Efficient Variants of the Backward-Oracle-Matching Algorithm

Simone Faro[1] and Thierry Lecroq[2]

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

Abstract. In this article we present two efficient variants of the BOM string matching algorithm which are more efficient and flexible than the original algorithm. We also present bit-parallel versions of them obtaining an efficient variant of the BNDM algorithm. Then we compare the newly presented algorithms with some of the most recent and effective string matching algorithms. It turns out that the new proposed variants are very flexible and achieve very good results, especially in the case of large alphabets.

Keywords: string matching, experimental algorithms, text processing, automaton

1 Introduction

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the *string matching problem* consists in finding *all* occurrences of the pattern p in the text t. It is an extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry, etc.

Many string matching algorithms have been proposed over the years (see [8]). The Boyer-Moore algorithm [5] deserves a special mention, since it has been particularly successful and has inspired much work.

Automata play a very important role in the design of efficient pattern matching algorithms. For instance the well known Knuth-Morris-Pratt algorithm [14] uses a deterministic automaton that searches a pattern in a text by performing its transitions on the text characters. The main result relative to the Knuth-Morris-Pratt algorithm is that its automaton can be constructed in $\mathcal{O}(m)$-time and -space, whereas pattern search takes $\mathcal{O}(n)$-time.

Automata based solutions have been also developed to design algorithms which have optimal sublinear performance on average. This is done by using factor automata [4,9,3,1], data structures which identify all factors of a word. Among the algorithms which make use of a factor automaton the BOM (Backward Oracle Matching) algorithm [1] is the most efficient, especially for long patterns. Another algorithm based on the bit-parallel simulation [2] of the nondeterministic factor automaton, and called BNDM (Backward Nondeterministic Dawg Match) algorithm [16], is very efficient for short patterns.

In this article we present two efficient variations of the BOM string matching algorithm which turn out to be more efficient and flexible than the original BOM algorithm. We also present a bit-parallel version of the previous solution which efficiently extends the BNDM algorithm.

Simone Faro, Thierry Lecroq: Efficient Variants of the Backward-Oracle-Matching Algorithm, pp. 146–160.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

The article is organized as follows. In Section 2 we introduce basic definitions and the terminology used along the paper. In Section 3 we survey some of the most effective string matching algorithms. Next, in Section 4, we introduce two new variations of the BOM algorithm. Experimental data obtained by running under various conditions all the algorithms reviewed are presented and compared in Section 5. Finally, we draw our conclusions in Section 6.

2 Basic Definitions and Terminology

A string p of length m is represented as a finite array $p[0\,..\,m-1]$, with $m \geq 0$. In particular, for $m = 0$ we obtain the empty string, also denoted by ε. By $p[i]$ we denote the $(i+1)$-st character of p, for $0 \leq i < m$. Likewise, by $p[i\,..\,j]$ we denote the substring of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, for $0 \leq i \leq j < m$. Moreover, for any $i, j \in \mathbb{Z}$, we put $p[i\,..\,j] = \varepsilon$ if $i > j$ and $p[i\,..\,j] = p[\mathbf{max}(i, 0), \mathbf{min}(j, m-1)]$ if $i \leq j$. A substring of the form $p[0\,..\,i]$ is called a *prefix* of p and a substring of the form $p[i\,..\,m-1]$ is called a *suffix* of p for $0 \leq i \leq m-1$. For any two strings u and w, we write $w \sqsupset u$ to indicate that w is a suffix of u. Similarly, we write $w \sqsubset u$ to indicate that w is a prefix of u. The *reverse* of a string $p[0\,..\,m-1]$ is the string built by the concatenation of its letters from the last to the first: $p[m-1]p[m-2]\cdots p[1]p[0]$.

A Finite State Automaton is a tuple $A = \{Q, q_0, F, \Sigma, \delta\}$, where Q is the set of states of the automaton, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, Σ is the alphabet of characters labeling transitions and $\delta() : (Q \times \Sigma) \to Q$ is the transition function. If $\delta(q, c)$ is not defined for a state $q \in Q$ and a character $c \in \Sigma$ we say that $\delta(q, c)$ is an undefined transition and write $\delta(q, c) = \perp$.

Let t be a text of length n and let p be a pattern of length m. When the character $p[0]$ is aligned with the character $t[s]$ of the text, so that the character $p[i]$ is aligned with the character $t[s+i]$, for $i = 0, \ldots, m-1$, we say that the pattern p has *shift* s in t. In this case the substring $t[s\,..\,s+m-1]$ is called the *current window* of the text. If $t[s\,..\,s+m-1] = p$, we say that the shift s is *valid*.

Most string matching algorithms have the following general structure. First, during a *preprocessing phase*, they calculate useful mappings, in the form of tables, which later are accessed to determine nontrivial shift advancements. Next, starting with shift $s = 0$, they look for all valid shifts, by executing a *matching phase*, which determines whether the shift s is valid and computes a *positive* shift increment.

For instance, in the case of the naive string matching algorithm, there is no preprocessing phase and the matching phase always returns a unitary shift increment, i.e. all possible shifts are actually processed.

In contrast the Boyer-Moore algorithm [5] checks whether s is a valid shift, by scanning the pattern p from right to left and, at the end of the matching phase, it computes the shift increment as the maximum value suggested by two heuristics: the *good-suffix heuristic* and the *bad-character heuristic*, provided that both of them are applicable (see [8]).

3 Very Fast String Matching Algorithms

In this section we briefly review the BOM algorithm and other efficient algorithms for exact string matching that have been recently proposed. In particular, we present

algorithms in the Fast-Search family [7], algorithms in the q-Hash family [15] and some among the most efficient algorithms based on factor automata.

3.1 Fast-Search and Forward-Fast-Search Algorithms

The Fast-Search algorithm [6] is a very simple, yet efficient, variant of the Boyer-Moore algorithm. Let p be a pattern of length m and let t be a text of length n over a finite alphabet Σ. The Fast-Search algorithm computes its shift increments by applying the bad-character rule if and only if a mismatch occurs during the first character comparison, namely, while comparing characters $p[m-1]$ and $t[s+m-1]$, where s is the current shift. Otherwise it uses the good-suffix rule.

The Forward-Fast-Search algorithm [7] maintains the same structure of the Fast-Search algorithm, but it is based upon a modified version of the good-suffix rule, called *forward good-suffix* rule, which uses a look-ahead character to determine larger shift advancements. Thus, if the first mismatch occurs at position $i < m-1$ of the pattern p, the forward good-suffix rule suggests to align the substring $t[s+i+1\,..\,s+m]$ with its rightmost occurrence in p preceded by a character different from $p[i]$. If such an occurrence does not exist, the forward good-suffix rule proposes a shift increment which allows to match the longest suffix of $t[s+i+1\,..\,s+m]$ with a prefix of p. This corresponds to advance the shift s by $\overrightarrow{gs}_P(i+1, t[s+m])$ positions, where

$$\begin{aligned}\overrightarrow{gs}_P(j,c) =_{\text{Def}} \mathbf{min}(\{0<k\le m \;\mid\; & p[j-k\,..\,m-k-1] \sqsupset p \\ & \text{and } (k \le j-1 \rightarrow p[j-1] \ne p[j-1-k]) \\ & \text{and } p[m-k]=c\} \;\cup\; \{m+1\}) \;,\end{aligned}$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$.

