

# New experimental results on exact string-matching

Thierry Lecroq \*

## Abstract

String-matching consists in locating the occurrences of a pattern in a text. We present experimental results on the number of text character inspections and running times for seven exact string-matching algorithms. The general tendency shows that the Tuned Boyer-Moore, Reverse-Factor, Backward Oracle Matching and String Matching with a Position Tree algorithms are efficient in different conditions.

KEY WORDS: String matching; pattern matching; algorithms on words

## Introduction

String-matching is a very important subject in the wider domain of text processing. String-matching algorithms are basic components used in implementations of practical softwares existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (system or software design). Finally, they also play an important role in theoretical computer science by providing challenging problems.

Although data are memorized in various ways, text remains the main form to exchange information. This is particularly evident in literature or linguistics where data are composed of huge corpus and dictionaries. This apply as well to computer science where a large amount of data are stored in linear files. And this is also the case, for instance, in molecular biology because biological molecules can often be approximated as sequences of nucleotides or amino acids. Furthermore, the quantity of available data in these fields tend to double every eighteen months. This is the reason why algorithms should be efficient even if the speed and capacity of storage of computers increase regularly.

String-matching consists in finding one, or more generally, all the occurrences of a string (called a pattern) in a text. The pattern is denoted by  $x = x[0..m-1]$ ; its length denoted by  $|x|$  is equal to  $m$ . The text is denoted by  $y = y[0..n-1]$ ;

---

\*LIFAR-ABISS, Faculté des Sciences et Techniques, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, e-mail: [Thierry.Lecroq@univ-rouen.fr](mailto:Thierry.Lecroq@univ-rouen.fr), URL: <http://www-igm.univ-mlv.fr/~lecroq>. This work was partially supported by a NATO grant PST.CLG.977017.

its length denoted by  $|y|$  is equal to  $n$ . Both the pattern and the text are words built upon a finite alphabet denoted by  $\Sigma$  which size is equal to  $\sigma$ .

Applications require two kinds of solution depending on which string, the pattern or the text, is given first. Algorithms based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern and solve the first kind of problem. The notion of indexes realized by trees or automata is used in the second kind of solutions. We are interested here in algorithms of the first kind.

String-matching algorithms access the text character through a *window* of size  $m$ . They work as follows: they position the window on the left end of the text then compare the characters of the pattern with the characters of the window — this specific work is called an *attempt* — and after a whole match of the pattern or after a mismatch they *shift* the window to the right. They repeat the same procedure again until the window goes beyond the right end of the text. This is known as the *sliding window mechanism*. We associate each attempt with the position  $j$  in the text when the window is positioned on  $y[j - m + 1 .. j]$ .

The brute force algorithm locates all the occurrences of the pattern in time  $O(n \times m)$ . Numerous linear solutions have been designed to improve this brute force algorithm (see [1]). Some of these solutions are even sub-linear in practice. Most of the linear string-matching algorithm preprocess the pattern before the actual searching phase. One of the most famous string-matching algorithm is the Boyer-Moore algorithm [2]. It has the (justified) reputation to be very efficient in most practical cases. It gains its efficiency by scanning, at each attempt, the characters of the window from right to left, recognizing some suffix of the pattern, performing a shift and thus, in most cases, avoiding to look at all the characters of the text. Several experiments on exact string-matching have already been reported ([3], [4], [5], [6], [7], [8], [9] and [10]). In this article we compare the performances of three different implementations of the Boyer-Moore algorithm, namely the Tuned Boyer-Moore algorithm [7], the Quick Search algorithm [11] and the Berry-Ravindran algorithm [9], with algorithms using trees or automata to recognize factors of the text, namely the Skip Search algorithm [12], the Reverse Factor algorithm [13], the Backward Oracle Matching algorithm [14] and the String Matching with a Position Tree algorithm [15].

This article is organized as follows: we first describe briefly the string-matching algorithms used for the tests. We then present the experimental conditions. After that we give the results concerning the number of text character inspections followed by the running times. Finally we give our conclusions.

## The string-matching algorithms

In this section, we will briefly describe the different string-matching algorithms used in the experiments. We denote by  $per(x)$  the period of the pattern  $x$ .

$c$	a	b	c
$bmBc[c]$	1	2	7

Figure 1: Values of the bad-character shift for the Tuned Boyer-Moore algorithm for the pattern `bbabbaa`.

## Tuned Boyer-Moore algorithm

The Tuned Boyer-Moore algorithm [7] is an implementation of a simplified version of the Boyer-Moore algorithm [2] which is very fast in practice. The most costly part of a string-matching algorithm consists in checking whether the characters of the pattern match the characters of the window. To avoid doing this part too often, it is possible to unrolled several shifts, in order to locate first the rightmost character of the pattern, before actually comparing the other characters. The algorithm uses the bad-character shift function to find  $x[m - 1]$  in  $y$  and keeps on shifting, until finding it, doing blindly three shifts in a row.

When the window is positioned on  $y[j - m + 1..j]$ , the bad-character shift consists in aligning the text character  $y[j]$  with its rightmost occurrence in  $x[0..m - 2]$ . If  $y[j]$  does not occur in the pattern  $x$ , no occurrence of  $x$  in  $y$  can include  $y[j]$ , and the left end of the window is positioned on the character immediately after  $y[j]$ , namely  $y[j + 1]$ . The bad-character shift is stored in a table  $bmBc$  of size  $\sigma$ . Formally, for  $c \in \Sigma$ :

$$bmBc[c] = \begin{cases} \min\{i \mid 1 \leq i < m - 1 \text{ and } x[m - 1 - i] = c\} & \text{if } c \text{ occurs in } x, \\ m & \text{otherwise.} \end{cases}$$

An example is given figure 1.

The strategy of the Tuned Boyer-Moore algorithm requires to save the value of  $bmBc[x[m - 1]]$  in a variable *shift* and then to set  $bmBc[x[m - 1]]$  to 0. It requires also to add  $m$  occurrences of  $x[m - 1]$  at the end of  $y$  in order to be able to compute the last shifts. When  $x[m - 1]$  is found the  $m - 1$  other characters of the window are checked and after that a shift of length *shift* is applied.

The comparisons between pattern and text characters during each attempt can be done in any order. The Tuned Boyer-Moore algorithm is given figure 2. This algorithm has a quadratic worst-case time complexity but a very good practical behaviour.

## Quick Search algorithm

After an attempt where the window is positioned on  $y[j - m + 1..j]$ , the length of the shift is at least equal to one. So, the character  $y[j + 1]$  is necessarily involved in the next attempt, and thus can be used for the bad-character shift of the current attempt. The bad-character shift of the present algorithm is slightly modified to take into account the last character of  $x$  as follows ( $c \in \Sigma$ ):

$$qsBc[c] = \begin{cases} \min\{i \mid 0 \leq i < m \text{ and } x[m - 1 - i] = c\} & \text{if } c \text{ occurs in } x, \\ m & \text{otherwise.} \end{cases}$$

```

TUNED-BOYER-MOORE( $x, m, y, n$ )
1   $shift \leftarrow bmBc[x[m-1]]$ 
2   $bmBc[x[m-1]] \leftarrow 0$ 
3   $y[n..n+m-1] \leftarrow x[m-1]^m$ 
4   $j \leftarrow m-1$ 
5  while  $j < n$  do
6       $k \leftarrow bmBc[y[j]]$ 
7      while  $k \neq 0$  do
8           $j \leftarrow j+k$ 
9           $k \leftarrow bmBc[y[j]]$ 
10          $j \leftarrow j+k$ 
11          $k \leftarrow bmBc[y[j]]$ 
12          $j \leftarrow j+k$ 
13          $k \leftarrow bmBc[y[j]]$ 
14     if  $x[0..m-2] = y[j-m+1..j-1]$  and  $j < n$  then
15         REPORT( $j-m+1$ )
16      $j \leftarrow j+shift$ 

```

Figure 2: The Tuned Boyer-Moore algorithm.

$c$	$a$	$b$	$c$
$qsBc[c]$	1	3	8

Figure 3: Values of the bad-character shift for the Quick Search algorithm for the pattern `bbabbaa`.

An example is displayed figure 3.

During the searching phase the comparisons between pattern and text characters during each attempt can be done in any order. The algorithm is called Quick Search [11] after his inventor. It is shown figure 4. It has a quadratic worst-case time complexity but it has a good practical behaviour.

```

QUICK-SEARCH( $x, m, y, n$ )
1   $j \leftarrow m-1$ 
2  while  $j < n$  do
3      if  $x = y[j-m+1..j]$  then
4          REPORT( $j-m+1$ )
5       $j \leftarrow j+qsBc[y[j+1]]$ 

```

Figure 4: The Quick Search algorithm.

<i>brBc</i>	a	b	c
a	1	1	1
b	3	4	9
c	9	8	9

Figure 5: Values of the bad-character shifts for the Berry-Ravindran algorithm for the pattern `bbabbaa`.

```

BERRY-RAVINDRAN(x, m, y, n)
1  y[n + 1] ← 0
2  j ← m - 1
3  while j < n do
4      if x = y[j - m + 1 .. j] then
5          REPORT(j - m + 1)
6      j ← j + brBc[y[j + 1], y[j + 2]]

```

Figure 6: The Berry-Ravindran algorithm.

## Berry-Ravindran algorithm

Berry and Ravindran [9] designed an algorithm which performs the shift by considering the bad-character shift for the two consecutive text characters immediately to the right of the window.

