A fast implementation of the Boyer–Moore string matching algorithm

Maxime Crochemore* Thierry Lecroq†

Abstract

String matching is the problem of finding all the occurrences of a pattern in a text. We present a new method to compute a combinatorial shift function ("best matching shift") of the well-known Boyer–Moore string matching algorithm. Moreover we conduct experiments showing that the algorithm using this best matching shift is the most efficient in particular cases such as the search for patterns of length from 7 to 15 in natural language texts.

1 INTRODUCTION

The string matching problem consists in finding one or more usually all the occurrences of a pattern x of length m in a text y of length n. It can occur in information retrieval, bibliographic search and more recently it has some applications in molecular biology. It has been extensively studied and numerous techniques and algorithms have been designed to solve this problem (see [12, 29, 14, 9, 28, 22, 6, 11]). We are interested here in the problem where the pattern is given first and can then be searched for in various texts. Thus a preprocessing phase is allowed on the pattern.

Basically a string-matching algorithm uses a window to scan the text. The size of this window is equal to the length of the pattern. It first aligns the left ends of the window and of the text. Then it checks if the pattern occurs in the window (this specific work is called an *attempt*) and *shifts* the window to the right. It repeats the same procedure again until the right end of the window goes beyond the right end of the text. One of the most famous string matching algorithm was given in 1977 by Boyer and Moore [4]. Its main feature is that at each attempt it scans the characters of the pattern from right to left which enables it to "jump" over some portions of the text and therefore to save some comparisons. The Boyer-Moore algorithm is among the most efficient exact string matching algorithm. A simplified version of it, or the entire algorithm, is often implemented in text editors for the search and substitute commands.

*King's College London and Université Paris-Est, www.dcs.kcl.ac.uk/staff/mac/

†LITIS (Laboratoire d'Informatique, du Traitement de l'Information et des Systèmes), Université de Rouen, monge.univ-mlv.fr/lecroq

The algorithm scans the characters of the pattern from right to left beginning with the rightmost symbol. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the *occurrence shift* (also called *bad-character shift*) and the *matching shift* (also called *good-suffix shift*).

Assume that during an attempt the algorithm matches a suffix u of x in y and that a mismatch occurs between a character a of the pattern and a character b of the text. Then the matching shift consists in aligning the factor u in y with its rightmost occurrence in $x[0..m-2]$. There exist three different matching shifts depending on the condition imposed to the character c that precedes this occurrence of u in $x[0..m-2]$. If there is no condition on c the shift is called the weak matching shift; if c must be different from a then the shift is called the strong matching shift; and if c must be equal to b then the shift is called the best matching shift. If there exists no such occurrence of u in $x[0..m-2]$, the shift consists in aligning the longest suffix v of u with a matching prefix of x. The first linear computation of the strong matching shift was given by Rytter [25]. In [10], we gave a simpler version of this computation. In [9, 11], we described the linear time and space computation of the "best factor" automaton that implements the best matching shift. Independently and recently, in [20] an efficient (but in quadratic space) computation of the best matching shift was presented. The authors designed the PAMA algorithm that implements the best matching shift in quadratic space and memorizes some information from one attempt to the next as the Turbo-BM algorithm [8] does. In this paper we apply the same techniques as in [10] for computing the strong matching shift to simply compute the best matching shift in quadratic space. We perform experiments showing that in some cases the new algorithm is the fastest among all the tested algorithms.

This paper is organized as follows: Section 2 recalls briefly the Boyer–Moore algorithm; in Section 3 we give a method to compute the matching shift functions of the Boyer–Moore algorithm and in Section 4 we report experimental results. Throughout this paper the pattern is a word x of length m, $x = x[0..m-1]$. The text is a word y of length n, $y = y[0..n-1]$. Both x and y are built over a finite alphabet Σ of size σ.

2 BOYER–MOORE STRING-MATCHING ALGORITHM

The Boyer-Moore algorithm is considered as the most efficient string matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in a text editor for the "search" and "substitute" commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost symbol. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the pattern to the

Figure 1: Typical situation during the Boyer–Moore algorithm: a suffix u of the pattern is found and a mismatch occurs between a character a in the pattern x and a character b in the text y.

right. These two shift functions are called the *matching shift* and the *occurrence shift*.