The good-suffix rule and the forward good-suffix rule require tables of size m and $m \cdot |\Sigma|$, respectively. These can be constructed in time $\mathcal{O}(m)$ and $\mathcal{O}(m \cdot \mathbf{max}(m, |\Sigma|))$, respectively.

More effective implementations of the Fast-Search and Forward-Fast-Search algorithm are obtained along the same lines of the Tuned-Boyer-Moore algorithm [13] by making use of a fast-loop, using a technique described in Section 4.1 and shown in Figure 3(A). Then subsequent matching phase can start with the $(m-2)$-*nd* character of the pattern. At the end of the matching phase the algorithms uses the good-suffix rule for shifting.

3.2 The *q*-Hash Algorithms

Algorithms in the q-Hash family have been introduced in [15] where the author presented an adaptation of the Wu and Manber multiple string matching algorithm [18] to single string matching problem.

The idea of the q-Hash algorithm is to consider factors of the pattern of length q. Each substring w of such a length q is hashed using a function h into integer values within 0 and 255. Then the algorithm computes in a preprocessing phase a function $\mathit{shift}() : \{0, 1, \ldots, 255\} \rightarrow \{0, 1, \ldots, m-q\}$. Formally for each $0 \le c \le 255$ the value $\mathit{shift}(c)$ is defined by

$$\mathit{shift}(c) = \mathbf{min}\Big(\{0 \le k < m-q \mid h(p[m-k-q\,..\,m-k-1]) = c\} \cup \{m-q\}\Big) \;.$$

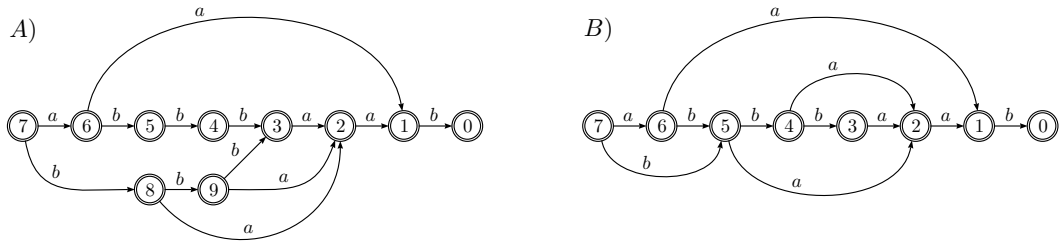


Figure 1. The factor automaton (A) and the factor oracle (B) of the reverse of pattern $p = baabbba$. The factor automaton recognizes all, and only, the factors of the reverse pattern. On the other hand note that the word aba is recognized by the factor oracle whereas it is not a factor.

The searching phase of the algorithm consists of reading, for each shift s of the pattern in the text, the substring $w = t[s+m-q\,..\,s+m-1]$ of length q. If $shift[h(w)] > 0$ then a shift of length $shift[h(w)]$ is applied. Otherwise, when $shift[h(w)] = 0$ the pattern x is naively checked in the text. In this case a shift of length sh is applied where $sh = m - 1 - i$ with

$$i = max\{0 \leq j \leq m - q | h(x[j\,..\,j+q-1]) = h(x[m-q+1\,..\,m-1]\}.$$

3.3 The Backward-Automaton-Matching Algorithms

Algorithms based on the Boyer-Moore strategy generally try to match suffixes of the pattern but it is possible to match some prefixes or some factors of the pattern by scanning the current window of the text from right to left in order to improve the length of the shifts. This can be done by the use of factor automata and factor oracles.

The factor automaton [4,9,3] of a pattern p, $Aut(p)$, is also called the factor DAWG of p (for Directed Acyclic Word Graph). Such an automaton recognizes all the factors of p. Formally the language recognized by $Aut(p)$ is defined as follows

$$\mathcal{L}(Aut(p)) = \{u \in \Sigma^* : \text{ exists } v, w \in \Sigma^* \text{ such that } p = vuw\}.$$

The factor oracle of a pattern p, $Oracle(p)$, is a very compact automaton which recognizes at least all the factors of p and slightly more other words. Formally $Oracle(p)$ is an automaton $\{Q, m, Q, \Sigma, \delta\}$ such that

1. Q contains exactly $m + 1$ states, say $Q = \{0, 1, 2, 3, \ldots, m\}$
2. m is the initial state
3. all states are final
4. the language accepted by $Oracle(p)$ is such that $\mathcal{L}(Aut(p)) \subseteq \mathcal{L}(Oracle(p))$.

Despite the fact that the factor oracle is able to recognize words that are not factors of the pattern, it can be used to search for a pattern in a text since the only factor of p of length greater or equal to m which is recognized by the oracle is the pattern itself. The computation of the oracle is linear in time and space in the length of the pattern.

In Figure 1 are shown the factor automaton and the factor oracle of the reverse of pattern $p = baabbba$.

The data structures factor automaton and factor oracle are used respectively in [10,11] and in [1] to get optimal pattern matching algorithms on the average. The algorithm which makes use of the factor automaton of the reverse pattern is called

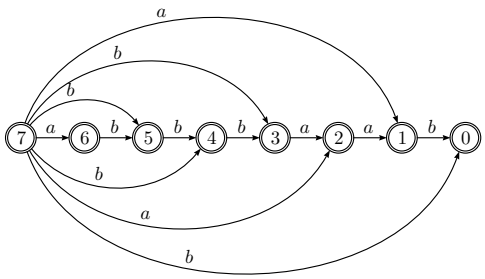


Figure 2. The nondeterministic factor automaton of the string *abbbaab*.

BDM (for Backward Dawg Matching) while the algorithm using the factor oracle is called BOM (for Backward Oracle Matching). Such algorithms move 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 and BOM algorithms is that if its 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 factor of length m is recognized then we have found an occurrence of the pattern.

The BDM and BOM algorithms have a quadratic worst case time complexity but are optimal in average since they perform $\mathcal{O}(n(log_\sigma m)/m)$ inspections of text characters reaching the best bound shown by Yao [19] in 1979.

3.4 The BNDM Algorithm

The BNDM algorithm [16] (for Backward Nondeterministic Dawg Match) is a bit-parallel simulation [2] of the BDM algorithm. It uses a nondeterministic automaton instead of the deterministic one in the BDM algorithm.