The preprocessing phase of the algorithm consists in computing for each pair of characters  $(a, b)$  with  $a, b \in \Sigma$  the rightmost occurrence of  $ab$  in  $axb$ .

$$\forall a, b \in \Sigma$$

$$brBc[a, b] = \min \begin{cases} 1 & \text{if } x[m-1] = a, \\ m-i+1 & \text{if } x[i]x[i+1] = ab, \\ m+1 & \text{if } x[0] = b, \\ m+2 & \text{otherwise.} \end{cases}$$

An example is given figure 5.

The preprocessing phase is in  $O(m + \sigma^2)$  space and time complexity.

After an attempt where the window is positioned on  $y[j - m + 1 .. j]$  a shift of length  $brBc[y[j + 1], y[j + 2]]$  is performed. The text character  $y[n]$  is usually equal to the null character (in most programming languages) and  $y[n + 1]$  is set to this null character in order to be able to compute the last shifts of the algorithm.

The Berry-Ravindran algorithm is displayed figure 6. It has an  $O(m \times n)$  time complexity.

## Alpha Skip Search algorithm

The preprocessing phase of the Alpha Skip Search algorithm [12] consists in building a trie  $\mathcal{T}(x)$  of all the factors of the length  $\ell = \log_{\sigma} m$  occurring in

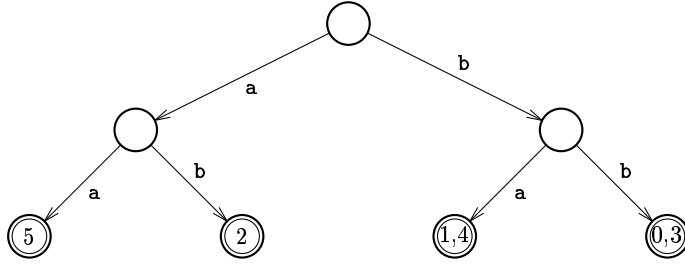


Figure 7: Trie for the pattern `bbabbaa`. The leaves contain the starting positions of their associated factor in `bbabbaa`. For instance `ba` occurs at positions 1 and 4 in `bbabbaa`.

the word  $x$ . The leaves of  $\mathcal{T}(x)$  represent all the distinct factors of length  $\ell$  of  $x$ . There is then one bucket for each leaf of  $\mathcal{T}(x)$  in which is stored the list of positions where the factor, associated with the leaf, occurs in  $x$ . The trie  $\mathcal{T}(x)$  can be considered as a Deterministic Finite Automaton  $(Q, q_0, T, \delta)$  where  $Q$  is a finite set of states (or set of nodes of the trie),  $q_0 \in Q$  is the starting state (or root of the trie),  $T \subseteq Q$  is the set of terminal states (or leaves of the trie) and  $\delta$  is the transition function. Then each element of  $T$  is associated with a distinct factor of  $x$ . We defined the table  $z$ , containing the buckets, as follows: for  $q \in T$

$$z[q] = \{i \mid \delta(q_0, x[i..i + \ell - 1]) = q\}.$$

The preprocessing phase of the Alpha Skip Search algorithm consists in computing the trie  $\mathcal{T}(x)$  and the table  $z$ . The worst case time of this preprocessing phase is linear if the alphabet size is considered to be a constant. An example is shown figure 7.

The searching phase consists in looking into the buckets of the text factors  $y[j..j + \ell - 1]$  for all  $j = k \times (m - \ell + 1) - 1$  with the integer  $k$  in the interval  $[1, \lfloor (n - \ell)/m \rfloor]$ . The Alpha Skip Search algorithm is shown figure 8.

The worst case time complexity of the searching phase is quadratic but the expected number of text character comparisons is  $O(\log_\sigma m \times (n/(m - \log_\sigma m)))$ .

## Reverse Factor algorithm

The Boyer-Moore type algorithms match some suffixes of the pattern but it is possible to match some prefixes of the pattern, reading the character of the window from right to left, and then improve the length of the shifts. This is made possible by the use of the smallest suffix automaton (also called dawg for directed acyclic word graph) of the reverse pattern. The algorithm applying this technique is called the Reverse Factor algorithm [13].

The smallest suffix automaton of a word  $w$  is a Deterministic Finite Automaton  $\mathcal{S}(w) = (Q, q_0, T, \delta)$  where  $Q$  is a finite set of states,  $q_0 \in Q$  is the starting

```

ALPHA-SKIP-SEARCH( $x, m, y, n$ )
1   $j \leftarrow m - 1 - \ell$ 
2  while  $j < n - \ell$  do
3       $q \leftarrow q_0$ 
4      for  $k \leftarrow 0$  to  $\ell - 1$  do
5           $q \leftarrow \delta(q, y[j + k])$ 
6      if  $q$  is defined then
7          for all  $i \in z[q]$  do
8              if  $x = y[j - i .. j - i + m - 1] = x$  then
9                  REPORT( $j - i$ )
10      $j \leftarrow j + m - \ell + 1$ 

```

Figure 8: The Alpha Skip Search algorithm.

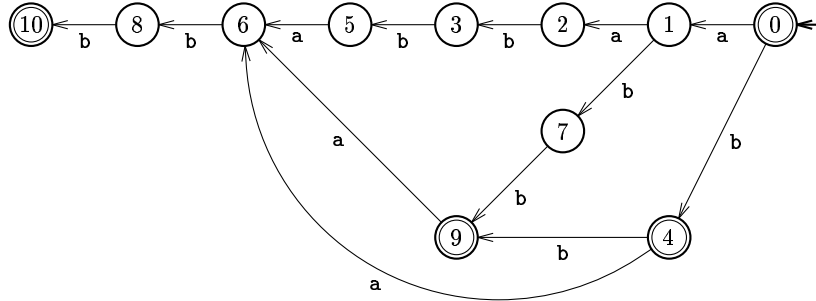


Figure 9: Suffix automaton for  $x^R = aabbabb$ .

state,  $T \subseteq Q$  is the set of terminal states and  $\delta$  is the transition function. The language accepted by  $\mathcal{S}(w)$  is  $\mathcal{L}(\mathcal{S}(w)) = \{u \in \Sigma^* \mid \exists v \in \Sigma^* \text{ such that } w = vu\}$ . The preprocessing phase of the Reverse Factor algorithm consists in computing the smallest suffix automaton for the reverse pattern  $x^R$ . It is linear in time and space in the length of the pattern (see [16] and [17]). An example is given figure 9.

During the searching phase, the Reverse Factor algorithm parses the characters of the window from right to left with the automaton  $\mathcal{S}(x^R)$  starting with state  $q_0$ . It goes until there is no more transition defined for the current character of the window from the current state of the automaton. At this moment it is easy to know what is the length of the longest prefix of the pattern which has been matched: it corresponds to the length of the path taken in  $\mathcal{S}(x^R)$  from the starting state  $q_0$  to the last final state encountered. Knowing the length of this longest prefix, it is trivial to compute the right shift to perform. The Reverse Factor algorithm is given figure 10.

The Reverse Factor algorithm has a quadratic worst case time complexity but it is optimal in average. It performs  $O(n \times (\log_\sigma m)/m)$  inspections of text characters on the average reaching the best bound shown by Yao in 1979 [18].

```

REVERSE-FACTOR( $x, m, y, n$ )
1   $j \leftarrow 0$ 
2  while  $j \leq n - m$  do
3       $i \leftarrow m - 1$ 
4       $q \leftarrow q_0$ 
5       $s \leftarrow m$ 
6      while  $\delta(q, y[i + j])$  is defined do
7           $q \leftarrow \delta(q, y[i + j])$ 
8          if  $q \in T$  then
9               $s \leftarrow j$ 
10     if  $i < 0$  then
11         REPORT( $j - m + 1$ )
12          $j \leftarrow j + per(x)$ 
13     else  $j \leftarrow j + s$ 

```

Figure 10: The Reverse Factor algorithm.

## Backward Oracle Matching algorithm

The Backward Oracle Matching algorithm [14] is a version of the Reverse Factor algorithm which uses the suffix oracle instead of the suffix automaton. The suffix oracle is a very compact automaton which recognizes at least all the suffixes of a word and slightly more other words. The suffix oracle of a word  $w$  is a Deterministic Finite Automaton  $\mathcal{O}(w) = (Q, q_0, T, \delta)$ . The language accepted by  $\mathcal{O}(w)$  is such that  $\{u \in \Sigma^* \mid \exists v \in \Sigma^* \text{ such that } w = vu\} \subseteq \mathcal{L}(\mathcal{O}(w))$ . Its construction is linear in time and space in the length of the pattern. It has exactly  $m + 1$  states and at most  $2m - 1$  transitions. Actually the oracle consists in a skeleton which is the path (of exactly  $m + 1$  states and  $m$  transitions), from the starting state to the last state, spelling the word  $w$  itself. Those transitions will not be stored and will be deduced from the word. So each prefix of the word (including the empty one) is associated with a state of the oracle. In addition to this skeleton there are a few more transitions (no more than  $m - 1$ ). Each incoming transition to a state is labelled by the last character of the prefix associated with the state. Thus those extra transitions can be represented as going from a source state to a target state and the labels can be omitted since they can be computed from the word. So the representation of the oracle consists only in the word and the set of the extra transitions without their labels. An example is displayed figure 11.

The preprocessing phase of the Backward Oracle Matching algorithm consists thus in computing the oracle for the reverse pattern  $x^R$ . The searching phase of the Backward Oracle Matching is exactly the same as the one of the Reverse Factor algorithm except that the transitions are computed differently. The fact that the oracle recognizes words that are not factor of the reverse pattern  $x^R$  is not a problem since it recognizes only one word of length  $m$  which is the reverse pattern itself.

