Information and Computation 213 (2012) 3–12

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

A compact representation of nondeterministic (suffix) automata for the bit-parallel approach

Domenico Cantone, Simone Faro, Emanuele Giaquinta [*]

Università di Catania, Dipartimento di Matematica e Informatica, Viale Andrea Doria 6, I-95125 Catania, Italy

A R T I C L E I N F O

Article history:
Available online 2 February 2012

A B S T R A C T

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. [7]), whereas the Boyer–Moore (BM) algorithm provided the first sublinear solution on average [2]. 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 simulate, respectively, a deterministic automaton for the language Σ^*P and a deterministic suffix automaton for the language of the suffixes of P.

The bit-parallelism technique, introduced in [1], 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. A variant of the Shift-Or algorithm, called Shift-And, was described in [11]. Later, a very fast BDM-like algorithm (BNDM), based on the bit-parallel simulation of the nondeterministic suffix automaton, was presented in [8].

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 degrade 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 deterministic 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,

[*] Corresponding author.
E-mail addresses: cantone@dmi.unict.it (D. Cantone), faro@dmi.unict.it (S. Faro), giaquinta@dmi.unict.it (E. Giaquinta).

0890-5401/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2011.03.006

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 of size σ, we denote by Σ^m, with $m \geqslant 0$, 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 $P[0 \ldots m-1]$ 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 $P[i \ldots j]$ is the substring of P contained between its $(i+1)$-st and $(j+1)$-st characters, for $0 \leqslant i \leqslant j < m$. Also, we define $\mathit{first}(P) = P[0]$ and $\mathit{last}(P) = P[|P|-1]$. For any two strings P and P', we say that P' is a suffix of P if $P' = P[i \ldots m-1]$, 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 $P' = P[0 \ldots i]$, 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 $\delta : Q \times \Sigma \to \mathscr{P}(Q)$ is defined by

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

- $F = \{q_m\}$ (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 $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathscr{P}(Q)$ is defined by

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

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

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

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

Much the same definition of reachable configurations holds for the automaton $\mathcal{S}(P)$, 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` "≪" 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 "∼".

3. The bit-parallelism technique

Bit-parallelism is a technique introduced by Baeza-Yates and Gonnet in [1] 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 [11] (a variant of Shift-Or [1]) and BNDM [8]. 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 σ 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 $i = 0, \dots, m-1$. For a configuration D of the NFA, a transition on character c can then be implemented by the bitwise operations

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

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 $i = 0, 1, \dots, m-1$, 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

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

The BNDM algorithm works by shifting a window of length m over the text. Specifically, for each window alignment, it searches for 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 split 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. While this approach has no drawback when applied to Shift-And, in the case of the BNDM algorithm it limits the maximum possible shift length to 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 this encoding scale much better as m grows, at the price of a larger space complexity. We will illustrate such a 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.

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

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

For a given 1-factorization $\boldsymbol{u} = \langle u_1, u_2, \dots, u_k \rangle$ of P, we define

$$r_j^{\boldsymbol{u}} =_{\mathrm{Def}} |u_1 u_2 \dots u_{j-1}|, \tag{1}$$

for $j = 1, 2, \dots, k+1$ (so that $r_1^{\boldsymbol{u}} = 0$ and $r_{k+1}^{\boldsymbol{u}} = m$) and call the numbers $r_1^{\boldsymbol{u}}, r_2^{\boldsymbol{u}}, \dots, r_{k+1}^{\boldsymbol{u}}$ the *indices of* $\boldsymbol{u}$. Plainly, $r_j^{\boldsymbol{u}}$ is the index in P of the first character of the factor u_j, for $j = 1, 2, \dots, k$.

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

Observe that the size k of a 1-factorization $\boldsymbol{u}$ of a string $P \in \Sigma^m$ satisfies the condition

$$\left\lceil \frac{m}{\sigma} \right\rceil \leqslant k \leqslant m.$$

Indeed, as the length of any factor in $\boldsymbol{u}$ is limited by the size σ of the alphabet Σ, then $m \leqslant k\sigma$, which implies $\lceil \frac{m}{\sigma} \rceil \leqslant k$. The second inequality is immediate and occurs when P has the form a^m, for some $a \in \Sigma$, in which case P has only the 1-factorization of size m whose factors are all equal to the single character string a.