Figure 2 shows the nondeterministic version of the factor automaton for the reverse of pattern $p = baabbba$. For each character $c \in \Sigma$, a bit vector $B[c]$ is initialized in the preprocessing phase. The i-th bit is 1 in this vector if c appears in the reversed pattern in position i. Otherwise the i-th bit is set to 0. The state vector D is initialized to 1^m. The same kind of right to left scan in a window of size m is performed as in the BOM algorithm while the state vector is updated in a similar fashion as in the Shift-And algorithm [2]. If the m-th bit is 1 after this update operation, we have found a prefix starting at position j where j is the number of updates done in this window. Thus if j is the first position in the window, a match has been found.

A simplified version of the BNDM, called SBNDM, as been presented in [12]. This algorithm differs from the original one in the main loop which starts each iteration with a test of two consecutive text characters. Moreover it implements a fast-loop to obtain better results on average. Experimental results show that this simplified variant is always more efficient than the original one.

4 New Variations of the BOM Algorithm

In this section we present two variations of the BOM algorithm which perform better in most cases. The first idea consists in extending the BOM algorithm with a fast-loop over oracle transitions, along the same lines of the Tuned-Boyer-Moore algorithm [13]. Thus we are able to perform factor searching if and only if a portion of the pattern

has already matched against a portion of the current window of the text. We present this idea in Section 4.1.

An other efficient variation of the BOM algorithm can be obtained by applying the idea suggested by Sunday in the Quick-Search algorithm and then implemented in the Forward-Fast-Search algorithm. This consists in taking into account, while shifting, the character which follows the current window of the text, since it is always involved in the next alignment. Such a variation is presented in Section 4.2.

4.1 Extending the BOM Algorithm with a Fast-Loop

In this section we present an extension of the BOM algorithm by introducing a fast-loop with the aim of obtaining better results on the average. We discuss the application of different variations of the fast-loop, listed in Figure 3, and present experimental results in order to identify the best choice.

The idea of a fast loop has been proposed in [5]. The fast-loop we are using here has first introduced in the Tuned-Boyer-Moore algorithm [13] and later largely used in almost all variations of the Boyer-Moore algorithm. Generally a fast-loop is implemented by iterating the bad character heuristic in a checkless cycle, in order to quickly locate an occurrence of the rightmost character of the pattern. Suppose $bc() : \Sigma \to \{0, 1, \ldots, m\}$ is the function which implements the bad-character heuristic defined, for all $c \in \Sigma$, by

$$bc(c) = \mathbf{min}(\{0 \leq k < m \mid p[m-1-k] = c\} \cup \{m\}) \ .$$

If we suppose that $t[j]$ is the rightmost character of the current window of the text for a shift s, i.e. $j = s + m - 1$, then the original fast-loop can be implemented in a form similar to that presented in Figure 3(A).

In order to avoid testing the end of the text we could append the pattern at the end of the text, i.e. set $t[n \,..\, n+m-1]$ to p. Thus we exit the algorithm only when an occurrence of p is found. If this is not possible (because memory space is occupied) it is always possible to store $t[n-m \,..\, n-1]$ in z then set $t[n-m \,..\, n-1]$ to p and check z at the end of the algorithm without slowing it.

However algorithms based on the bad character heuristic obtain good results only in the case of large alphabets and short patterns. It turns out moreover from experimental results [15] that the strategy of using an automaton to match prefixes or factors is much better when the length of the pattern increases.

This behavior is due to the fact that for large patterns an occurrence of the rightmost character of the window, i.e. $t[j]$, can be found in the pattern and the probability that the rightmost occurrence is near the rightmost position increases for longer patterns and smaller alphabets. In this latter case an iteration of the fast loop leads to a short shift. In contrast, when using an oracle for matching, it is common that after a small number of characters we are not able to perform other transitions. So generally this strategy looks for a number of characters greater than 1, for each iteration, but leads to shift of length m.

As a first step we can translate the idea of the fast-loop over to automaton transitions. This consists in shifting the pattern along the text with no more check until a non-undefined transition is found with the rightmost character of the current window of the text. This can be translated in the fast-loop presented in Figure 3(B).

It turns out from experimental results presented in Figure 4 that the variation of the BOM algorithm which uses the fast-loop on transitions (col.B) performs better

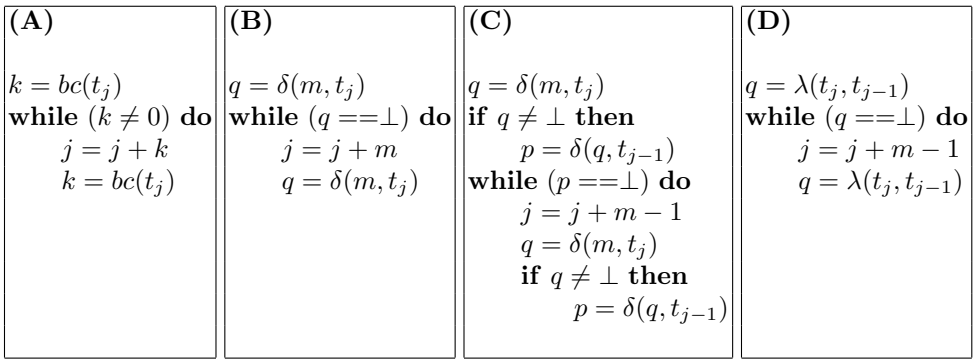


Figure 3. Different variations of the fast-loop where $t_j = t_{s+m-1}$ is the rightmost character of the current window of the text. (A) The original fast-loop based on the bad character rule. (B) A modified version of the fast-loop based on automaton transitions. (C) A fast-loop based on two subsequent automaton transitions. (D) An efficient implementation of the previous fast loop which encapsulate two subsequent transitions in a single λ table.

Experimental results with $\sigma = 8$

m	BOM	(A)	(B)	(C)	(D)
4	157.62	95.95	135.95	109.03	55.35
8	85.48	58.66	78.70	58.63	34.16
16	43.04	43.36	43.00	37.15	26.82
32	26.63	35.00	28.29	25.93	21.25
64	17.39	28.05	17.13	17.00	14.42
128	15.28	23.68	15.75	15.87	12.87
256	10.79	19.86	9.60	9.76	8.53
512	6.18	14.29	6.11	5.76	4.76
1024	3.29	8.20	3.45	3.35	2.64

Experimental results with $\sigma = 16$

m	BOM	(A)	(B)	(C)	(D)
4	103.28	66.81	86.28	93.53	40.63
8	71.59	38.72	60.27	44.02	21.73
16	39.61	26.57	35.70	23.68	14.91
32	18.68	21.80	18.82	15.71	12.73
64	12.67	20.09	12.55	12.73	12.49
128	14.22	19.38	14.14	12.35	10.38
256	8.81	19.05	8.12	7.83	6.88
512	4.62	17.73	4.62	4.53	3.60
1024	2.35	11.49	2.66	2.89	2.67