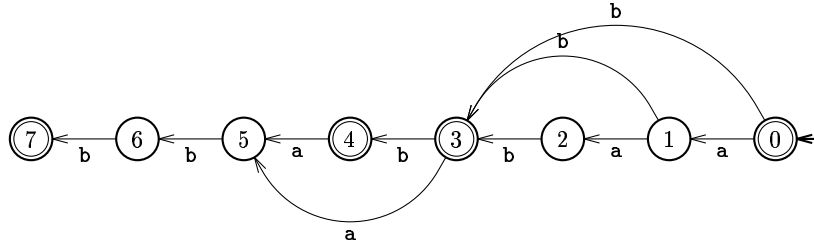


Figure 11: Suffix oracle for  $x^R = aabbabb$ . All the factors of  $x^R$  are recognized but the word  $ababb$  is recognized though it is not a factor of  $x^R$ . Only the edges  $(0, 3)$ ,  $(1, 3)$  and  $(3, 5)$  are stored in addition to the pattern  $x$ .

The Backward Oracle Matching algorithm has a quadratic worst case time complexity but it is optimal in average. It performs  $O(n(\log_\sigma m)/m)$  inspections of text characters on the average reaching the best bound shown by Yao in 1979.

### String Matching with a Position Tree algorithm

The Reverse Factor and Backward Oracle Matching algorithms scan the window from right to left as long as there is a transition defined for the current state with the current character, recognizing factors of the pattern in the text. Most of the times a shift could be performed without scanning all the characters of the factor because some position in the pattern are uniquely determined by exactly one factor. A solution consists in using a position tree [19] for the reverse pattern  $x^R$ . In the sequel we assume that a word is always terminated by a special null character 0 that does not occur elsewhere (which is often the case in programming languages). A position tree for a word  $w$  of length  $m$  has exactly  $m + 1$  leaves identifying the  $m + 1$  factors of the word beginning at each of its position. So every path of the tree from the root to a leaf identifies only one position and each branch of the tree either leads directly to a leaf or to a subtree which contains at least one fork. The position tree for  $x^R$  can be interpreted in terms of a Deterministic Finite Automaton (DFA)  $\mathcal{P} = (Q, q_0, T, \delta)$  where  $Q$  is the set of nodes of the tree,  $q_0$  is the root of the tree,  $T$  is the set of leaves and  $\delta$  is the transition function. This structure is augmented by a function  $J$  from  $T$  to  $\mathbf{N}$  which associates a starting position in  $x^R$  to each leaf. For all  $q \in T$

$$J(q) = i \text{ such that } \delta(q_0, x[i \dots i + \text{depth}(q) - 1]) = q$$

An example is given figure 12.

We are now able to describe the searching phase of the String Matching with a Position Tree algorithm [15]. Each attempt of the algorithm scans a text portion from right to left until there is no more transition defined from the current node with the current text character. The process always starts with the root as the current node. During this process, the  $J$  value of the last node  $q$  is remembered such that  $\delta(q, 0)$  is defined. This value is used to perform a shift

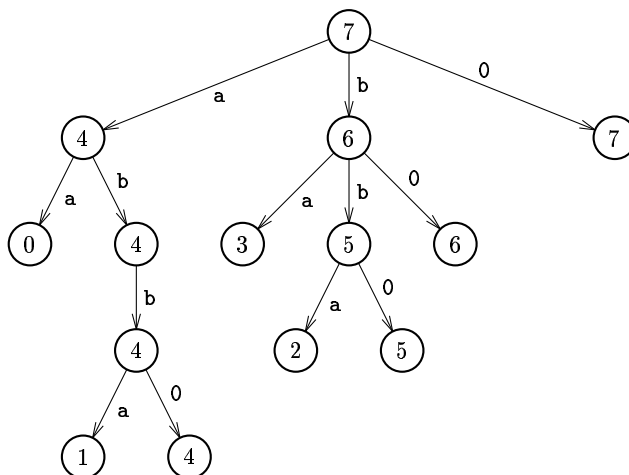


Figure 12: Position tree for  $x^R = aabbabb$ . The nodes contain the  $J$  values. For instance, if  $ab$  is the rightmost factor of the window ( $ba$  read from right to left), a shift of length 3 should be applied.

if the scan does not end on a leaf otherwise the  $J$  value of the leaf is used for the shift. It is only when a suffix (leading to a null length shift) of the pattern is recognized that a test for a full occurrence of the whole pattern is naively performed. Actually, the length of the recognized suffix is easily known and the naive check is only performed for the corresponding prefix of the pattern. To avoid checking at each attempt if the window has gone beyond the right end of the text,  $m$  occurrences of the special null character 0 are concatenated at the end of the text and  $J(\delta(q_0, 0))$  is set to 0. It is only when an occurrence of the pattern is found that the test is performed [20]. When an occurrence of  $x$  has been found a shift of length  $shift = \min\{i \mid 0 \leq i \leq m - 1 \text{ and } x[m - 1 - i] = x[m - 1]\} \cup \{m\}$  is performed. The String Matching with a Position Tree algorithm can now be written as in figure 13.

The worst case time and space complexity of the preprocessing of this algorithm is unfortunately quadratic ( $O(m^2)$ ) and the worst case time complexity of the searching phase is quadratic ( $O(m \times n)$ ) but it has a good behavior in practice.

## The experimental results

We counted the number of text character inspections and we measured the running times for the seven algorithms presented above: the Tuned Boyer-Moore algorithm (TBM for short), the Quick Search algorithm (QS), the Berry-Ravindran algorithm (BR), the Alpha Skip Search algorithm (AS), the Reverse

```

STRING-MATCHING-POSITION-TREE( $x, m, y, n$ )
1   $j \leftarrow m - 1$ 
2   $s \leftarrow 0$ 
3  while  $j < n$  do
4      repeat  $j \leftarrow j + s$ 
5           $i \leftarrow j$ 
6           $s \leftarrow m$ 
7           $q \leftarrow q_0$ 
8          while  $\delta(q, y[i])$  is defined do
9               $q \leftarrow \delta(q, y[i])$ 
10             if  $\delta(q, 0)$  is defined then
11                  $s \leftarrow J(\delta(q, 0))$ 
12              $i \leftarrow i - 1$ 
13             if  $q \in T$  then
14                  $s \leftarrow J(q)$ 
15         until  $s = 0$ 
16         if  $x = y[j - m + 1 .. j]$  and  $j < n$  then
17             REPORT( $j - m + 1$ )
18          $s \leftarrow shift$ 

```

Figure 13: The String Matching with a Position Tree algorithm.

Factor algorithm (RF), the Backward Oracle Matching algorithm (BOM) and the String Matching with a Position Tree algorithm (PT). The experiments were conducted on different kinds of text: binary alphabet, four-letter alphabet, eight-letter alphabet, twenty-letter alphabet, ninety-letter alphabet, a DNA sequence, C code and natural language (English). The five first texts were randomly built. Each time we searched for a hundred patterns. The patterns were randomly built when the texts were built so and the patterns were randomly chosen in the texts otherwise.

## The texts

### Binary alphabet

The text is composed of 500,000 characters and was randomly built. The distribution is uniform with 49.97 per cent of one letter and 50.03 per cent of the second one.

### Four-letter alphabet

The text is composed of 500,000 characters and was randomly built. The distribution is uniform with 24.97, 24.99, 25.01 and 25.03 per cent of each character respectively.

### **Eight-letter alphabet**

The text is composed of 500,000 characters and was randomly built. The distribution is uniform from 12.48 per cent of the least frequent character to 12.52 per cent of the most frequent one.

### **Twenty-letter alphabet**

The text is composed of 500,000 characters and was randomly built. The distribution is uniform from 4.95 per cent of the least frequent character to 5.06 per cent of the most frequent one.

### **Ninety-letter alphabet**

The text is composed of 500,000 characters and was randomly built. The distribution is uniform from 1.0864 per cent of the least frequent character to 1.148 per cent of the most frequent one.

### **DNA sequence**

A DNA sequence is composed of four nucleotides: Adenine, Cytosine, Guanine and Thymine. The sequence we used is composed of 180,136 base pairs of the region of the replication origin of the *Bacillus subtilis* chromosome. It is composed of 29.63 per cent of Adenine, 20.56 per cent of Cytosine, 22.95 per cent of Guanine and 26.86 per cent of Thymine.

### **C code**

We used a program written in C. The alphabet is composed of 93 different characters. The text is composed of 140,240 characters. The distribution can be found in [21].

### **Natural language**

We used Lewis Carroll's novel *Alice's Adventures in Wonderland*. The alphabet is composed of 70 different characters. The text is composed of 148,188 characters. The distribution can be found in [21].

### **Text character inspections**

For each run of the algorithms we counted the average number  $c$  of inspections for one text character. It means that we counted each time a text character is accessed, either to perform a comparison with a pattern character or to perform a shift or to compute a transition in an automaton.

Table 1: Number of text character inspections for a binary alphabet.

$\sigma = 2$	8	9	10	20	40	80	160	320	640	1280
TBM	1.285	1.266	1.236	1.295	1.253	1.279	1.249	1.270	1.311	1.314
QS	1.571	1.614	1.638	1.669	1.642	1.623	1.633	1.626	1.656	1.642
BR	1.555	1.566	1.665	1.556	1.547	1.562	1.494	1.514	1.551	1.542
AS	.917	.869	.830	.457	.246	.132	.072	.040	.025	.010
RF	<b>.621</b>	<b>.556</b>	<b>.512</b>	<b>.294</b>	<b>.169</b>	<b>.097</b>	<b>.056</b>	<b>.033</b>	<b>.023</b>	<b>.009</b>
BOM	.701	.631	.595	.348	.202	.116	.067	.040	.026	.010
PT	1.065	.941	.883	.477	.258	.144	.081	.047	.028	.013

Table 2: Number of text character inspections for a four-letter alphabet.