As we shall show below, the size of the bit-vector D in our encoding depends on the size of the 1-factorization used; as a result, a minimal 1-factorization yields the smallest vector. A greedy approach to construct a 1-factorization of smallest size for a string P consists in computing the longest prefix u_1 of P containing no repeated characters and then recursively 1-factorize the string P deprived of its prefix u_1, as in the procedure Greedy-1-Factorize shown below.

```
Greedy-1-Factorize($P$)

if $P$ is the empty string $\varepsilon$ then
  return the empty sequence $\langle\,\rangle$
else
  $u_1 \leftarrow$ longest prefix of $P$ containing no repeated character
  $P' \leftarrow$ the suffix of $P$ such that $P = u_1 P'$
  return the sequence obtained by prepending the factor $u_1$ to the
         sequence Greedy-1-Factorize($P'$)
endif
```

The correctness of the procedure Greedy-1-Factorize is shown in the following lemma:

Lemma 1. *The call* Greedy-1-Factorize(P), *for a string $P \in \Sigma^m$, computes a minimal* 1*-factorization of P.*

Proof. Let $\boldsymbol{u} = \langle u_1, u_2, \ldots, u_k \rangle$ be the 1-factorization computed by the call Greedy-1-Factorize(P) and let $\boldsymbol{v} = \langle v_1, v_2, \ldots, v_h \rangle$ be any 1-factorization of P. We just need to show that $k \leqslant h$.

By construction, the character *first*(u_{i+1}) occurs in u_i, for $i = 1, 2, \ldots, k-1$, otherwise the factor u_i could have been extended by at least one more character.

We say that the factor v_j of $\boldsymbol{v}$ *covers* the factor u_i of $\boldsymbol{u}$ if $r_j^{\boldsymbol{v}} \leqslant r_i^{\boldsymbol{u}} < r_{j+1}^{\boldsymbol{v}}$ (see (1)), i.e., if j is the largest index such that the string $v_1 v_2 \ldots v_{j-1}$ is a prefix of the string $u_1 u_2 \ldots u_{i-1}$.

Plainly, each factor of $\boldsymbol{u}$ is covered by exactly one factor of $\boldsymbol{v}$. Thus, for our purposes, it is enough to show that each factor of $\boldsymbol{v}$ can cover at most one factor of $\boldsymbol{u}$, so that the number of factors in $\boldsymbol{v}$ must be at least as large as the number of factors in $\boldsymbol{u}$. Indeed, if this were not the case then

$$r_j^{\boldsymbol{v}} \leqslant r_i^{\boldsymbol{u}} < r_{i+1}^{\boldsymbol{u}} < r_{j+1}^{\boldsymbol{v}},$$

for some $i \in \{1, 2, \ldots, h\}$ and $j \in \{1, 2, \ldots, k\}$. But then, the string $u_i.$*first*(u_{i+1}), which contains two occurrences of the character *first*(u_{i+1}), would be a factor of v_j, which is a contradiction. □

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 states of the automaton $\mathcal{A}(P) = (Q, \Sigma, \delta, q_0, F)$ for the language $\Sigma^* P$, where

$$Q_i =_{\mathrm{Def}} \{q_{r_i+1}, \ldots, q_{r_{i+1}}\}, \quad \text{for } i = 1, \ldots, k,$$

and $r_1, r_2, \ldots, r_{k+1}$ are the indices of $\langle u_1, u_2, \ldots, u_k \rangle$.

Notice that the labels of the arrows entering the states

$$q_{r_i+1}, \ldots, q_{r_{i+1}},$$

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. Upon 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, which shows that the above encoding (D, a) can be inverted.

To illustrate 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 $\bar{u}_i = u_i.\mathit{first}(u_{i+1})$. We also put $\bar{u}_k = u_k$ and call each set $\bar{u}_i$ the *closure* of u_i.