Experimental results with $\sigma = 32$

m	BOM	(A)	(B)	(C)	(D)
4	78.76	55.23	57.75	88.57	37.44
8	51.68	30.37	42.03	39.84	18.59
16	35.40	19.92	30.18	20.34	12.29
32	20.62	16.12	19.34	12.20	11.58
64	12.11	14.84	11.55	10.63	11.10
128	12.60	15.63	11.26	10.01	7.46
256	7.58	16.73	6.32	5.90	3.79
512	4.29	17.90	3.73	3.83	3.20
1024	2.87	14.19	2.67	2.79	2.01

Experimental results with $\sigma = 64$

m	BOM	(A)	(B)	(C)	(D)
4	64.84	50.93	42.34	88.52	37.04
8	39.35	27.44	29.29	38.84	17.99
16	26.09	17.12	22.03	20.07	11.57
32	19.45	14.09	17.11	11.81	10.76
64	13.15	13.58	12.28	10.37	10.70
128	13.11	17.67	10.86	9.76	6.35
256	6.25	18.04	5.79	5.55	3.60
512	2.91	18.00	3.12	5.32	1.98
1024	2.71	16.89	2.58	2.42	1.57

Figure 4. Experimental results obtained by comparing the original BOM algorithm (in the first column) against variations implemented using the four fast-loop presented in Figure 3. The results have been obtained by searching 200 random patterns in a 40Mb text buffer with a uniform distribution over an alphabet of dimension σ. Running times are expressed in hundredths of seconds.

than the original algorithm (first column), especially for large alphabets. However it is not flexible since its performances decrease when the length of the pattern increases or when the dimension of the alphabet is small. This is the fast-loop finds only a small number of undefined transitions for small alphabets or long patterns.

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition. This can be translated in the fast-loop presented in Figure 3(C). From

experimental results it turns out that such a variation (Figure 4, col.C) obtains better results than the previous one only for long pattern and large alphabets. This is for each iteration of the fast-loop the algorithm performs two subsequent transitions affecting the overall performance.

To avoid this problem we could encapsulate the two first transitions of the oracle in a function $\lambda() : (\Sigma \times \Sigma) \rightarrow Q$ defined, for each $a, b \in \Sigma$, by

$$\lambda(a,b) = \begin{cases} \perp & \text{if } \delta(m,a) = \perp \\ \delta(\delta(m,a),b) & \text{otherwise.} \end{cases}$$

Thus the fast loop can be implemented as presented in Figure 3(D). At the end of the fast-loop the algorithm could start standard transitions with the Oracle from state $q = \lambda(t[j], t[j-1])$ and character $t[j-2]$. The function λ can be implemented with a two dimensional table in $\mathcal{O}(\sigma^2)$ time and space.

The resulting algorithm, here named Extended-BOM algorithm, is very fast and flexible. Its pseudocode is presented in Figure 6(A). From experimental results in Figure 4 it turns out that the Extended-BOM algorithm (col.D) is the best choice in most cases and, differently from the original algorithm, it has very good performance also for short patterns.

4.2 Looking for the Forward Character

The idea of looking for the forward character for shifting has been originally introduced by Sunday in the Quick-Search algorithm [17] and then efficiently implemented in the Forward-Fast-Search algorithm [7]. Specifically, it is based on the following observation: when a mismatch character is encountered while comparing the pattern with the current window of the text $t[s \,..\, s+m-1]$, the pattern is always shifted to the right by at least one character, but never by more than m characters. Thus, the character $t[s+m]$ is always involved in testing for the next alignment.

In order to take into account the forward character of the current window of the text without skip safe alignment we construct the *forward factor oracle* of the reverse pattern. The forward factor oracle of a word p, $FOracle(p)$, is an automaton which recognizes at least all the factors of p, eventually preceded by a word $x \in \Sigma \cup \{\varepsilon\}$. More formally the language recognized by $FOracle(p)$ is defined by

$$\mathcal{L}(FOracle(p)) = \{xw \mid x \in \Sigma \cup \{\varepsilon\} \text{ and } w \in \mathcal{L}(Oracle(p))\}$$

Observe that in the previous definition the prefix x could be the empty string. Thus if w is a word recognized by the factor oracle of p then the word cw is recognized by the forward factor oracle, for all $c \in \Sigma \cup \{\varepsilon\}$.

The forward factor oracle of a word p can be constructed, in time $\mathcal{O}(m + \Sigma)$, by simply extending the factor oracle of p with a new initial state which allows to perform transitions starting at the text character of position $s + m$ of the text, avoiding to skip valid shift alignments.

Suppose $Oracle(p) = \{Q, m, Q, \Sigma, \delta\}$, for a pattern p of length m. We construct $FOracle(p)$ by adding a new initial state $(m+1)$ and introducing transitions from state $(m+1)$. More formally, given a pattern p of length m, $FOracle(p)$ is an automaton $\{Q', (m+1), Q, \Sigma, \delta'\}$, where

1. $Q' = Q \cup \{(m+1)\}$

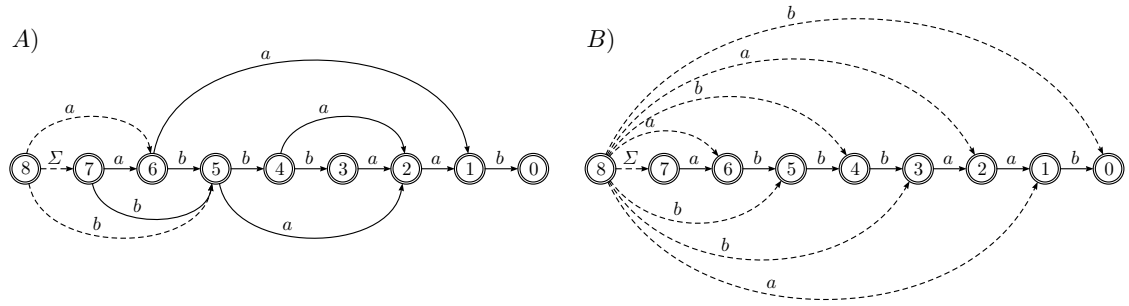


Figure 5. (A) The forward factor oracle of the reverse pattern $p = baabbba$ (B) The nondeterministic version of the forward factor automaton of the reverse pattern $p = baabbba$

2. $(m + 1)$ is the initial state
3. all states are final
4. $\delta'(q, c) = \delta(q, c)$ for all $c \in \Sigma$, if $q \neq (m + 1)$
5. $\delta'(m + 1, c) = \{m, \delta(m, c)\}$ for all $c \in \Sigma$

Figure 5(A) shows the forward factor oracle of the reverse pattern $p = baabbba$. The dashed transitions are those outgoing from the new initial state. A transition labeled with all characters of the alphabet has been introduced from state $(m + 1)$ to state m. Note that, according to rule n.5, the forward factor oracle of the reverse pattern p is a non-deterministic automaton. For example, starting from the initial state 8 in Figure 5(A), after reading the couple of characters aa, both states 6 and 1 are active.