$\sigma = 4$	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	.492	.458	.451	.440	.409	.420	.396	.424	.440	.412	.439	.413
QS	.727	.678	.673	.646	.635	.627	.623	.641	.655	.637	.663	.626
BR	.684	.618	.575	.532	.510	.346	.289	.268	.276	.270	.280	.264
AS	.555	.523	.499	.481	.466	.197	.139	.060	.040	.019	.014	.005
RF	<b>.392</b>	<b>.344</b>	<b>.315</b>	<b>.285</b>	<b>.265</b>	<b>.154</b>	<b>.089</b>	<b>.051</b>	<b>.029</b>	<b>.017</b>	<b>.012</b>	<b>.004</b>
BOM	.400	.352	.323	.293	.274	.161	.096	.056	.032	.019	.014	.005
PT	.548	.474	.440	.394	.361	.208	.121	.068	.038	.022	.014	.006

### Binary alphabet

The results are shown table 1 and figure 14. In the case of a small alphabet RF algorithm is always performing the smallest number of text character inspections even scanning less than one text character out of ten when searching for long patterns. It is easy to see that the BOM algorithm is scanning a little bit more characters than the RF algorithm. It is due to the fact that the suffix oracle recognizes more words than the suffix automaton. It is a well-known fact that the bad-character shift is not very efficient for small alphabet, thus it is not surprising that the three algorithms TBM, QS and BR are always outperformed by the four other algorithms. Among those Boyer-Moore type algorithms, TBM algorithm is better than the other two (QS and BR).

### Four-letter alphabet

The results are shown table 2 and figure 15. As for a binary alphabet, RF algorithm is always performing better than the other algorithms. The difference is less important between the Boyer-Moore type algorithms (TBM, QS and BR) and the others (AS, RF, BOM and PT), it is especially true for short patterns ( $m < 20$ ). TBM is better than QS and BR for those short patterns, but BR becomes the most efficient of the three for long patterns.

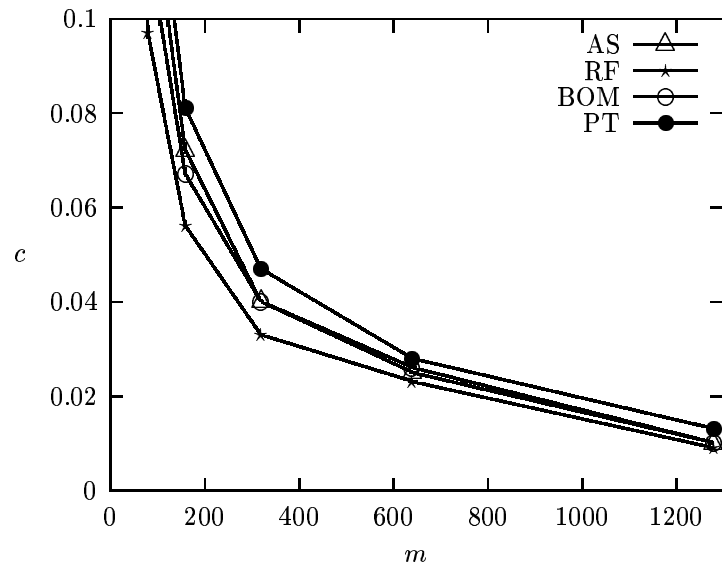


Figure 14: Number of text character inspections for a binary alphabet.

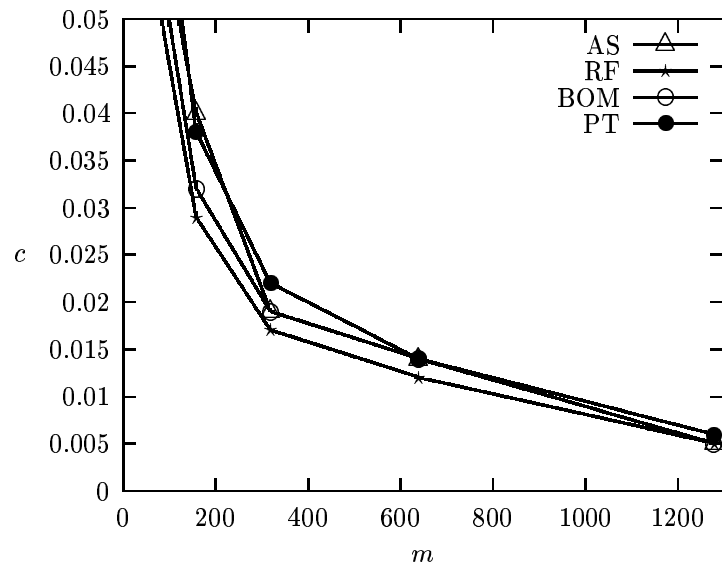


Figure 15: Number of text character inspections for a four-letter alphabet.

Table 3: Number of text character inspections for an eight-letter alphabet.

$\sigma = 8$	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	.292	.264	.244	.230	.219	.179	.162	.165	.167	.170	.165	.165
QS	.465	.427	.401	.381	.361	.305	.277	.283	.279	.284	.278	.286
BR	.485	.434	.393	.360	.332	.194	.116	.076	.059	.056	.056	.055
AS	.333	.306	.285	.269	.257	.199	.171	.043	.030	.024	.008	.004
RF	<b>.283</b>	<b>.251</b>	<b>.227</b>	<b>.209</b>	<b>.192</b>	<b>.113</b>	<b>.063</b>	<b>.036</b>	<b>.020</b>	<b>.012</b>	<b>.007</b>	<b>.003</b>
BOM	<b>.283</b>	.252	.228	<b>.209</b>	.193	.114	.064	.037	.021	.012	.008	<b>.003</b>
PT	.360	.322	.293	.272	.250	.146	.078	.045	.025	.014	.009	.004

Table 4: Number of text character inspections for a twenty-letter alphabet.

$\sigma = 20$	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.208</b>	<b>.183</b>	<b>.164</b>	<b>.149</b>	<b>.137</b>	<b>.085</b>	.063	.057	.056	.056	.057	.056
QS	.348	.311	.283	.260	.242	.158	.118	.106	.105	.105	.105	.104
BR	.410	.365	.329	.299	.275	.151	.081	.043	.024	.014	.010	.008
AS	.228	.203	.184	.169	.157	.105	.078	.065	.059	.055	.005	.004
RF	.215	.190	.170	.155	.143	<b>.085</b>	<b>.049</b>	<b>.027</b>	<b>.014</b>	<b>.008</b>	<b>.004</b>	<b>.002</b>
BOM	.215	.190	.170	.155	.143	<b>.085</b>	<b>.049</b>	<b>.027</b>	<b>.014</b>	<b>.008</b>	<b>.004</b>	<b>.002</b>
PT	.245	.219	.199	.183	.170	.107	.059	.030	.016	.009	.005	<b>.002</b>

### Eight-letter alphabet

The results are shown table 3 and figure 16. In this case, RF algorithm is still the best but BOM algorithm is coming closer since for large alphabet the two algorithms recognize the same factors. Among the Boyer-Moore type algorithms, BR algorithm takes the lead on TBM algorithm for pattern length around 40.

### Twenty-letter alphabet

The results are shown table 4 and figure 17. For large enough alphabet, the bad-character shift is very useful. In this case, TBM algorithm is the best for short patterns, up to length 20, then both RF and BOM algorithms becomes the best. Among the Boyer-Moore type algorithms, BR algorithm takes the lead on TBM algorithm for pattern length around 80.

### Ninety-letter alphabet

The results are shown table 5 and figure 18. In this case, TBM algorithm is the best for patterns up to length 80, then RF, BOM and PT algorithms becomes the best. Among the Boyer-Moore type algorithms, BR algorithm takes the

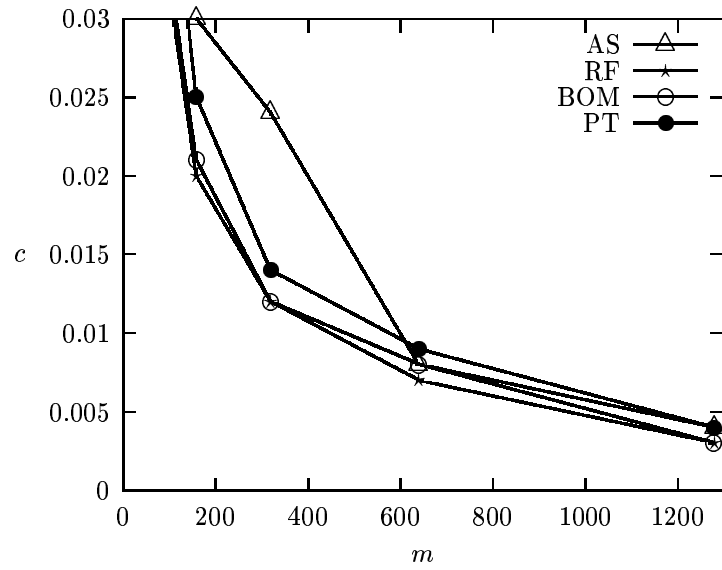


Figure 16: Number of text character inspections for an eight-letter alphabet.