Plainly, any 2-gram can occur at most once in the closure $\bar{u}_i$ of any factor of our 1-factorization $\langle u_1, u_2, \ldots, u_k\rangle$ 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 $\bar{u}_i$ or, equivalently, iff

$$\bigl(\mathit{last}(u_i) \neq c_1 \wedge q_{i,c_2} \in \delta(q_{i,c_1}, c_2)\bigr) \vee \bigl(i < k \wedge \mathit{last}(u_i) = c_1 \wedge q_{i+1,c_2} \in \delta(q_{i,c_1}, c_2)\bigr), \tag{2}$$

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 σ, 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 $\mathit{last}(u_i) = c$, 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.

Lemma 2 *(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) *$\mathit{last}(u_i) \neq a$, $q_{i,a} \in \delta^*(q_0, Sa)$, and $q_{i,c} \in \delta(q_{i,a}, c)$, or*
(ii) *$i \geqslant 1$, $\mathit{last}(u_{i-1}) = a$, $q_{i-1,a} \in \delta^*(q_0, Sa)$, 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 (2), iff (for $i = 1, \ldots, k$)

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

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:

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

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

$$D \mathbin{\&} 10^{k-1} \mathbin{\&} L[a] \neq 0^k.$$

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.

A drawback of the new encoding is that the handling of the self-loop and of the acceptance condition is more complex with respect to the original encoding. However, it is possible to simplify them at the expense of an overhead of at most two bits in the representation, by forcing the first and last factor in a 1-factorization to have length 1. Note that the handling of the self-loop is relevant for the automaton $\mathcal{A}(P)$ only, while the acceptance condition concerns both the $\mathcal{A}(P)$ and $\mathcal{S}(P)$ automata. In particular, to simplify the handling of the self-loop, we compute a factorization where the length of the first factor is equal to 1. Let $\langle v_1, v_2, \ldots, v_l\rangle$ be a minimal 1-factorization of $P[1 \ldots m-1]$; we define the following 1-factorization $\langle u_1, u_2, \ldots, u_{l+1}\rangle$ of P, where

```
F-PREPROCESS($P, m$)

for $c \in \Sigma$ do $S[c] \leftarrow 0$
for $c \in \Sigma$ do $L[c] \leftarrow 0$
for $c, c' \in \Sigma$ do $B[c][c'] \leftarrow 0$
$b \leftarrow 0,\ e \leftarrow 0,\ k \leftarrow 0$
while $e < m$ do
    while $e < m$ and $S[P[e]] < k+1$ do
        $S[P[e]] \leftarrow k+1,\ e \leftarrow e+1$
    for $i \leftarrow b+1$ to $e-1$ do
        $B[P[i-1]][P[i]] \leftarrow B[P[i-1]][P[i]] \mid (1 \ll k)$
    $L[P[e-1]] \leftarrow L[P[e-1]] \mid (1 \ll k)$
    if $e < m$ then
        $B[P[e-1]][P[e]] \leftarrow B[P[e-1]][P[e]] \mid (1 \ll k)$
    $b \leftarrow e$
    $k \leftarrow k+1$
return $(B, L, k)$
```

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

```
F-Shift-And($P, m, T, n$)

$(B, L, k) \leftarrow$ F-PREPROCESS($P, m$)
$D \leftarrow 0^k$
$a \leftarrow T[0]$
for $j \leftarrow 1$ to $n-1$
    if $a = P[0]$ then $D \leftarrow D \mid 0^{k-1}1$
    $D \leftarrow D \,\&\, B[a][T[j]]$
    $H \leftarrow D \,\&\, L[a]$
    $D \leftarrow (D \,\&\, \sim H) \mid (H \ll 1)$
    $a \leftarrow T[j]$
    if $(D \,\&\, 10^{k-1} \,\&\, L[a]) \neq 0^k$
        then Output($j$)
```