Observe moreover that we have to read at least two consecutive characters to find an undefined transition. This is state m is always active after reading any character of the alphabet.

Suppose we start transitions from the initial state of $FOracle(p)$. Then after reading a word $w = au$, with $a \in \Sigma$ and $u \in \Sigma^+$, at most two different states could be active, i.e., state $x = \delta^*(w)$ and state $y = \delta^*(u)$. Where we recall that δ is the transition function of $Oracle(p)$ and where $\delta^*() : \Sigma^* \leftarrow Q$ is the *final state* function induced by δ and defined recursively by

$$\delta^*(w) = \delta(\delta^*(w'), c), \quad \text{for each } w = w'c, \text{ with } w' \in \Sigma^*, c \in \Sigma.$$

The idea consists in simulating the behavior of the nondeterministic forward factor oracle by following transition for only one of the two active states. More precisely we are interested only in transitions from state q where

$$q = \begin{cases} y = \delta^*(u) \text{ if } u[0] = p[m-1] \\ x = \delta^*(w) \text{ otherwise} \end{cases}$$

To prove the correctness of our strategy, suppose first we have read a word $w = au$, as defined above, and $u[0] \neq p[m-1]$. If $Oracle(p)$ recognizes a word u (i.e. $\delta^*(u) \neq \perp$) then by definition $FOracle(p)$ recognize the word au, since $a \in \Sigma \cup \{\varepsilon\}$.

Suppose now that $u[0] = p[m-1]$. If $Oracle(p)$ recognizes a word w then it recognizes also word u which is a suffix of w. Thus by definition $FOracle(p)$ recognizes the word xu, with $x = \varepsilon$.

The simulation of the forward factor oracle can be done by simply changing the computation of the λ table in the following way

(A)

```
EXTENDED-BOM(p, m, t, n)
 1.  δ ← precompute-factor-oracle(p)
 2.  for a ∈ Σ do
 3.      q ← δ(m, a)
 4.      for b ∈ Σ do
 5.          if q = ⊥ then λ(a, b) ← ⊥
 6.          else λ(a, b) ← δ(q, b)
 7.  t[n .. n + m − 1] ← p
 8.  j ← m − 1
 9.  while j < n do
10.      q ← λ(t[j], t[j − 1])
11.      while q = ⊥ do
12.          j ← j + m − 1
13.          q ← λ(t[j], t[j − 1])
14.      i ← j − 2
15.      while q ≠ ⊥ do
16.          q ← δ(q, t[i])
17.          i ← i − 1
18.      if i < j − m + 1 then
19.          output(j)
20.          i ← i + 1
21.      j ← j + i + m
```

(B)

```
FORWARD-BOM(p, m, t, n)
 1.  δ ← precompute-factor-oracle(p)
 2.  for a ∈ Σ do
 3.      q ← δ(m, a)
 4.      for b ∈ Σ do
 5.          if q = ⊥ then λ(a, b) ← ⊥
 6.          else λ(a, b) ← δ(q, b)
 7.  q ← δ(m, p[m − 1])
 8.  for a ∈ Σ do λ(a, p[m − 1]) ← q
 9.  t[n .. n + m − 1] ← p
10.  j ← m − 1
11.  while j < n do
12.      q ← λ(t[j + 1], t[j])
13.      while q = ⊥ do
14.          j ← j + m
15.          q ← λ(t[j + 1], t[j])
16.      i ← j − 1
17.      while q ≠ ⊥ do
18.          q ← δ(q, t[i])
19.          i ← i − 1
20.      if i < j − m + 1 then
21.          output(j)
22.          i ← i + 1
23.      j ← j + i + m
```

Figure 6. (A) The Extended-BOM algorithm which extend the original BOM algorithm by using an efficient fast-loop. (B) The Forward-BOM algorithm which performs a look ahead for character of position $t[j+1]$ in text to obtain larger shift advancements.

```
FORWARD-SBNDM(p, m, t, n)
 1.  for all c ∈ Σ do B[i] ← 1
 2.  for i = 0 to m − 1 do B[p[i]] ← B[p[i]] | (1 << (m − i))
 3.  j ← m − 1
 4.  while j < n do
 5.      D ← (B[t[j + 1]] << 1) & B[t[j]]
 6.      if D ≠ 0 then pos ← j
 7.      while D ← (D + D) & B[t[j − 1]] do j ← j − 1
 8.      j ← j + m − 1
 9.      if j = pos then
10.          output(j)
11.          j ← j + 1
12.      else j ← j + m
```

Figure 7. The Forward SBNDM algorithm which simulates using bit-parallelism the non deterministic forward automaton of the reverse pattern.

$$\lambda(a,b) = \begin{cases} \delta(m,b) & \text{if } \delta(m,a) = \bot \ \vee \ b = p[m-1] \\ \delta(\delta(m,a),b) & \text{otherwise} \end{cases}$$

Figure 6(B) shows the code of the Forward-Bom algorithm. Here the fast loop has been modified to take into account also the forward character of position $t[s+m]$. However if there is no transition for the first two characters, $t[s+m]$ and $t[s+m-1]$, the algorithm can shift the pattern of m position to the right. Line 1 of the preprocessing phase can be performed in $\mathcal{O}(m)$-time, lines 2 to 6 in $\mathcal{O}(\sigma^2)$ and line 8 in $\mathcal{O}(\sigma)$. Thus the preprocessing phase can be performed in $\mathcal{O}(m+\sigma^2)$ time and space.

This idea can be applied also to the SBNDM algorithm based on bit-parallelism. In this latter case we have to add a new first state and change the preprocessing in

order to perform correct transitions from the first state. Moreover we need $m + 1$ bits for representing the NFA, thus we are able to search only for patterns with $1 \leq m < w$, if w is the dimension of a machine word. Figure 7 shows the code of the Forward-SBNDM algorithm.