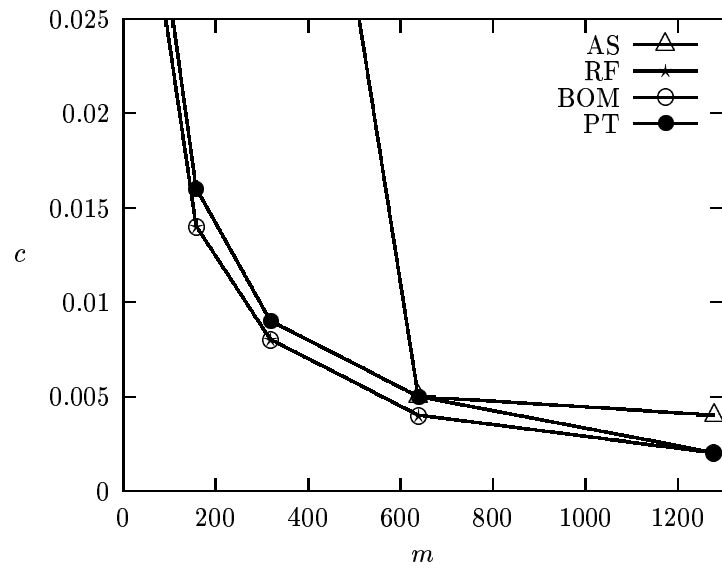


Figure 17: Number of text character inspections for a twenty-letter alphabet.

Table 5: Number of text character inspections for a ninety-letter alphabet.

$\sigma = 90$	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.175</b>	<b>.151</b>	<b>.132</b>	<b>.118</b>	<b>.107</b>	<b>.056</b>	<b>.031</b>	<b>.019</b>	.013	.011	.010	<b>.001</b>
QS	.298	.262	.234	.212	.194	.107	.061	.037	.026	.022	.021	.002
BR	<b>.382</b>	<b>.339</b>	<b>.305</b>	<b>.277</b>	<b>.254</b>	<b>.139</b>	<b>.072</b>	<b>.037</b>	<b>.019</b>	<b>.009</b>	<b>.004</b>	<b>.002</b>
AS	.179	.155	.137	.123	.112	.061	.036	.023	.017	.014	.012	.003
RF	.178	.153	.135	.122	.111	.060	.034	<b>.019</b>	<b>.011</b>	<b>.006</b>	<b>.003</b>	<b>.001</b>
BOM	.178	.153	.135	.122	.111	.060	.034	<b>.019</b>	<b>.011</b>	<b>.006</b>	<b>.003</b>	<b>.001</b>
PT	.184	.159	.142	.128	.117	.066	.040	.024	.013	<b>.006</b>	<b>.003</b>	<b>.001</b>

Table 6: Number of text character inspections for a DNA sequence.

DNA	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	.500	.463	.448	.433	.424	.413	.407	.411	.402	.430	.411	.416
QS	.714	.677	.657	.644	.627	.656	.622	.613	.620	.625	.616	.611
BR	.697	.630	.585	.542	.504	.363	.283	.272	.266	.258	.261	.257
AS	.597	.535	.512	.492	.476	.408	.373	.357	.348	.344	.345	.347
RF	<b>.390</b>	<b>.347</b>	<b>.312</b>	<b>.286</b>	<b>.264</b>	<b>.156</b>	<b>.090</b>	<b>.051</b>	<b>.030</b>	<b>.018</b>	<b>.014</b>	.015
BOM	.398	.356	.320	.294	.272	.163	.096	.056	.032	.020	.015	.016
PT	.541	.476	.427	.393	.362	.215	.122	.069	.038	.022	.015	<b>.013</b>

lead on TBM algorithm for pattern length around 320.

### DNA sequence

The results are shown table 6 and figure 19. Despite the small difference in the text character distribution, the results are very similar with a random text on a four-letter alphabet. RF algorithm is always performing better than the other algorithms, except for very long patterns where PT algorithm is slightly better. TBM is better than QS and BR for short patterns, but BR becomes the most efficient of the three for long patterns.

### C code

The results are shown table 7 and figure 20. Boyer-Moore type algorithms are the best for pattern length up to 40, then for longer patterns, RF, BOM and PT are the most efficient.

### Natural language

The results are shown table 8 and figure 21. Boyer-Moore type algorithms are the best for pattern length up to 9, then for longer patterns, RF, BOM and PT

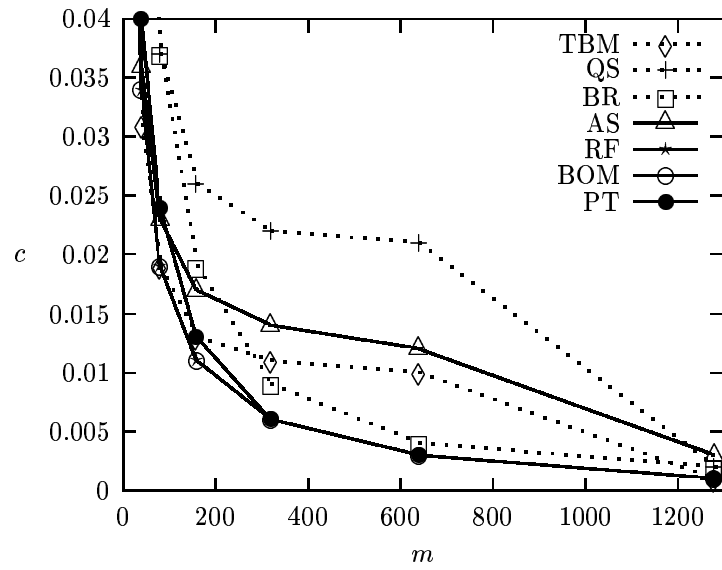


Figure 18: Number of text character inspections for a ninety-letter alphabet.

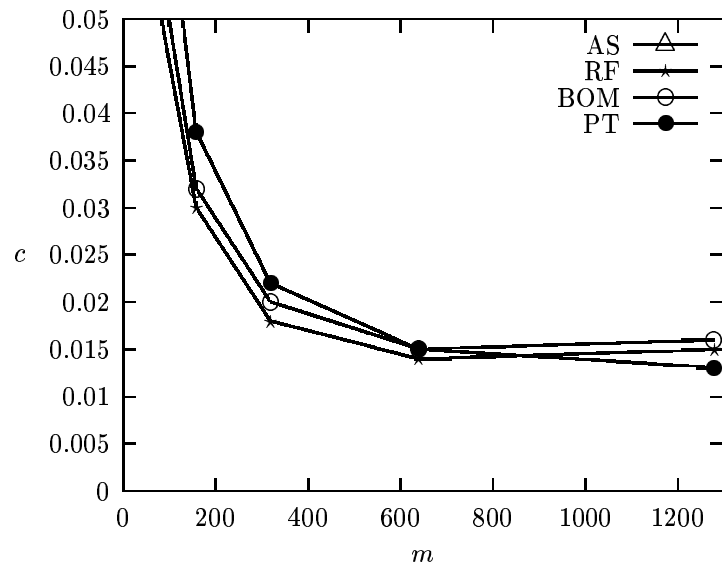


Figure 19: Number of text character inspections for a DNA sequence.

Table 7: Number of text character inspections for a C program.

C	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.277</b>	<b>.257</b>	<b>.245</b>	<b>.240</b>	<b>.232</b>	<b>.215</b>	.208	.045	.040	.026	.020	.028
QS	.424	.392	.374	.355	.340	.294	.257	.081	.061	.042	.033	.038
BR	.516	.471	.452	.419	.401	.319	.267	.072	.046	.025	.018	.019
AS	.289	.274	.265	.259	.256	.276	.316	.248	.179	.131	.102	.102
RF	.284	.268	.254	.246	.241	.236	<b>.203</b>	<b>.035</b>	<b>.022</b>	<b>.016</b>	<b>.015</b>	.020
BOM	.284	.268	.254	.246	.241	.236	<b>.203</b>	.036	<b>.022</b>	.016	.016	.020
PT	.329	.312	.295	.295	.282	.268	.254	.142	.051	.023	.017	<b>.017</b>

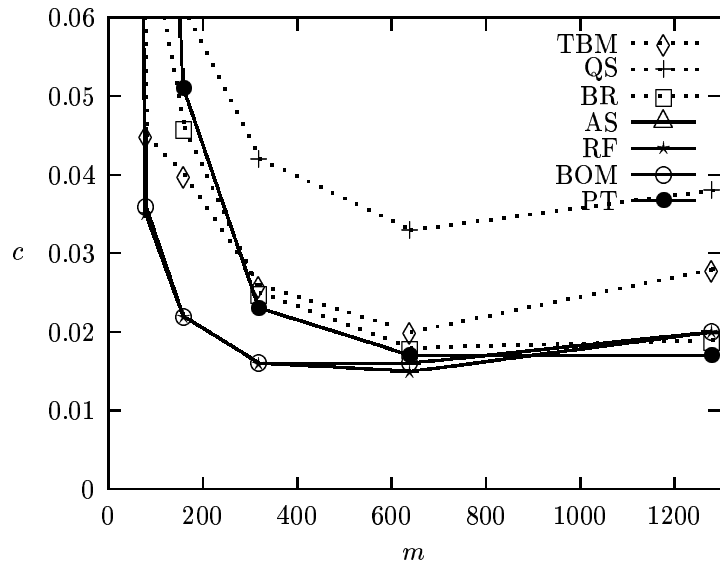


Figure 20: Number of text character inspections for a C program.

Table 8: Number of text character inspections for an English text.

English	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.236</b>	<b>.208</b>	<b>.188</b>	<b>.171</b>	.162	.098	.068	.047	.039	.032	.029	.029
QS	.384	.345	.316	.291	.273	.172	.120	.086	.068	.057	.049	.044
BR	.443	.391	.358	.326	.304	.169	.094	.053	.032	.021	.017	.017
AS	.267	.241	.220	.205	.194	.138	.108	.095	.092	.092	.087	.086
RF	.238	.210	.189	<b>.171</b>	<b>.157</b>	<b>.090</b>	<b>.052</b>	<b>.031</b>	<b>.018</b>	<b>.013</b>	.012	.016
BOM	.238	.210	.189	<b>.171</b>	.158	<b>.090</b>	.053	<b>.031</b>	.019	<b>.013</b>	.012	.016
PT	.275	.244	.222	.201	.184	.103	.062	.035	.021	<b>.013</b>	<b>.010</b>	<b>.012</b>

are the most efficient.