Assume that a suffix u of x has been matched and a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i + j] = b$ of the text during an attempt where x is aligned with $y[j..j+m-1]$. Then, $x[i+1..m-1] = y[i + j + 1..j + m - 1] = u$ and $a = x[i] \neq y[i + j] = b$ (see Fig. 1).

The matching shift consists in aligning the substring $u = x[i + 1..m - 1] = y[i + j + 1..j + m - 1]$ with one of its reoccurrences in x. Informally, let us distinguish three matching shift cases on the grounds of the restrictions imposed on the character c preceding this reoccurrence:

weak matching shift :
: there is no condition on the character c preceding u, it is then possible that $c = a$ (see Fig. 2).

strong matching shift :
: the character c must be different from the character a (see Fig. 3).

best matching shift :
: the character c must be equal to b (see Fig. 4).

It is not too difficult to see that the following inequality holds:

$$|\text{weak matching shift}| \leq |\text{strong matching shift}| \leq |\text{best matching shift}|$$

where the absolute value of a shift denotes its length.

If there exists no other occurrence of u, the matching shift consists in aligning the longest suffix v of $y[i + j + 1..j + m - 1]$ with a matching prefix of x (see Fig. 5).

The occurrence shift consists in aligning the text character $y[i + j]$ with its rightmost occurrence in $x[0..m - 2]$ (see Fig. 6). If $y[i + j]$ does not appear in the pattern x, no occurrence of x in y can include $y[i + j]$, and the left end of the pattern is aligned with the character immediately after $y[i + j]$, namely $y[i + j + 1]$ (see Fig. 7).

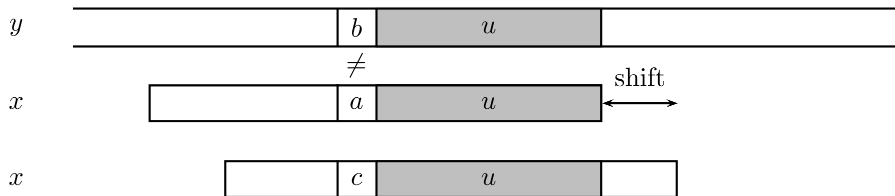

Figure 2: Weak matching shift: c can be equal to a.

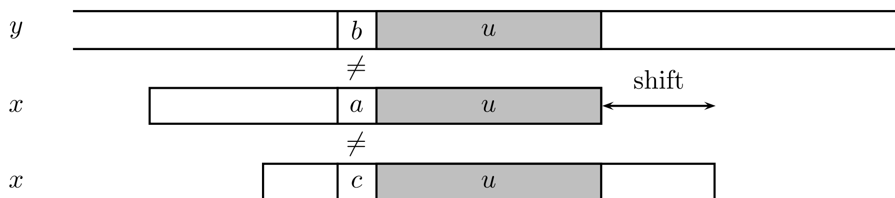

Figure 3: Strong matching shift: $c \neq a$.

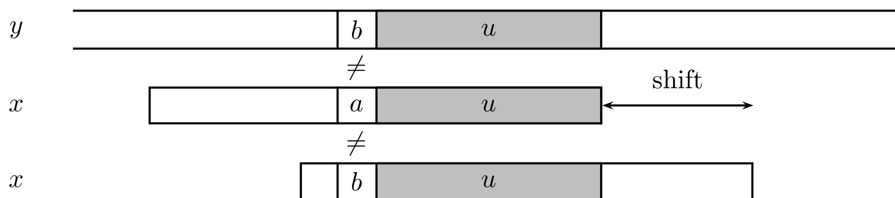

Figure 4: Best matching shift.

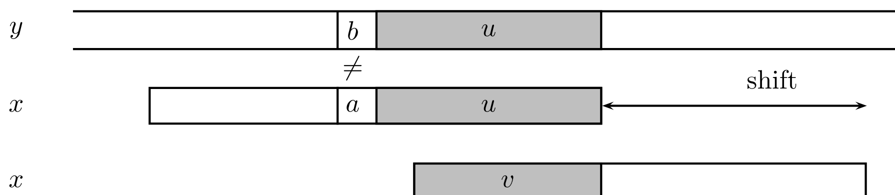

Figure 5: Matching shift, only a prefix of u reappears in x.