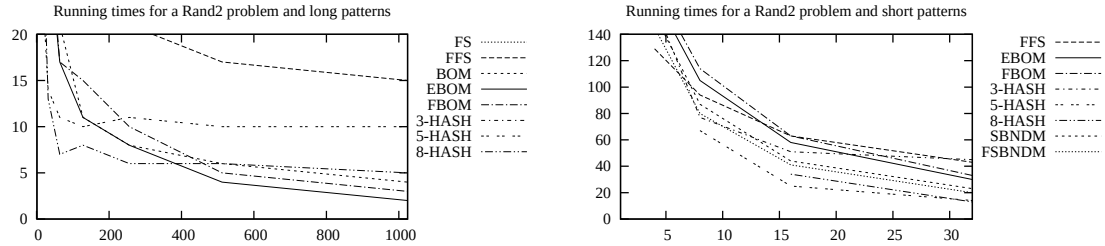
5 Experimental Results

We present next experimental data which allow to compare in terms of running time the following string matching algorithms under various conditions: Fast-Search (FS), Forward-Fast-Search (FFS), BOM (BOM), SBNDM (SBNDM), q-Hash (q-HASH with $q = 3, 5, 8$), Extended-BOM (EBOM), Forward-BOM (FBOM) and Forward-SBNDM (FSBNDM).

All algorithms have been implemented in the **C** programming language and were used to search for the same strings in large fixed text buffers on a PC with Intel Core2 processor of 1.66 GHz. In particular, the algorithms have been tested on seven Randσ problems, for $\sigma = 2, 4, 8, 16, 32, 64, 128$, on a genome, on a protein sequence and on a natural language text buffer. Searching have been performed for patterns of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$, and 1024. In the following tables, running times are expressed in hundredths of seconds.

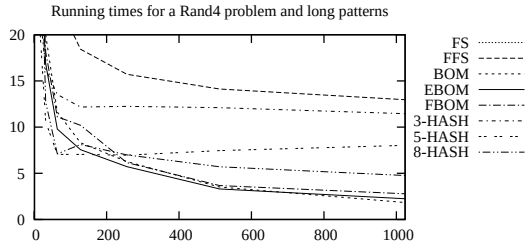
5.1 Running Times for Random Problems

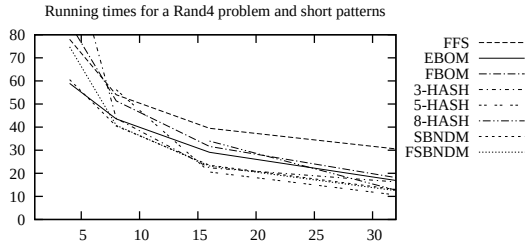
For the case of random texts the algorithms have been tested on seven Randσ problems. Each Randσ problem consists of searching a set of 400 random patterns of a given length in a 20Mb random text over a common alphabet of size σ, whit a uniform distribution of characters.



m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	153.52	**129.07**	209.07	169.22	177.31	162.98	-	-	155.38	145.47
8	115.44	94.42	133.73	105.08	114.17	77.03	**67.91**	-	87.42	80.72
16	83.60	63.05	71.75	58.91	63.23	51.65	**25.27**	34.33	44.87	41.31
32	61.96	43.40	38.55	30.58	33.24	45.38	14.85	**13.50**	23.88	20.77
64	48.16	32.69	21.24	17.43	17.91	44.65	11.53	**7.42**	-	-
128	39.55	24.90	11.91	11.73	15.63	44.02	10.09	**8.34**	-	-
256	32.80	21.14	8.45	8.43	10.00	44.92	11.02	**6.86**	-	-
512	28.07	17.27	6.36	**4.87**	5.87	45.65	10.04	6.21	-	-
1024	23.39	15.47	4.00	**2.79**	3.95	44.72	10.59	5.14	-	-

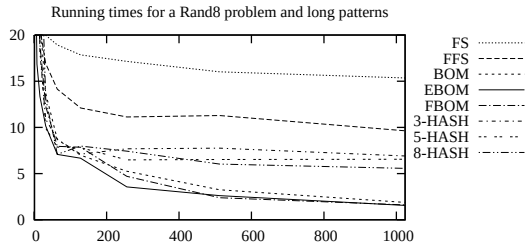
Running times for a Rand2 problem

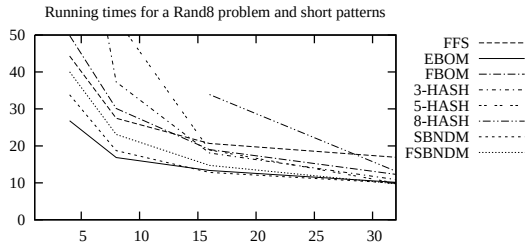




m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	82.12	78.03	111.55	**58.93**	84.93	117.82	-	-	60.57	74.73
8	60.00	54.02	61.31	43.57	51.38	43.77	56.23	-	**40.46**	40.79
16	49.05	39.49	35.58	29.11	31.66	22.40	**20.54**	33.98	23.49	23.15
32	41.72	30.56	19.98	16.88	18.13	16.27	**10.60**	12.70	12.97	12.48
64	37.11	23.71	11.63	9.79	11.11	13.53	**7.05**	7.11	-	-
128	32.02	18.43	8.30	7.56	10.20	12.17	**7.05**	8.12	-	-
256	28.54	15.72	6.27	**5.72**	6.16	12.25	6.97	6.99	-	-
512	26.07	14.13	3.52	**3.31**	3.67	12.10	7.46	5.71	-	-
1024	22.14	12.97	**1.83**	2.25	2.78	11.46	8.02	4.76	-	-

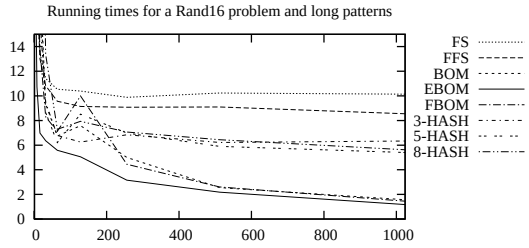
Running times for a Rand4 problem

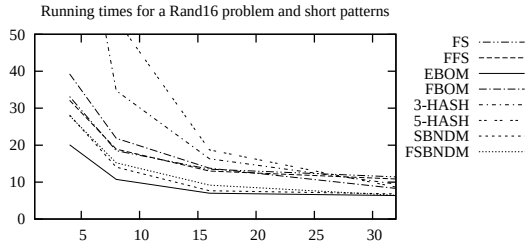




m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	46.34	44.32	78.94	**26.81**	49.87	105.12	-	-	33.82	40.05
8	29.61	27.46	43.23	**16.85**	30.11	37.30	54.47	-	18.70	23.09
16	22.46	20.67	21.72	13.34	19.01	18.07	18.90	33.90	**12.81**	14.71
32	19.97	16.91	13.70	10.15	12.29	10.89	9.92	13.14	9.92	**9.73**
64	18.93	14.14	8.70	**7.07**	7.95	8.71	7.87	7.09	-	-
128	17.85	12.10	6.99	**6.66**	7.85	7.11	7.81	7.98	-	-
256	17.15	11.13	5.26	**3.56**	4.70	7.68	6.48	7.43	-	-
512	16.02	11.29	3.25	2.61	**2.38**	7.75	6.53	6.03	-	-
1024	15.35	9.63	1.88	**1.55**	1.61	6.91	6.56	5.57	-	-