### Summary

The results presented here are a little different with those of [21] due to the fact that in [21], text character inspections performed to compute the shift in QS algorithm were not counted. However, results in [21] already shown that for long patterns RF type algorithms are the best ones, this is also confirmed in [10]. Note that there exists a linear version of RF algorithm named Turbo Reverse Factor algorithm [13] which performs even less text character inspections but is slower in terms of running times [21].

### Running times

The measure of text character inspections is an objective parameter of the performance of string-matching algorithms. But the saving of text character inspections can be done at a very high price. Running times constitute another mean of comparing the algorithms. In order to evaluate the practical performances of the seven string-matching algorithms, we implemented them in C in a homogeneous way to keep the comparison significant. For each of them we counted the running time of both the preprocessing and the searching phases. The times are expressed in hundredth of seconds. The target machine is a PC, with a AMD-K6 II processor at 330Mhz running Linux kernel 2.2. The compiler is gcc.

### Binary alphabet

The results are shown table 9 and figure 22. TBM algorithm is fast for short patterns, up to length around 9. RF algorithm is then the fastest for pattern length around 600 then BOM algorithm becomes the best. This is due to the fact that for long patterns the construction and use of the suffix automaton is very expensive to the contrary of the oracle which needs less resources.

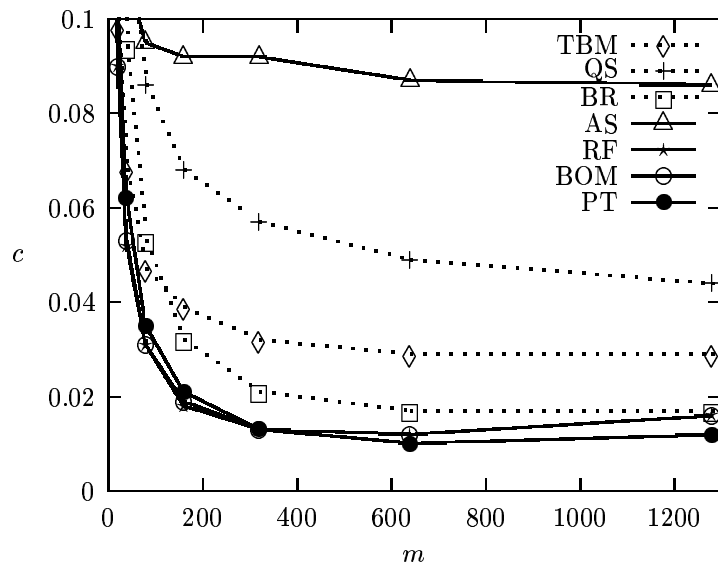


Figure 21: Number of text character inspections for an English text.

Table 9: Running times for a binary alphabet.

$\sigma = 2$	8	9	10	20	40	80	160	320	640	1280
TBM	<b>3.68</b>	3.65	3.58	3.86	3.96	4.06	3.49	3.55	3.70	4.07
QS	5.80	5.96	6.02	6.35	6.58	6.47	5.73	5.88	6.03	6.06
BR	5.47	5.60	5.93	5.77	5.92	6.12	5.10	5.34	5.56	5.68
AS	5.84	5.49	5.19	3.91	2.45	2.97	2.44	4.24	8.11	17.23
RF	3.92	<b>3.60</b>	<b>3.36</b>	<b>1.82</b>	<b>1.41</b>	<b>1.01</b>	<b>.82</b>	<b>1.02</b>	1.88	3.14
BOM	37.30	33.72	31.81	19.15	11.80	7.19	3.78	2.27	<b>1.60</b>	<b>.84</b>
PT	5.01	4.48	4.19	2.32	1.37	1.13	.91	1.38	2.89	6.52

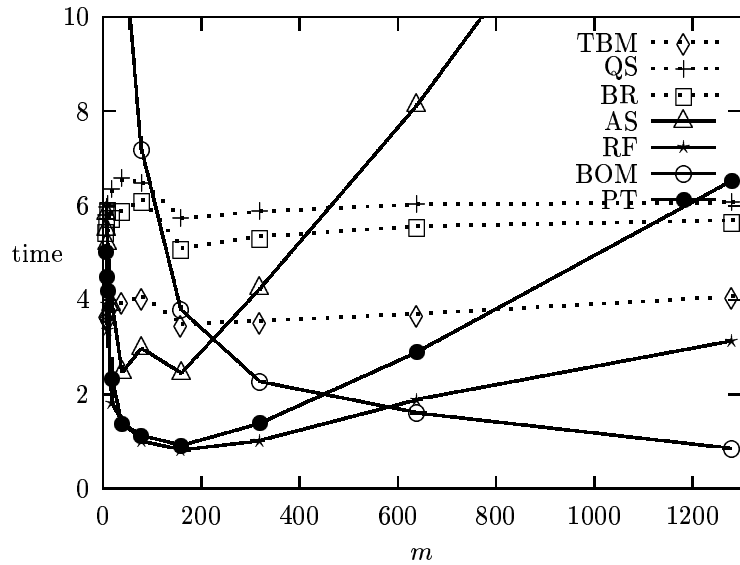


Figure 22: Running times for a binary alphabet.

#### Four-letter alphabet

The results are shown table 10 and figure 23. TBM algorithm is fast for short patterns, up to length around 15. RF algorithm is then the fastest for pattern length between approximately 15 to 30. Then PT is the best for pattern length between 30 to 60, then RF becomes again the best up to pattern length around 550, and for longer patterns BOM algorithm takes the lead. For long patterns, TBM and BR algorithms are also efficient.

Table 10: Running times for a four-letter alphabet.

$\sigma = 4$	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>1.26</b>	<b>1.24</b>	<b>1.24</b>	<b>1.19</b>	<b>1.07</b>	1.11	1.06	1.09	1.16	1.10	1.25	1.24
QS	3.40	3.13	3.14	3.03	3.00	3.00	3.16	3.27	2.82	2.89	3.04	2.87
BR	3.08	2.91	2.83	2.75	2.48	1.82	1.66	1.56	1.60	1.51	1.60	1.57
AS	4.21	3.95	3.71	3.50	3.28	1.90	1.29	1.13	1.27	2.33	4.47	10.87
RF	2.44	2.09	1.90	1.70	1.67	<b>1.07</b>	.84	<b>.55</b>	<b>.50</b>	<b>.56</b>	1.44	2.08
BOM	22.21	19.64	18.12	16.61	15.56	9.68	6.39	4.07	1.85	1.20	<b>.99</b>	<b>.66</b>
PT	2.92	2.47	2.31	2.02	1.83	1.16	<b>.73</b>	.66	.67	.74	1.97	4.60

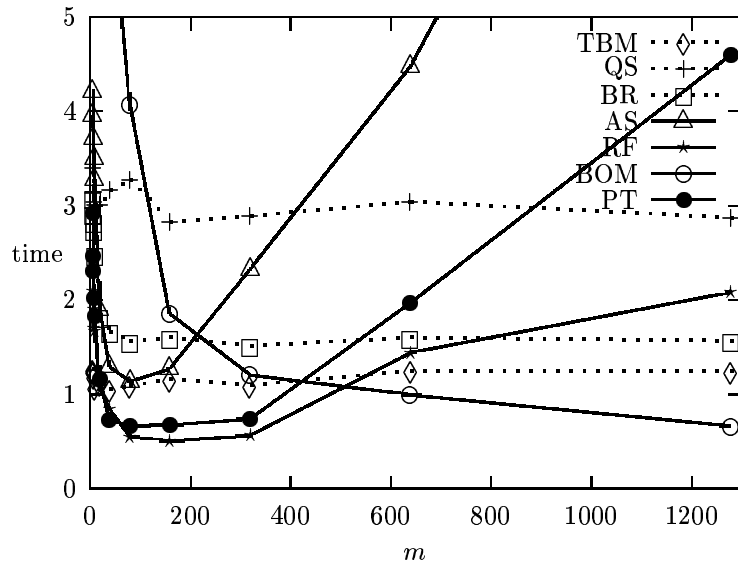


Figure 23: Running times for a four-letter alphabet.

### Eight-letter alphabet

The results are shown table 11 and figure 24. TBM algorithm is the fastest for pattern length up to 60, then RF algorithm becomes the fastest for pattern length between 60 and 500 and for longer patterns TBM is again the fastest. For long patterns BR and BOM algorithms are also efficient.

### Twenty-letter alphabet

The results are shown table 12 and figure 25. TBM algorithm is the fastest for pattern length up to 120, then RF algorithm becomes the fastest for pattern

Table 11: Running times for an eight-letter alphabet.