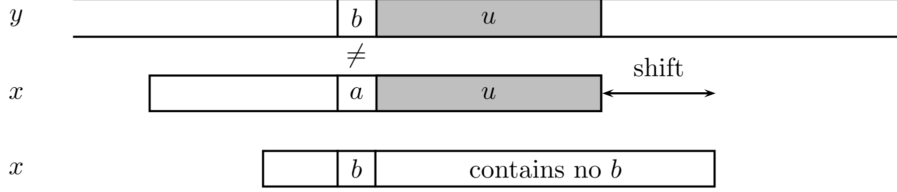

Figure 6: Occurrence shift, b appears in x.

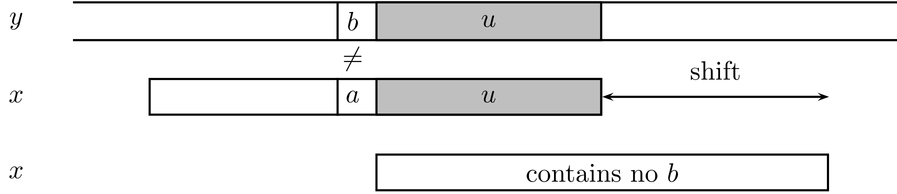

Figure 7: Occurrence shift, b does not appear in x.

The three shift functions will be denoted by *wMatch*, *sMatch*, and *bMatch*. We will define these three variables with the aid of the condition functions *Cs*, *Cos* and *Cob*:

For $0 \leq i \leq m-1$, $1 \leq s \leq m$, and $a \in \Sigma$, let us define the following conditions.

- The condition of suffix *Cs* is defined for a position i and a shift s:

$$Cs(i,s) = \begin{cases} 0 < s \leq i \text{ and } x[i-s+1\mathinner{.\,.}m-s-1] \text{ is a suffix of } x \\ \text{or} \\ s > i \text{ and } x[0\mathinner{.\,.}m-s-1] \text{ is a suffix of } x \end{cases}$$

- The strong condition of occurrence *Cos* is defined for a position i and a shift s:

$$Cos(i,s) = \begin{cases} 0 \leq s \leq i \text{ and } x[i-s] \neq x[i] \\ \text{or} \\ s > i \end{cases}$$

- The best condition of occurrence *Cob* is defined for a position i, a character a and a shift s:

$$Cob(i,a,s) = \begin{cases} 0 \leq s \leq i \text{ and } x[i-s] = a \\ \text{or} \\ s > i \end{cases}$$

Then, for $0 \leq i \leq m-1$ and $a \in \Sigma$:

```
BOYER-MOORE(x, m, y, n)
1  j ← 0
2  while j ≤ n − m
3      do i ← m − 1
4         while i ≥ 0 and x[i] = y[i + j]
5             do i ← i − 1
6         if i < 0
7           then REPORT(j)
8                j ← j + MATCH(0, y[i + j])
9           else j ← j + max(MATCH(i, y[i + j]), occ[y[i + j]] − m + i + 1)
```

Figure 8: The Boyer–Moore string matching algorithm.

- the weak matching shift is defined by:

$$wMatch[i] = \min\{s > 0 \mid Cs(i, s) \text{ holds}\}$$

- the strong matching shift is defined by:

$$sMatch[i] = \min\{s > 0 \mid Cs(i, s) \text{ and } Cos(i, s) \text{ hold}\}$$

- the best matching shift is defined by:

$$bMatch[i, a] = \min\{s > 0 \mid Cs(i, s) \text{ and } Cob(i, a, s) \text{ hold}\}$$

Remark: $wMatch[0] = sMatch[0] = bMatch[0, y[j]]$ is equal to the period of x for all $0 \leq j \leq n - 1$.

The occurrence shift is defined as follows. For $a \in \Sigma$:

$$occ[a] = \begin{cases} \min\{i \mid 1 \leq i \leq m - 1 \text{ and } x[m - 1 - i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

The Boyer-Moore algorithm is shown in Fig. 8. The function MATCH(i, a) can return either $wMatch[i]$, $sMatch[i]$ or $bMatch[i, a]$. In the three cases the algorithm will locate all the occurrences of x in y. When shifting the pattern, it applies the maximum between the occurrence shift and the matching shift.

Remark: the occurrence is useless when the best matching shift is used.

3 COMPUTING THE BEST MATCHING SHIFT

For $0 \leq i \leq m-1$ we denote by $suf[i]$ the length of the longest suffix of x ending at position i in x. Let us denote by $lcsuf(u, v)$ the longest common suffix of two words u and v. The algorithm SUFFIXES in Fig. 9 computes the table suf in linear time and space.

