&} {\sim} L[a])[i] = 1 \vee ((D \mathbin{\&} B[a][c] \mathbin{\&} L[a]) \ll 1)[i] = 1) && \text{iff} \\
&((D \mathbin{\&} B[a][c] \mathbin{\&} {\sim} L[a]) \mid ((D \mathbin{\&} B[a][c] \mathbin{\&} L[a]) \ll 1))[i] = 1\,.
\end{aligned}
$$

Hence $D' = (D \mathbin{\&} B[a][c] \mathbin{\&} {\sim} L[a]) \mid ((D \mathbin{\&} B[a][c] \mathbin{\&} L[a]) \ll 1)$, so that D' can be computed by the following bitwise operations:

$$
\begin{aligned}
&D \leftarrow D \mathbin{\&} B[a][c] \\
&H \leftarrow D \mathbin{\&} L[a] \\
&D \leftarrow (D \mathbin{\&} {\sim} H) | (H \ll 1)\,.
\end{aligned}
$$

To check whether the final state q_m belongs to a configuration encoded as (D, a), we have only to verify that $q_{k,a} = q_m$. This test can be broken into two steps: first, one checks if any of the states in Q_k is active, i.e. $D[k] = 1$; then, one verifies that the last character read is the last character of u_k, i.e. $L[a][k] = 1$. The whole test can then be implemented with the bitwise test $D \mathbin{\&} M \mathbin{\&} L[a] \neq 0^k$, where $M = (1 \ll (k-1))$.

The same considerations also hold for the suffix automaton $\mathcal{S}(P)$. The only difference is in the handling of the initial state. In the case of the automaton $\mathcal{A}(P)$, state q_0 is always active, so we have to activate state q_1 when the current text symbol is equal to $P[0]$. To do so it is enough to perform a bitwise or of D with $0^{k-1}1$ when $a = P[0]$, as $q_1 \in Q_1$. Instead, in the case of the suffix automaton $\mathcal{S}(P)$, as the initial state has an ε-transition to each state, all the bits in D must be set, as in the BNDM algorithm.

The preprocessing procedure which builds the arrays B and L described above and relative to a minimal 1-factorization of the given pattern $P \in \Sigma^m$ is reported in Figure 1. Its time complexity is $\mathcal{O}(|\Sigma|^2 + m)$. The variants of the Shift-And and BNDM algorithms based on our encoding of the configurations of the automata $\mathcal{A}(P)$ and $\mathcal{S}(P)$ are reported in Figure 2 (algorithms F-Shift-And and F-BNDM, respectively). Their worst-case time complexities are $\mathcal{O}(n\lceil k/w \rceil)$ and $\mathcal{O}(nm\lceil k/w \rceil)$, respectively, while their space complexity is $\mathcal{O}(|\Sigma|^2\lceil k/w \rceil)$, where k is the size of a minimal 1-factorization of the pattern.

5 Experimental results

In this section we present and comment the experimental results relative to an extensive comparison of the BNDM and the F-BNDM algorithms and the

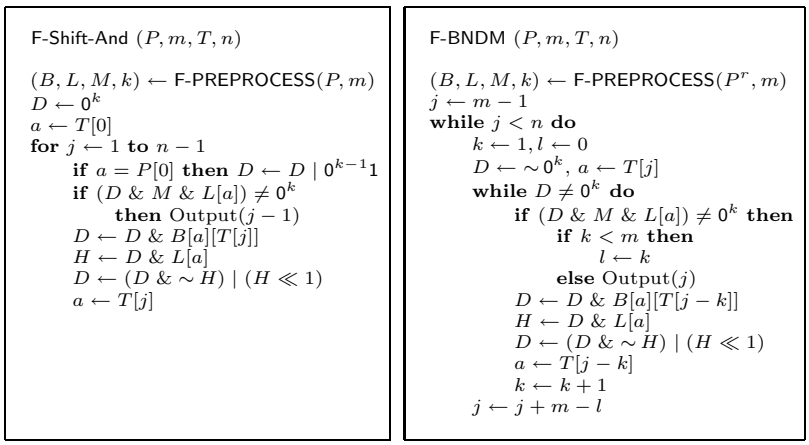

Fig. 2. Variants of Shift-And and BNDM based on the 1-factorization encoding.