$\sigma = 8$	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.80</b>	<b>.75</b>	<b>.72</b>	<b>.70</b>	<b>.63</b>	<b>.56</b>	<b>.54</b>	.54	.52	.60	<b>.53</b>	<b>.60</b>
QS	2.16	2.02	1.88	1.76	1.72	1.48	1.39	1.44	1.37	1.39	1.39	1.53
BR	2.44	2.18	1.71	1.92	1.75	1.32	1.06	.90	.82	.81	.82	.83
AS	3.34	2.86	2.73	2.41	2.16	1.41	1.13	.82	.87	1.69	3.12	6.30
RF	1.75	1.58	1.48	1.37	1.29	.85	<b>.66</b>	<b>.40</b>	<b>.32</b>	<b>.44</b>	.68	1.73
BOM	16.48	14.89	13.58	12.60	11.74	7.36	4.69	2.64	1.33	.89	.73	.61
PT	1.88	1.70	1.60	1.48	1.34	.87	.67	.53	.46	.62	1.35	3.71

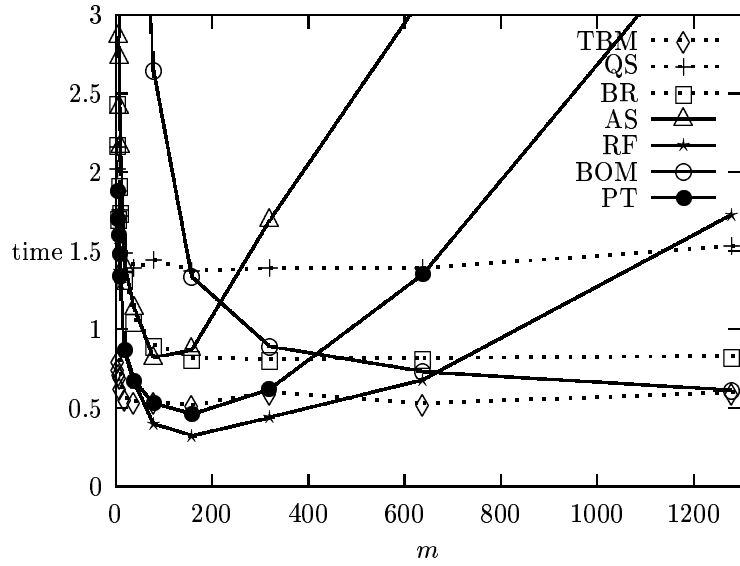


Figure 24: Running times for an eight-letter alphabet.

length between 120 and 220 and for longer patterns TBM algorithm is again the fastest. For long patterns BR, QS and BOM algorithms are also efficient.

Table 12: Running times for a twenty-letter alphabet.

$\sigma = 20$	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.60</b>	<b>.56</b>	<b>.52</b>	<b>.48</b>	<b>.46</b>	<b>.37</b>	<b>.33</b>	<b>.23</b>	.31	<b>.26</b>	<b>.32</b>	<b>.34</b>
QS	1.69	1.49	1.40	1.36	1.21	.87	.72	.66	.64	.60	.67	.63
BR	2.19	1.91	1.82	1.70	1.59	1.26	1.01	.76	.62	.56	.57	.57
AS	2.75	2.27	2.00	1.80	1.69	1.10	.88	.77	.80	.98	2.40	4.52
RF	1.41	1.28	1.17	1.12	1.06	.73	.56	.32	<b>.22</b>	.41	.55	1.35
BOM	13.47	12.15	11.05	10.22	9.56	6.12	4.02	1.91	1.06	.73	.59	.71
PT	1.36	1.27	1.11	1.07	1.07	.75	.59	.44	.46	.45	1.21	4.34

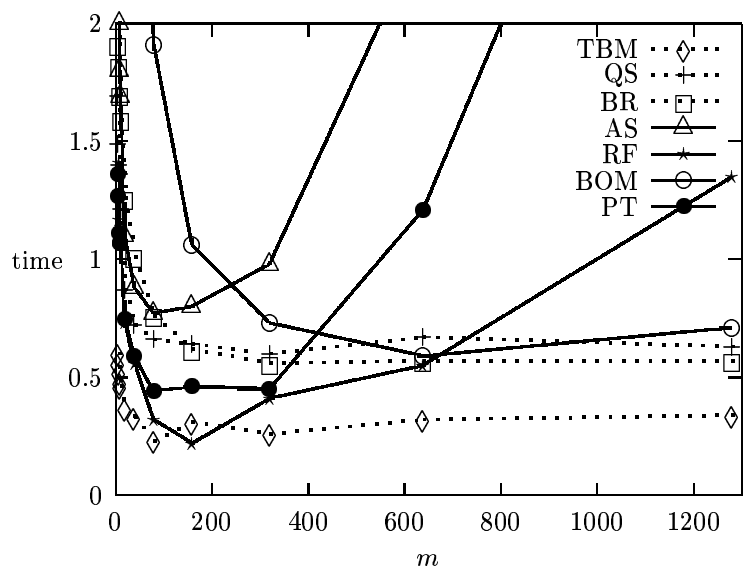


Figure 25: Running times for a twenty-letter alphabet.

Table 13: Running times for a ninety-letter alphabet.

$\sigma = 90$	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.56</b>	<b>.46</b>	<b>.46</b>	<b>.43</b>	<b>.44</b>	<b>.34</b>	<b>.25</b>	<b>.19</b>	<b>.10</b>	<b>.07</b>	<b>.08</b>	<b>.06</b>
QS	1.49	1.34	1.23	1.13	1.02	.68	.46	.29	.22	.23	.24	.07
BR	2.90	2.51	2.37	2.05	2.02	1.39	1.03	.80	.63	.53	.52	.50
AS	2.31	1.92	1.72	1.59	1.49	.92	.71	.50	.67	.81	1.74	2.61
RF	1.17	1.06	.99	.90	.91	.64	.48	.30	.32	.30	.46	.64
BOM	11.88	10.57	9.62	8.93	8.34	5.47	3.91	2.21	1.28	.86	.71	1.00
PT	1.01	.93	.84	.77	.76	.57	.49	.40	.45	.44	1.04	3.71

Table 14: Running times for a DNA sequence.

DNA	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.51</b>	<b>.48</b>	<b>.49</b>	<b>.49</b>	<b>.42</b>	.42	.44	.44	.44	<b>.41</b>	<b>.41</b>	<b>.45</b>
QS	1.02	1.00	.95	.96	.95	.99	.94	.92	.94	.91	.91	.94
BR	1.23	1.24	1.13	1.09	1.08	.89	.86	.80	.79	.82	.83	.76
AS	1.37	1.26	1.15	1.13	1.02	.83	.79	.69	.94	1.03	1.86	2.81
RF	.84	.73	.71	.69	.60	<b>.41</b>	.38	<b>.23</b>	<b>.34</b>	.57	.99	2.05
BOM	8.35	7.53	6.89	6.35	5.99	3.97	2.12	1.19	.74	.48	.50	.71
PT	.99	.88	.84	.72	.71	.46	<b>.31</b>	.38	.46	.68	1.60	4.80

### Ninety-letter alphabet

The results are shown table 13 and figure 26. In this case, TBM algorithm is always the best.

### DNA sequence

The results are shown table 14 and figure 27. As in the case of a random text on a four-letter alphabet, TBM algorithm is fast for short patterns, up to length around 15. RF algorithm is then the fastest for pattern length between approximately 15 to 30. Then PT algorithm is the best for pattern length between 30 to 60, then RF algorithm becomes again the best but up to pattern length around 220. Then TBM algorithm becomes the fastest again. This is probably due to the fact that the text here is shorter (180,136 comparing to 500,000 characters). For long patterns, TBM and BR algorithms are also efficient.

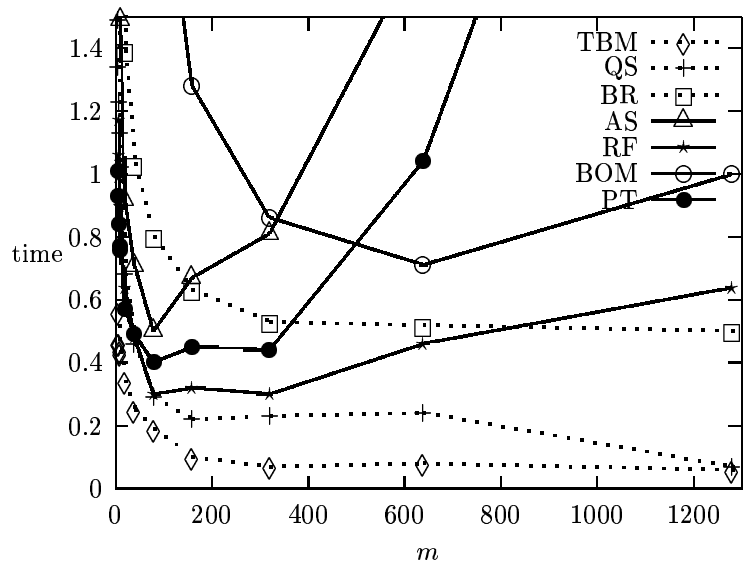


Figure 26: Running times for a ninety-letter alphabet.

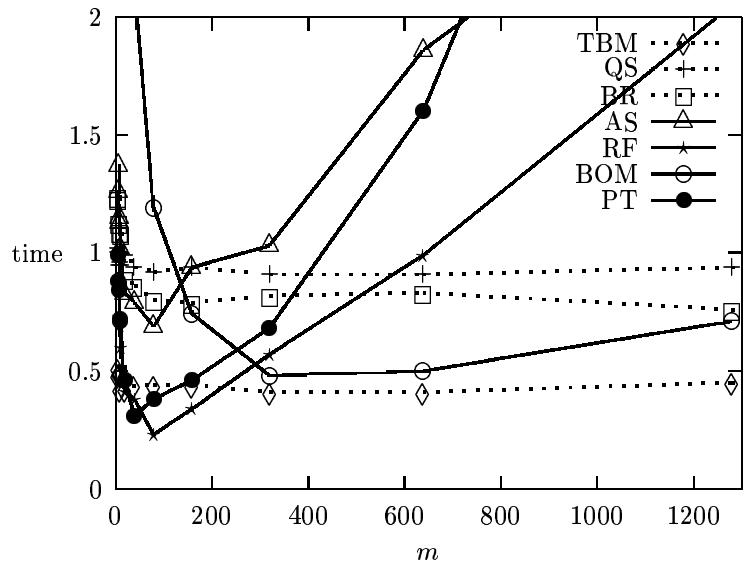


Figure 27: Running times for a DNA sequence.