(a)

```
F-BNDM($P, m, T, n$)

$(B, L, k) \leftarrow$ F-PREPROCESS($P^r, m$)
$j \leftarrow m-1$
while $j < n$ do
    $i \leftarrow 1, l \leftarrow 0$
    $D \leftarrow 1^k,\ a \leftarrow T[j]$
    while $D \neq 0^k$ do
        if $(D \,\&\, 10^{k-1} \,\&\, L[a]) \neq 0^k$ then
            if $i < m$ then
                $l \leftarrow i$
            else Output($j$)
        $D \leftarrow D \,\&\, B[a][T[j-i]]$
        $H \leftarrow D \,\&\, L[a]$
        $D \leftarrow (D \,\&\, \sim H) \mid (H \ll 1)$
        $a \leftarrow T[j-i]$
        $i \leftarrow i+1$
    $j \leftarrow j+m-l$
```

(b)

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

$$u_i = \begin{cases} P[0] & \text{if } i = 1, \\ v_{i-1} & \text{if } 2 \leqslant i \leqslant l+1. \end{cases}$$

Observe that the size of Q_1 in the corresponding partition is 1; it follows that to handle the self-loop one does not need to perform the check $a = P[0]$ but just perform a bitwise or with $0^{k-1}1$, as there is one state only in the first subset and thus $q_{1,a}$ is undefined for all $a \neq P[0]$.

Similarly, to simplify the acceptance condition, we compute a factorization where the length of the last factor is equal to 1; let $\langle v_1, v_2, \ldots, v_l \rangle$ be a minimal 1-factorization of $P[0 \ldots m-2]$; we define the following 1-factorization $\langle u'_1, u'_2, \ldots, u'_{l+1} \rangle$ of P where

$$u'_i = \begin{cases} v_i & \text{if } 1 \leqslant i \leqslant l, \\ P[m-1] & \text{if } i = l+1. \end{cases}$$

In this case, the bitwise and with $L[a]$ in the acceptance condition test is not needed anymore, as there is only one state in the last subset.

Let k be the size of a minimal 1-factorization of P. Plainly, in both cases, $k-1 \leqslant l \leqslant k$; in particular, if $l = k$, we have an overhead of 1 bit. If we combine the two techniques, the overhead in the representation is of at most two bits.

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 Fig. 1. Its time complexity is $\mathcal{O}(\sigma^2\lceil k/w \rceil + m)$. 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 Fig. 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. q-Grams based 1-factorization

It is possible to achieve higher compactness by transforming the pattern into a sequence of overlapping q-grams and computing the 1-factorization of the resulting string. This technique has been extensively used to boost the performance of several string matching algorithms [5,10]. More precisely, given a pattern P of length m defined over an alphabet Σ of size σ, the q-gram encoding of P is the string $P^{(q)}_0 P^{(q)}_1 \dots P^{(q)}_{m-q}$, defined over the alphabet Σ^q of the q-grams of Σ, where

$$P^{(q)}_i = P[i]P[i+1]\dots P[i+q-1],$$

i.e., the pattern is transformed into the sequence of its $m-q+1$ overlapping substrings of length q, where each substring $P^{(q)}_i$ is regarded as a symbol belonging to Σ^q. Clearly, the size of the 1-factorization of the resulting string is at least $\lceil \frac{m-q+1}{\sigma^q} \rceil$. Hence, the size of the 1-factorization can be significantly reduced by using a q-grams representation. The only drawback is that the space needed for the tables B and L can grow up to $(\sigma^{2q}+\sigma^q)k$, where k is the size of the 1-factorization, if one follows a naive approach. Observe that, for each pair $(P^{(q)}_i, P^{(q)}_{i+1})$, with $i = 0,\dots,m-q-1$, the corresponding transition must be encoded into the table B. However, as we are using overlapping q-grams, we have that $P^{(q)}_i[1\dots q-1] \sqsubset P^{(q)}_{i+1}$, i.e., the last $q-1$ symbols of $P^{(q)}_i$ are equal to the first $q-1$ symbols of $P^{(q)}_{i+1}$. Thus, there is no need to use the full q-gram $P^{(q)}_{i+1}$ as index in the table, rather we can use only its last symbol. More precisely, let $\langle u^q_1, u^q_2, \dots, u^q_k \rangle$ be a 1-factorization of the q-gram encoding of P; we encode the substrings of length $2q$ (or, equivalently, the 2-grams over Σ^q) present in the closure of the factors u^q_i by a $\Sigma^q \times \Sigma$ matrix B of k bit vectors, where the i-th bit of $B[C_1][c_2]$ is set iff the substring $C_1.C_1[1\dots q-1].c_2$ is present in $\bar{u}^q_i$, for every $C_1 \in \Sigma^q, c_2 \in \Sigma$. This method reduces the space complexity to $(\sigma^{q+1}+\sigma^q)k$. In general, however, it is not feasible to use a direct access table for the tables B and T with this encoding.