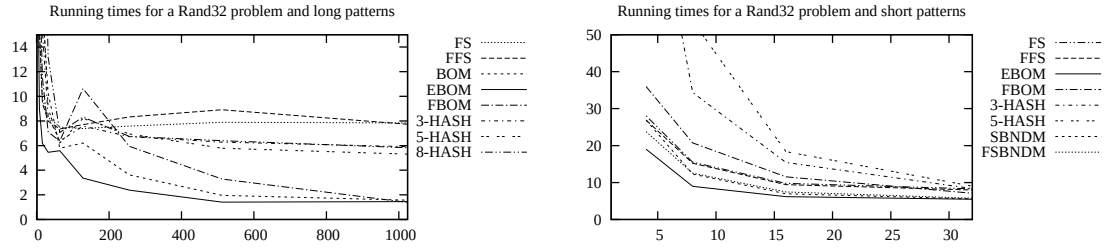
Running times for a Rand8 problem





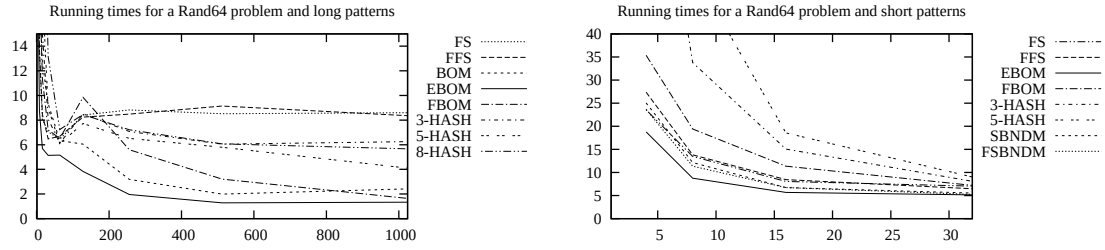
m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	33.17	32.13	52.02	**20.09**	39.26	102.31	-	-	28.16	27.98
8	18.52	18.91	35.48	**10.73**	21.87	34.74	54.09	-	14.04	15.22
16	13.48	13.01	19.61	**6.98**	13.76	16.33	18.71	33.78	7.66	9.18
32	11.41	10.83	9.33	**6.36**	8.29	9.46	8.64	13.35	6.80	6.43
64	10.54	9.57	6.74	**5.58**	7.12	6.79	6.21	7.29	-	-
128	10.39	9.14	7.58	**5.05**	9.99	6.25	8.52	7.93	-	-
256	9.88	9.08	5.00	**3.16**	4.45	6.84	6.98	7.07	-	-
512	10.23	9.10	2.55	**2.18**	2.61	6.22	5.90	6.44	-	-
1024	10.14	8.55	1.57	**1.18**	1.45	6.33	5.40	5.62	-	-

Running times for a Rand16 problem



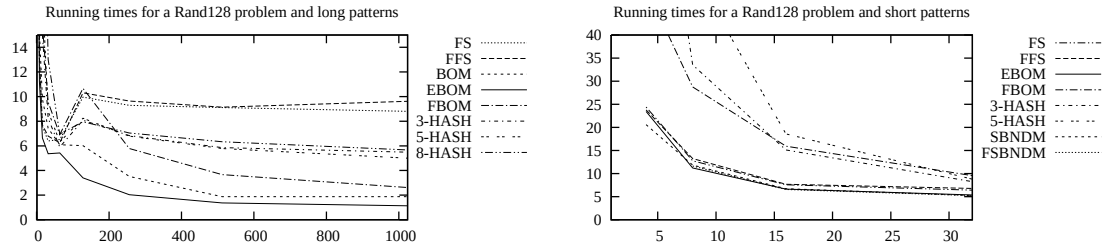
m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	28.04	26.91	35.98	**19.03**	35.98	100.51	-	-	26.88	23.75
8	15.51	15.23	24.54	**8.98**	20.74	34.34	53.71	-	12.28	12.54
16	9.78	9.44	17.46	**6.18**	11.56	15.44	18.36	34.14	6.95	7.46
32	8.29	7.98	10.26	**5.46**	7.11	8.36	9.02	13.16	5.59	5.75
64	7.50	7.35	5.78	**5.58**	6.37	6.37	6.22	7.07	-	-
128	7.38	7.70	6.21	**3.36**	10.62	7.58	8.21	8.32	-	-
256	7.59	8.33	3.62	**2.38**	5.94	6.73	6.95	6.75	-	-
512	7.89	8.91	1.96	**1.41**	3.28	6.28	5.78	6.40	-	-
1024	7.84	7.73	1.57	1.45	**1.39**	5.91	5.31	5.83	-	-

Running times for a Rand32 problem



m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	23.55	27.38	29.10	**18.79**	35.38	97.23	-	-	25.05	23.67
8	13.48	13.82	18.51	**8.76**	19.41	33.79	53.80	-	12.15	11.37
16	8.06	8.44	12.64	**5.69**	11.35	15.07	18.56	33.32	6.72	6.72
32	7.04	6.47	9.33	**5.14**	7.20	8.09	9.00	13.15	5.55	5.25
64	6.44	6.68	6.34	**5.16**	6.52	6.13	6.09	7.23	-	-
128	8.41	8.24	6.05	**3.84**	9.85	8.51	7.72	8.45	-	-
256	8.82	8.49	3.19	**1.96**	5.59	7.08	6.52	7.21	-	-
512	8.52	9.14	1.99	**1.28**	3.21	6.05	5.79	6.07	-	-
1024	8.60	8.36	2.41	**1.33**	1.64	6.25	4.10	5.67	-	-

Running times for a Rand64 problem



m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	24.42	23.79	26.16	23.45	50.15	99.57	-		27.21	**20.60**
8	12.77	13.24	15.06	**11.21**	28.77	33.54	53.99	-	11.83	11.65
16	7.58	7.69	9.81	6.64	15.92	15.10	18.59	33.84	6.72	**6.55**
32	6.40	6.79	7.43	5.36	9.56	8.24	8.79	13.01	5.25	**5.24**
64	6.52	6.25	6.09	**5.42**	6.92	5.90	6.32	6.96	-	-
128	9.96	10.31	6.02	**3.40**	10.67	8.08	8.24	7.96	-	-
256	9.29	9.64	3.52	**2.03**	5.78	6.83	6.79	7.04	-	-
512	9.11	9.14	1.87	**1.37**	3.67	5.86	5.78	6.33	-	-
1024	8.80	9.62	1.88	**1.13**	2.62	5.49	5.00	5.67	-	-

Running times for a Rand128 problem

Experimental results show that the Extended-BOM and the Forward-BOM algorithms obtain the best run-time performance in most cases. In particular for small

alphabets and short patterns the presented variations are second to algorithms in the q-Hash family. Moreover for large alphabets and short patterns algorithms based on bit-parallelism are the best choice. Note however that for alphabets of medium dimension, when the pattern is short, the performance of the Extended-BOM algorithm outperform those of bit-parallel algorithms that until now have been considered the best choice for short patterns.