Shift-And and F-Shift-And algorithms. In particular, in the BNDM case we have implemented two variants for each algorithm, named *single word* and *multiple words*, respectively. Single word variants are based on the automaton for a suitable substring of the pattern whose configurations can fit in a computer word; a naive check is then used to verify whether any occurrence of the subpattern can be extended to an occurrence of the complete pattern: specifically, in the case of the BNDM algorithm, the prefix pattern of length $\min(m, w)$ is chosen, while in the case of the F-BNDM algorithm the longest substring of the pattern which is a concatenation of at most w consecutive factors is selected. Multiple words variants are based on the automaton for the complete pattern whose configurations are splitted, if needed, over multiple machine words. The resulting implementations are referred to in the tables below as BNDM* and F-BNDM*.

We have also included in our tests the LBNDM algorithm [8]. When the alphabet is considerably large and the pattern length is at least two times the word size, the LBNDM algorithm achieves larger shift lengths. However, the time for its verification phase grows proportionally to m/w, so there is a treshold after which its performance degrades significantly.

For the Shift-And case, only test results relative to the multiple words variant have been included in the tables below, since the overhead due to a more complex bit-parallel simulation in the single word case is not paid off by the reduction of the number of calls to the verification phase.

The main two factors on which the efficiency of BNDM-like algorithms depends are the maximum shift length and the number of words needed for representing automata configurations. For the variants of the first case, the shift length can be at most the length of the longest substring of the pattern that fits in a computer word. This, for the BNDM algorithm, is plainly equal to $\min(w, m)$, so the word size is an upper bound for the shift length, whereas in the case of

the F-BNDM algorithm it is generally possible to achieve shifts of length larger than w, as our encoding allows to pack more state configurations per bit on the average as shown in a table below. In the multi-word variants, the shift lengths for both algorithms, denoted BNDM* and F-BNDM*, are always the same, as they use the very same automaton; however, the 1-factorization based encoding involves a smaller number of words on the average, especially for long patterns, thus providing a considerable speedup.

All algorithms have been implemented in the C programming language and have been compiled with the `GNU C Compiler`, using the optimization options `-O2 -fno-guess-branch-probability`. All tests have been performed on a 2 GHz Intel Core 2 Duo and running times have been measured with a hardware cycle counter, available on modern CPUs. We used the following input files: (i) the English King James version of the "Bible" (with $\sigma = 63$); (ii) a protein sequence from the *Saccharomyces cerevisiae* genome (with $\sigma = 20$); and (iii) a genome sequence of $4,638,690$ base pairs of *Escherichia coli* (with $\sigma = 4$).

Files (i) and (iii) are from the Canterbury Corpus [1], while file (ii) is from the Protein Corpus [7]. For each input file, we have generated sets of 200 patterns of fixed length m randomly extracted from the text, for m ranging over the values 32, 64, 128, 256, 512, 1024, 1536, 2048, 4096. For each set of patterns we reported the mean over the running times of the 200 runs.

m	Shift-And*	F-Shift-And*	LBNDM	BNDM	BNDM*	F-BNDM	F-BNDM*
32	8.85	22.20	2.95	**2.81**	2.92	2.92	2.99
64	51.45	22.20	**1.83**	2.82	3.31	2.00	1.97
128	98.42	22.21	**1.82**	2.83	3.58	2.35	2.23
256	142.27	92.58	**1.38**	2.82	2.79	1.91	2.14
512	264.21	147.79	**1.09**	2.84	2.47	1.81	1.75
1024	508.71	213.70	**1.04**	2.84	2.67	1.77	1.72
1536	753.02	283.57	**1.40**	2.84	2.95	1.77	1.73
2048	997.19	354.32	2.24	2.84	3.45	**1.75**	1.90
4096	1976.09	662.06	10.53	2.83	6.27	**1.72**	2.92

Experimental results on the King James version of the Bible ($\sigma = 63$)

m	Shift-And*	F-Shift-And*	LBNDM	BNDM	BNDM*	F-BNDM	F-BNDM*
32	6.33	15.72	1.50	1.58	1.64	**1.43**	1.56
64	38.41	15.70	0.99	1.57	1.70	**0.89**	0.96
128	70.59	40.75	0.70	1.57	1.42	**0.64**	1.01
256	104.42	73.59	**0.52**	1.57	1.39	0.59	1.01
512	189.16	108.33	**0.41**	1.57	1.29	0.56	0.88
1024	362.83	170.52	**0.54**	1.58	1.46	0.55	0.91
1536	540.25	227.98	2.09	1.57	1.73	**0.56**	1.04
2048	713.87	290.24	7.45	1.58	2.12	**0.56**	1.20
4096	1413.76	541.53	32.56	1.58	4.87	**0.59**	2.33