For values of σ^q still suitable for a direct access table, a useful technique is to lazily allocate only the rows of B that have at least one nonzero element. Let $g_q(P)$ be the number of distinct q-grams in P; then $g_q(P) \leqslant \min(\sigma^q, m-q+1)$. We can have at most $g_q(P)-1$ nonzero rows in B (there is no transition starting from the last q-gram), which can be significantly less than σ^q. The resulting space complexity is then $\mathcal{O}(\sigma^q + (g_q(P)\sigma + \sigma^q)k)$.

An approach suitable to store the tables B and T, for arbitrary values of q, is to use a hash table, where the keys are the q-grams of the pattern. The main problem that arises when engineering a hash table is to choose a good hash function. Moreover, as we require that lookup in the searching phase be as fast as possible, the hash function must also be very efficient to compute. Since we are using overlapping q-grams, a given q-gram shares $q-1$ symbols with the previous one. To exploit this redundancy, we need a hash function that allows a recursive computation of the hash value of a generic q-gram Xb starting from the hash value of the previous q-gram aX, for $|X| \in \Sigma^{q-1}$ and $a, b \in \Sigma$. A method that satisfies such a requirement is hashing by integer division, which has been used in the Karp-Rabin string matching algorithm [6]. In this case, the hash function has the following definition

$$h(s_0, s_1, \dots, s_{q-1}) = \sum_{j=0}^{q-1} r^{q-j-1} ord(s_j) \bmod n, \tag{3}$$

where the radix r and n are parameters and $ord : \Sigma \to \mathbb{N}$ is a function that maps a symbol to a number in the range $0,\dots,r-1$. For a given string s, the hash values of its overlapping q-grams can then be computed by using the following recursive definition

1. $h(s^q_0) = \sum_{j=0}^{q-1} r^{q-j-1} ord(s_j) \bmod n$;
2. $h(s^q_i) = (rh(s^q_{i-1}) + ord(s_{i+q-1}) - r^q ord(s_{i-1})) \bmod n$

for $i = 1,\dots,|s|-q$. The radix r is usually chosen in $\mathbb{Z}^*_n$ in such a way that the cycle length $\min\{k \mid r^k \equiv 1 \pmod n\}$ is maximal (if the cycle length is smaller than the length of the string, permutations of the same string could have the same hash value). As our domain is the set of strings of fixed length q, in this case it is enough to ensure that the cycle length is at least q. Another possibility would be to use hashing by cyclic polynomial, which is described in [3] together with an in-depth survey of recursive hashing functions for q-grams. The space complexity of this approach is $\mathcal{O}(g_q(P)((\sigma+1)k+q))$. If we handle collisions with chaining, the time complexity gets an additional multiplicative term equal to $(1+\alpha)q$, where α is the load factor. The q term in both the space and time complexities is due to the fact that, for each inserted q-gram, we have to store also the original string and, on searching, when an entry's hash matches, we have to compare the full q-gram. For small values of q, this overhead is negligible. If $q \log \sigma \in \mathcal{O}(w)$, where w is the word size in bits, the check can be performed in constant time by storing in the hash table, for each inserted q-gram $s_0 s_1, \dots, s_{q-1}$, its signature $h(s_0, s_1, \dots, s_{q-1})$, with $r = \sigma$ and $n = \sigma^q$, instead of the original string and then computing incrementally the signatures of the q-grams of the text as shown above.

Table 1
Experimental results on the King James version of the Bible ($\sigma = 63$).