5.2 Running Times for Real World Problems

The tests on real world problems have been performed on a genome sequence and on a natural language text buffer. A genome is a DNA sequence composed of the four nucleotides, also called base pairs or bases: Adenine, Cytosine, Guanine and Thymine. The genome we used for these tests is a sequence of $4,638,690$ base pairs of *Escherichia coli*. We used the file E.coli file of the Large Canterbury Corpus (http://www.data-compression.info/Corpora/CanterburyCorpus/).

The tests on the protein sequence has been performed using a 2.4Mb file containing a protein sequence from the human genome with 22 different characters.

For the experiments on the natural language text buffer we used the file world192.txt (The CIA World Fact Book) of the Large Canterbury Corpus. The alphabet is composed of 94 different characters. The text is composed of $2,473,400$ characters.

From experimental results it turns out that the Extended-BOM algorithm obtains in most cases the best results and sporadically is second to algorithms of the q-Hash family. Again better results are obtained for medium dimensions of the alphabet.

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	18.64	16.91	23.25	**12.65**	19.09	25.48	-	-	12.96	17.30
8	13.85	11.63	13.04	10.27	11.40	9.90	12.34	-	**8.73**	9.01
16	11.48	8.47	7.73	6.77	6.47	4.76	**4.39**	7.74	5.28	5.50
32	9.58	6.44	4.53	3.52	4.07	3.20	2.77	2.85	3.04	**2.62**
64	8.56	4.92	2.50	1.95	2.42	2.65	**1.60**	1.84	-	-
128	7.05	4.01	1.74	**1.73**	1.91	2.42	1.84	2.08	-	-
256	6.41	3.35	1.33	**1.32**	1.33	2.90	1.60	1.41	-	-
512	5.66	3.20	0.94	0.82	**0.78**	2.39	1.60	1.61	-	-
1024	5.97	2.19	0.98	0.66	**0.51**	2.50	1.21	1.21	-	-

Running times for a genome sequence ($\sigma = 4$)

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	4.33	2.93	8.30	**2.14**	5.51	14.49	-	-	5.19	3.59
8	**1.68**	2.64	4.21	2.27	3.58	4.38	8.09	-	2.31	1.85
16	1.71	1.57	2.66	**1.05**	1.92	2.50	2.58	4.54	1.25	1.05
32	1.41	1.47	1.62	**0.87**	1.27	1.30	1.37	1.64	0.89	0.89
64	1.21	1.02	1.10	**0.63**	1.18	0.85	0.82	1.25	-	-
128	1.09	1.33	1.13	**0.67**	1.51	0.98	1.14	1.22	-	-
256	1.37	1.44	0.59	0.51	**0.47**	0.90	0.90	0.82	-	-
512	1.20	1.56	0.50	**0.27**	0.30	0.77	0.90	0.88	-	-
1024	1.25	1.64	0.39	0.35	**0.27**	0.87	0.70	0.74	-	-

Running times for a protein sequence ($\sigma = 22$)

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	3.66	3.73	5.70	**2.70**	4.79	11.70	-	-	3.65	3.25
8	2.12	2.01	3.95	**1.52**	2.44	3.83	6.50	-	1.96	1.82
16	1.54	1.29	2.73	**0.82**	1.80	2.10	1.96	3.66	1.14	1.05
32	1.14	1.09	1.35	1.06	1.29	0.95	1.60	1.05	**0.55**	0.86
64	0.91	0.82	1.14	0.82	1.45	0.70	0.66	**0.63**	-	-
128	1.10	1.17	0.86	**0.79**	1.32	0.86	0.90	0.94	-	-
256	0.93	1.28	**0.48**	0.59	0.67	0.83	0.75	0.70	-	-
512	0.78	1.21	0.59	**0.27**	0.71	0.66	0.66	0.40	-	-
1024	0.65	1.55	0.69	0.52	0.80	0.63	**0.28**	0.47	-	-

Running times for a natural language text buffer ($\sigma = 93$)

6 Conclusion

We presented two efficient variants of the Backward Oracle Matching algorithm which is considered one of the most effective algorithm for exact string matching. The first variation, called Extended-BOM, introduces an efficient fast-loop over transitions of the oracle by reading two consecutive characters for each iteration. The second variation, called Forward-BOM, extends the previous one by using a look-ahead character at the beginning of transitions in order to obtain larger shift advancements.

It turns out from experimental results that the new proposed variations are very fast in practice and obtain the best results in most cases, especially for long patterns and alphabets of medium dimension.

References

1. C. Allauzen, M. Crochemore, and M. Raffinot: *Factor oracle: a new structure for pattern matching*, in SOFSEM'99, J. Pavelka, G. Tel, and M. Bartosek, eds., LNCS 1725, Milovy, Czech Republic, 1999, Springer-Verlag, Berlin, pp. 291–306.
2. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.
3. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theor. Comput. Sci., 40(1) 1985, pp. 31–55.
4. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel: *Linear size finite automata for the set of all subwords of a word: an outline of results.* Bull. Eur. Assoc. Theor. Comput. Sci., 21 1983, pp. 12–20.
5. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
6. D. Cantone and S. Faro: Fast-Search*: a new efficient variant of the Boyer-Moore string matching algorithm.* WEA 2003, LNCS 2647(4/5) 2003, pp. 247–267.
7. D. Cantone and S. Faro: *Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm.* J. Autom. Lang. Comb., 10(5/6) 2005, pp. 589–608.
8. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's College Publications, 2004.
9. M. Crochemore: *Optimal factor tranducers*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., vol. 12 of NATO Advanced Science Institutes, Series F, Springer-Verlag, Berlin, 1985, pp. 31–44.
10. M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter: *Speeding up two string matching algorithms.* Algorithmica, 12(4/5) 1994, pp. 247–267.
11. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
12. J. Holub and B. Durian: *Fast variants of bit parallel approach to suffix automata.* Talk given in: The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation, http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf, 2005.
13. A. Hume and D. M. Sunday: *Fast string searching.* Softw. Pract. Exp., 21(11) 1991, pp. 1221–1248.
14. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(1) 1977, pp. 323–350.
15. T. Lecroq: *Fast exact string matching algorithms.* Inf. Process. Lett., 102(6) 2007, pp. 229–235.
16. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., LNCS 1448, Piscataway, NJ, 1998, Springer-Verlag, Berlin, pp. 14–33.
17. D. M. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
18. S. Wu and U. Manber: *A fast algorithm for multi-pattern searching*, Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.
19. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM J. Comput., 8(3) 1979, pp. 368–387.