Experimental results on a protein sequence from the *Saccharomyces cerevisiae* genome ($\sigma = 20$)

m	Shift-And*	F-Shift-And*	LBNDM	BNDM	BNDM*	F-BNDM	F-BNDM*
32	10.19	25.04	4.60	**3.66**	3.82	5.18	4.88
64	59.00	42.93	3.42	3.64	5.39	**2.94**	3.69
128	93.97	114.22	3.43	3.65	5.79	**2.66**	5.37
256	162.79	167.11	11.68	3.64	4.79	**2.59**	4.11
512	301.55	281.37	82.94	3.66	4.16	**2.53**	3.54
1024	579.92	460.37	96.13	3.64	4.21	**2.50**	3.42
1536	860.84	649.88	91.45	3.64	4.54	**2.49**	3.66
2048	1131.50	839.32	89.45	3.64	4.98	**2.48**	3.98
4096	2256.37	1728.71	85.87	3.64	7.81	**2.48**	6.22

Experimental results on a genome sequence of *Escherichia coli* ($\sigma = 4$)

(A)	ecoli	protein	bible
32	32	32	32
64	63	64	64
128	72	122	128
256	74	148	163
512	77	160	169
1024	79	168	173
1536	80	173	176
2048	80	174	178
4096	82	179	182

(B)	ecoli	protein	bible
32	15	8	6
64	29	14	12
128	59	31	26
256	119	60	50
512	236	116	102
1024	472	236	204
1536	705	355	304
2048	944	473	407
4096	1882	951	813

(C)	ecoli	protein	bible
32	2.13	4.00	5.33
64	2.20	4.57	5.33
128	2.16	4.12	4.92
256	2.15	4.26	5.12
512	2.16	4.41	5.01
1024	2.16	4.33	5.01
1536	2.17	4.32	5.05
2048	2.16	4.32	5.03
4096	2.17	4.30	5.03

(A) The length of the pattern fitting in w bits.
(B) The size of the minimal 1-factorization of the pattern.
(C) The ratio between m and the size of the minimal 1-factorization of the pattern.

Concerning the BNDM-like algorithms, the experimental results show that in the case of long patterns both variants based on the 1-factorization encoding are considerably faster than their corresponding variants BNDM and BNDM*. In the first test suite, with $\sigma = 63$, the LBNDM algorithm turns out to be the fastest one, except for very long patterns, as the treshold on large alphabets is quite high. In the second test suite, with $\sigma = 20$, LBNDM is still competitive but, in the cases in which it beats the BNDM* algorithm, the difference is thin.

Likewise, the F-Shift-And variant is faster than the classical Shift-And algorithm in all cases, for $m \geqslant 64$.

6 Conclusions

We have presented an alternative technique, suitable for bit-parallelism, to represent the nondeterministic automaton and the nondeterministic suffix automaton of a given string. On the average, the new encoding allows to pack in a single machine word of w bits state configurations of (suffix) automata relative to strings of more than w characters long. When applied to the BNDM algorithm, and for long enough patterns, our encoding allows larger shifts in the case of the single word variant and a more compact encoding in the case of the multiple words variant, resulting in faster implementations.

Further compactness could be achieved with 2-factorizations (with the obvious meaning), or with hybrid forms of factorizations. Clearly, more involved factorizations will also result into more complex bit-parallel simulations and larger space complexity, thus requiring a careful tuning to identify the best degree of compactness for the application at hand.

References

1. R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *DCC '97: Proceedings of the Conference on Data Compression*, page 201, Washington, DC, USA, 1997. IEEE Computer Society. `http://corpus.canterbury.ac.nz/`.
2. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
4. M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.
5. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.
6. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *Proc. of the 9th Annual Symposium on Combinatorial Pattern Matching*, volume 1448 of *Lecture Notes in Computer Science*, pages 14–33. Springer-Verlag, Berlin, 1998.
7. C. G. Nevill-Manning and I. H. Witten. Protein is incompressible. In *DCC '99: Proceedings of the Conference on Data Compression*, page 257, Washington, DC, USA, 1999. IEEE Computer Society. `http://data-compression.info/Corpora/ProteinCorpus/`.
8. H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In M. A. Nascimento, E. Silva de Moura, and A. L. Oliveira, editors, *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-Verlag, Berlin, 2003.