Algorithm	Pattern length							
	32	64	128	256	512	1024	2048	4096
LBNDM	2.30	1.25	0.87	0.65	0.48	0.47	1.81	10.34
BNDM	1.74	1.79	1.80	1.76	1.76	1.74	1.74	1.76
BNDM*	2.47	2.49	2.04	1.90	1.83	2.03	2.64	4.93
F-BNDM	2.18	1.52	1.15	0.92	0.92	0.89	0.90	0.89
F-BNDM*	2.40	1.55	1.16	1.50	1.39	1.46	2.78	1.93
F-BNDM$_2$	2.38	1.72	1.27	0.86	0.54	0.53	0.53	0.54
F-BNDM$_3$	1.77	1.21	1.03	0.67	0.48	0.44	**0.46**	**0.45**
F-BNDM$_4$	**1.30**	**0.93**	**0.74**	**0.53**	**0.41**	**0.42**	0.49	0.48
Shift-And	7.50	7.51	7.51	7.51	7.51	7.51	7.51	7.51
F-Shift-And	21.69	21.68	21.69	21.69	21.69	21.69	21.69	21.67
Shift-And*	7.52	41.85	63.54	113.29	210.95	406.62	797.09	1603.40
F-Shift-And*	23.71	23.68	23.70	81.58	119.37	157.97	248.57	438.10

Table 2
Experimental results on a protein sequence from the *Saccharomyces cerevisiae* genome ($\sigma = 20$).

Algorithm	Pattern length							
	32	64	128	256	512	1024	2048	4096
LBNDM	1.36	0.78	0.49	0.35	0.29	0.44	7.38	35.25
BNDM	0.85	0.85	0.85	0.87	0.87	0.85	0.87	0.87
BNDM*	1.41	1.30	1.09	1.01	1.01	1.15	1.65	3.82
F-BNDM	1.03	0.69	0.48	0.44	0.42	0.41	0.40	0.41
F-BNDM*	1.23	0.73	0.75	0.86	0.93	1.71	2.18	1.46
F-BNDM$_2$	0.97	0.70	0.51	0.35	0.26	**0.26**	**0.27**	**0.26**
F-BNDM$_3$	0.88	0.56	0.40	0.24	0.24	0.33	0.54	0.64
F-BNDM$_4$	**0.71**	**0.45**	**0.29**	**0.21**	**0.22**	0.33	0.62	2.83
Shift-And	5.38	5.37	5.37	5.37	5.37	5.37	5.37	5.37
F-Shift-And	15.39	15.38	15.37	15.39	15.38	15.38	15.38	15.37
Shift-And*	5.39	30.02	45.50	81.22	151.28	291.47	571.06	1149.48
F-Shift-And*	16.79	16.78	41.45	61.81	89.37	126.46	198.08	353.08

6. Experimental results

In this section we present and comment the experimental results relative to an extensive comparison of the BNDM and F-BNDM algorithms and of the Shift-And and F-Shift-And algorithms. In particular, 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 Shift-And and BNDM algorithms, the prefix pattern of length $\min(m, w)$ is chosen, while in the case of the F-Shift-And and F-BNDM algorithms 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 split, if needed, over multiple machine words. The resulting implementations are referred to in Tables 1–4 as Shift-And*, F-Shift-And*, BNDM* and F-BNDM*. We also implemented *single word* versions of F-BNDM, named F-BNDM$_q$, that use the q-grams based 1-factorization, for $q \in \{2, 3, 4\}$. q-Grams are indexed using a hash table of size 2^{12}, with collisions resolution by chaining; the hash function used is (3), with parameters $r = 131$ and $n = 2^w$. Note that, since the hash is exactly w bits long for all values of q, there is no sensible loss in performance in using larger values of q for a fixed word size.

We have also included in our tests the LBNDM algorithm [9]. 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 threshold beyond which its performance degrades significantly.

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 automaton 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 $\min(w, m)$: hence 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 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.

Table 3
Experimental results on a genome sequence of *Escherichia coli* ($\sigma = 4$).