We are now able to give, in Fig. 10, the algorithm STRONG-MATCHING which computes in linear time and space the table $sMatch$ using the table suf.

```
SUFFIXES(x, m)
 1  suf[m − 1] ← m
 2  g ← m − 1
 3  for i ← m − 2 downto 0
 4       do if i > g and suf[i + m − 1 − f] ≠ i − g
 5             then suf[i] ← min{suf[i + m − 1 − f], i − g}
 6             else  g ← min{g, i}
 7                   f ← i
 8                   while g ≥ 0 and x[g] = x[g + m − 1 − f]
 9                        do g ← g − 1
10                   suf[i] ← f − g
11  return suf
```

Figure 9: Algorithm SUFFIXES.

```
STRONG-MATCHING(x, m)
1  j ← 0
2  for i ← m − 1 downto −1
3       do if i = −1 or suf[i] = i + 1
4             then while j < m − 1 − i
5                        do sMatch[j] ← m − 1 − i
6                           j ← j + 1
7  for i ← 0 to m − 2
8       do sMatch[m − 1 − suf[i]] ← m − 1 − i
9  return sMatch
```

Figure 10: Algorithm STRONG-MATCHING.

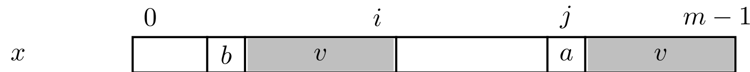

Figure 11: Variable i of algorithm STRONG-MATCHING. Situation where $suf[i] < i + 1$. The loop of lines 7-8 has the following invariants: $v = lcsuf(x, x[0 \,..\, i])$ and $a \neq b$ ($a, b \in \Sigma$) and $suf[i] = |v|$. Thus $sMatch[j] \leq m-1-i$ with $j = m - 1 - suf[i]$.

```
BEST-MATCHING(x, m)
 1  j ← 0
 2  for i ← m − 1 downto −1
 3      do if i = −1 or suf[i] = i + 1
 4            then while j < m − 1 − i
 5                      do for a ∈ A
 6                              do bMatch[j, a] ← m − 1 − i
 7                         j ← j + 1
 8  for i ← 0 to m − 2
 9      do sMatch[m − 1 − suf[i], x[i − suf[i]]] ← m − 1 − i
10  return sMatch
```

Figure 12: Algorithm Best-Matching.

The invariants of the second loop of algorithm Strong-Matching are presented in Fig. 11.

A slight modification of the algorithm Strong-matching leads to the algorithm Best-matching, presented in Fig. 12, that computes the best matching shift in time and space $O(m \times \sigma)$.

To prove the correctness of the algorithm Best-Matching we first recall the following lemma.

Lemma 3.1 ([9, 11]) *For $0 \leq i \leq m-2$, if $suf[i] = i+1$ then, for $0 \leq j < m-1-i$, $suf[j] \leq m-1-i$.*

The next lemma gives a bound on $bMatch[m-1-suf[i], a]$.

Lemma 3.2 *For $0 \leq i \leq m-2$ and $a \in \Sigma$, we have $bMatch[m-1-suf[i], a] \leq m-1-i$*

Proof Let $a \in \Sigma$ be such that $a \neq x[m-1-suf[i]]$. If $suf[i] < i+1$, the condition $Cs(m-1-suf[i], m-1-i)$ is satisfied since we have on one hand $m-1-i \leq m-1-suf[i]$ and on the other hand $x[i-suf[i]+1\,..\,i] = x[m-1-suf[i]+1\,..\,m-1]$. Moreover, the condition $Cob(m-1-suf[i], a, m-1-i)$ is also satisfied since $x[i-suf[i]] \neq x[m-1-suf[i]]$ by definition of suf. Thus $bMatch[m-1-suf[i], a] \leq m-1-i$.

Now if $suf[i] = i+1$, by Lemma 3.1, we have in particular for $j = m-1-suf[i] = m-i-2$, the inequality $bMatch[j, a] \leq m-1-i$. This ends the proof. □

We can now state the following:

Theorem 3.3 *The algorithm* Best-Matching *computes the table bMatch for the string x by the mean of the table suf for the same string.*

Proof We have to show, for each index j, $0 \leq j < m$ and for each letter $a \in \Sigma$, that the final value s attributed to $bMatch[j, a]$ by Best-Matching is the minimal value that satisfies the conditions $Cs(j, s)$ and $Cob(j, a, s)$.