Table 15: Running times for a C program.

C	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.21</b>	<b>.20</b>	<b>.19</b>	<b>.16</b>	<b>.14</b>	<b>.16</b>	<b>.09</b>	<b>.09</b>	<b>.06</b>	.07	.03	<b>.06</b>
QS	.47	.44	.45	.40	.36	.28	.18	.14	.08	<b>.05</b>	<b>.02</b>	<b>.06</b>
BR	.95	.88	.88	.85	.82	.74	.66	.49	.52	.50	.50	.61
AS	.72	.62	.56	.54	.50	.33	.35	.35	.47	.65	1.66	2.49
RF	.50	.52	.51	.46	.49	.47	.66	.18	.23	.39	.72	1.68
BOM	5.03	4.75	4.45	4.27	4.29	3.84	3.22	.75	.51	.44	.45	.82
PT	.48	.48	.49	.48	.46	.44	.49	.41	.56	1.13	4.37	9.78

Table 16: Running times for an English text.

English	6	7	8	9	10	20	40	80	160	320	640	1280
TBM	<b>.22</b>	<b>.21</b>	<b>.18</b>	<b>.15</b>	<b>.14</b>	<b>.12</b>	<b>.10</b>	<b>.08</b>	<b>.11</b>	<b>.05</b>	<b>.07</b>	.09
QS	.52	.50	.41	.41	.38	.28	.18	.18	.14	.10	.09	<b>.03</b>
BR	.95	.88	.87	.80	.75	.68	.57	.55	.48	.44	.50	.39
AS	.79	.72	.66	.56	.54	.43	.42	.35	.51	.67	1.72	2.55
RF	.46	.44	.39	.42	.39	.25	.33	.17	.24	.42	.71	1.62
BOM	4.61	4.20	3.86	3.59	3.29	1.76	1.12	.68	.47	.44	.50	.75
PT	.48	.46	.40	.38	.37	.22	.26	.25	.36	.66	1.76	8.01

### C code

The results are shown table 15 and figure 28. In this case TBM algorithm is the fastest being joined by QS algorithm for long patterns ( $m > 200$ ).

### Natural language

The results are shown table 16 and figure 29. In this case TBM algorithm is the fastest being joined by QS algorithm for long patterns ( $m > 800$ ).

### Summary

The general tendency is that for short patterns TBM is the best, then RF becomes the best, then for PT, then RF again then TBM or BOM. The values depend both on the size of the alphabet and the length of the text.

In [9] BR was the best but was only tested against QS not against the five other algorithms presented here.

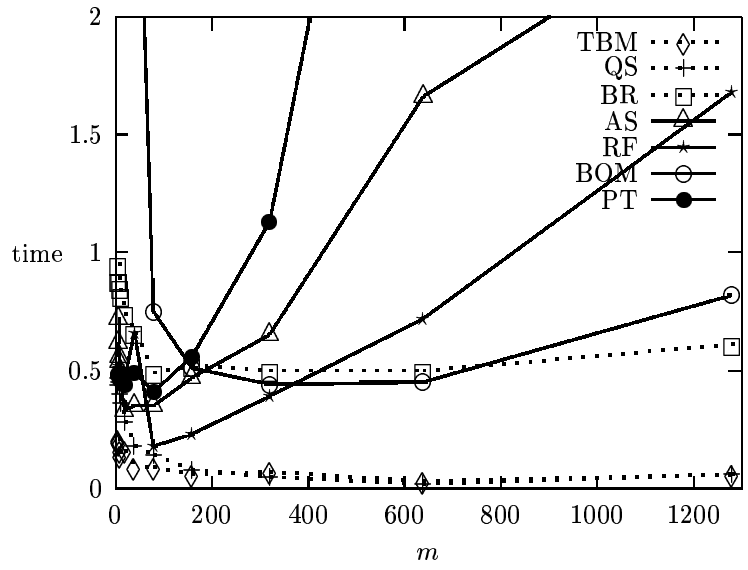


Figure 28: Running times for a C program.

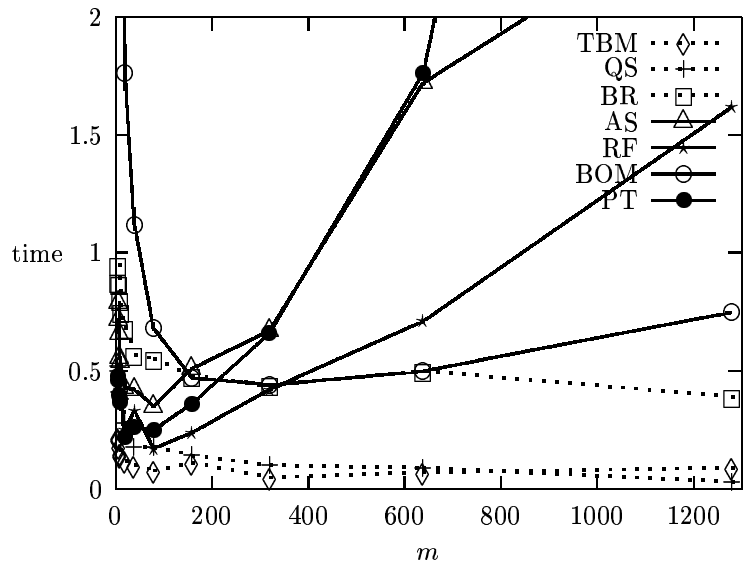


Figure 29: Running times for an English text.

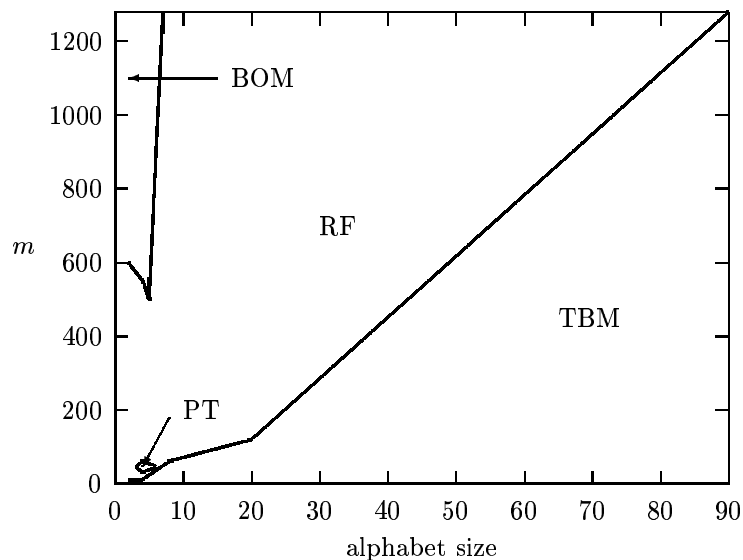


Figure 30: Area of performances of the string-matching algorithms for random texts of length 500,000.

## Conclusion

As expected, the algorithms recognizing factors of the pattern rather than suffixes perform less text character inspections. This is especially true for small alphabet. Those algorithms use data structures which need more resources than Boyer-Moore type algorithms. That is the reason why they are not always the fastest algorithms. For large alphabet, the Tuned Boyer-Moore algorithm is always the algorithm to be chosen. For small alphabet, for short patterns the Tuned Boyer-Moore algorithm is fast, this is typically the case for “search” and “substitute” commands in a text editor for instance. When the pattern length increases, the Reverse Factor algorithms becomes the more efficient up to a point where the String Matching with a Position Tree algorithm becomes the choice. When the pattern length keeps increasing, the Reverse Factor becomes again the fastest up to a point where depending on the condition, the Backward Oracle Matching or the Tuned Boyer-Moore becomes the best. Figure 30 shows the situation for random texts of 500,000 characters.

## References

- [1] C. Charras and T. Lecroq. Exact string matching algorithms. URL: <http://www-igm.univ-mlv.fr/~lecroq/string/>, 1997.

- [2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
- [3] R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
- [4] G. Davies and S. Bowsher. Algorithms for pattern matching. *Softw. Pract. Exp.*, 16(6):575–601, 1986.
- [5] R. A. Baeza-Yates. Improved string searching. *Softw. Pract. Exp.*, 19(3):257–271, 1989.
- [6] P. D. Smith. Experiments with a very fast substring search algorithm. *Softw. Pract. Exp.*, 21(10):1065–1074, 1991.
- [7] A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.
- [8] T. Raita. Tuning the Boyer-Moore-Horspool string searching algorithm. *Softw. Pract. Exp.*, 22(10):879–884, 1992.
- [9] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '99*, pages 16–28, Czech Technical University, Prague, Czech Republic, 1999. Collaborative Report DC-99-05.
- [10] P.D. Michailidis and K.G. Margaritis. String matching algorithms: Survey and experimental results. *Int. J. Comput. Math.*, 2000. to appear.
- [11] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.
- [12] C. Charras, T. Lecroq, and J.D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 55–64, Piscataway, NJ, 1998. Springer-Verlag, Berlin.
- [13] 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):247–267, 1994.
- [14] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99, Theory and Practice of Informatics*, number 1725 in Lecture Notes in Computer Science, pages 291–306, Milovy, Czech Republic, 1999. Springer-Verlag, Berlin.
- [15] C. Charras and T. Lecroq. Fast string matching with position tree. submitted to SODA, 2001.

- [16] 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):31–55, 1985.
- [17] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.
- [18] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.
- [19] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.
- [20] T. Raita. On guards and symbol dependencies in substring search. *Softw. Pract. Exp.*, 29(11):931–941, 1999.
- [21] T. Lecroq. Experimental results on string matching algorithms. *Softw. Pract. Exp.*, 25(7):727–765, 1995.