Algorithm	Pattern length							
	32	64	128	256	512	1024	2048	4096
LBNDM	2.92	2.23	2.39	7.69	78.87	100.24	95.99	95.88
BNDM	2.88	2.86	2.86	2.84	2.83	2.85	2.87	2.87
BNDM*	3.66	3.98	3.43	3.12	3.13	3.26	3.92	6.20
F-BNDM	3.73	2.15	1.97	1.92	1.87	1.84	1.83	1.82
F-BNDM*	3.86	2.64	4.16	3.10	3.09	3.84	2.42	3.61
F-BNDM$_2$	4.97	2.92	2.27	1.98	1.98	1.98	1.95	1.97
F-BNDM$_3$	3.63	2.45	2.04	1.14	1.02	0.99	1.40	1.03
F-BNDM$_4$	**2.04**	**1.63**	**1.57**	**0.96**	**0.65**	**0.61**	**0.61**	**0.62**
Shift-And	8.63	8.63	8.63	8.63	8.63	8.63	8.63	8.64
F-Shift-And	24.65	24.65	24.63	24.64	24.65	24.65	24.63	24.65
Shift-And*	8.65	47.87	72.78	129.89	241.83	466.42	913.41	1837.81
F-Shift-And*	26.85	41.88	102.89	134.04	198.85	320.72	546.19	1026.54

Table 4
(A) The length of the longest substring 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.

(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

All algorithms have been implemented in the C programming language and have been compiled with the `GNU C Compiler`, using the optimization options `-O3`. All tests have been performed on a 2.33 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[2] For each input file, we have generated sets of 100 patterns of fixed length m randomly extracted from the text, for m ranging over the values 32, 64, 128, 256, 512, 1024, 2048, 4096. In Tables 1–4 we report, for each set of patterns, the mean over the running times of the 100 runs.

The F-Shift-And variant turns out to be slower than Shift-And in the single word case, since the overhead due to a more complex bit-parallel simulation is not paid off by the reduction of the number of calls to the verification phase. Instead, the F-Shift-And* variant is faster than the classical Shift-And* algorithm in all cases, for $m \geqslant 64$. Observe that the multi-word versions of both algorithms are always slower than the corresponding single word versions.

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 faster than F-BNDM, except for very long patterns, as the threshold 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 F-BNDM algorithm, the difference is small. In almost all the tests, the q-grams versions of F-BNDM achieve the best running times. It is worth observing that, in the case of file (ii), the number of distinct q-grams in the patterns is very high on average, and thus a small value of q yields more stable results. Instead, in the case of file (iii), where the alphabet size is small and thus also the number of distinct q-grams is small, the version with $q = 4$ is the fastest one. It turns out that F-BNDM$_4$ is the fastest algorithm also in the case of file (i) (natural language), where, despite the large alphabet size, the number of distinct q-grams in the patterns is small on average.

7. 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 sin-

[1] http://corpus.canterbury.ac.nz/.
[2] http://data-compression.info/Corpora/ProteinCorpus/.

gle 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.

It would be interesting to investigate whether there exist other factorizations that can be used to obtain higher compactness. 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. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Comm. ACM 35 (10) (1992) 74–82.
[2] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 762–772.
[3] J.D. Cohen, Recursive hashing functions for n-grams, ACM Trans. Inf. Syst. 15 (3) (1997) 291–320.
[4] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[5] K. Fredriksson, Shift-or string matching with super-alphabets, Inform. Process. Lett. 87 (4) (2003) 201–204.
[6] R.M. Karp, M.O. Rabin, Efficient randomized pattern-matching algorithms, IBM J. Res. Dev. 31 (2) (1987) 249–260.
[7] D.E. Knuth, J.H. Morris Jr., V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1) (1977) 323–350.
[8] G. Navarro, M. Raffinot, Fast and flexible string matching by combining bit-parallelism and suffix automata, J. Exp. Algorithmics 5 (2000) 4.
[9] H. Peltola, J. Tarhio, Alternative algorithms for bit-parallel string matching, in: SPIRE, 2003, pp. 80–94.
[10] L. Salmela, J. Tarhio, J. Kytöjoki, Multipattern string matching with q-grams, J. Exp. Algorithmics 11 (2006), Article 1.1.
[11] S. Wu, U. Manber, Fast text searching allowing errors, Comm. ACM 35 (10) (1992) 83–91.