Let us first assume that s results from an assignment during the execution of the loop of Lines 2–7. Thus the first part of the condition Cs is not satisfied. We check then by the mean of Lemma 3.1 that s is the minimal value that satisfies the second part of condition $Cs(j, s)$. In this case, $s = m - 1 - i$ for a value i that is such that $suf[i] = i + 1$ and $j < m - 1 - i$. This last inequality shows that the condition $Cob(j, a, d)$ is also satisfied. This proves the result in this situation, that is to say $s = bMatch[j, a]$.

Let us now assume that s results from an assignment during the execution of the loop of Lines 8–9. We thus have $j = m - 1 - suf[i]$ and $s = m - 1 - i$, and, after Lemma 3.2, $bMatch[j, a] \leq s$. We also have $0 < s \leq i$, this shows that the second parts of conditions $Cs(j, s)$ and $Cob(j, x[i - suf[i]], s)$ cannot be satisfied. As the quantity $m - 1 - i$ decreases during the execution of the loop, s is the smallest value of $m - 1 - i$ for which $j = m - 1 - suf[i]$. We thus have $s = suf[j]$. This ends the proof. □

Theorem 3.4 *The algorithm* Best-Matching *applied to a string of length m executes in time $O(m \times \sigma)$ (even when including the computation time of the intermediate table suf) and requires an extra space $O(m \times \sigma)$.*

Proof The space necessary to the computation (in addition to the string x and the table *suf*) consits of the table *bMatch* and some integer variables. Thus a space $O(m \times \sigma)$.

The execution of the loop at Lines 2–7 takes a time $O(m \times \sigma)$ since each operation executes in constant time for each value of i or for each value of j, and since these variables take $m + 1$ distinct values.

The loop of Lines 8–9 executes also in time $O(m)$, this shows the result. Including the computation time of table *suf* gives the same conclusion. □

4 EXPERIMENTAL RESULTS

To evaluate the efficiency of the Boyer–Moore string matching algorithm using the best matching shift in quadratic space we perform several experiences with different algorithms on different data sets.

4.1 Algorithms

We have tested 28 algorithms:

- The brute force algorithm (`BF`).
- Five different implementations of the Boyer–Moore algorithm:
  - with the occurrence and the strong matching shifts without fast loop (`BM`).
  - with the occurrence and the strong matching shifts with a fast loop (`BM`) as suggested in [4] (`BMfast`).

- only with the strong matching shift (BM1).
- with the best matching shift without fast loop (BM2).
- with the best matching shift with fast loop (BM2fast).

- Ten different variants of the Boyer–Moore algorithms:
  - The Horspool algorithm [16] (HOR).
  - The Tuned-BM algorithm [17] (TBM) with 3 unrolled shifts.
  - The Quick Search algorithm [30] (QS).
  - The SSABS algorithm [26] (SSABS).
  - The Zhu-Takaoka algorithm [31] (ZT).
  - The Berry-Ravindran algorithm [3] (BR).
  - The Smith algorithm [27] (Smith).
  - The Raita algorithm [24] (Raita).
  - The Skip Search algorithm [7] (Skip).
  - The Fast Search algorithm [5] (FS).

- Nine algorithms based on an index structure recognizing all the factors of x^R.
  - The Reverse Factor algorithm [19, 8] (RF1) where the suffix automaton of x^R is implemented in quadratic space with a transition matrix.
  - A simplification of the Reverse Factor algorithm (RF0) where the suffix automaton of x^R is implemented in quadratic space with a transition matrix. The simplification consists in each attempt to match the longest factor u of x that is a suffix of the window and in case of a mismatch to align the left end of the window with the beginning of u rather than with the longest suffix of u prefix of x as in the original Reverse Factor algorithm (RF1).
  - The Backward Oracle Matching [2] algorithm (BOM) where the factor oracle of x^R is implemented in linear space with lists of transitions.
  - The Backward Suffix Oracle Matching [2] algorithm (BSOM) where the factor oracle of x^R is implemented in linear space with lists of transitions.
  - The Backward Oracle Matching [2] algorithm (BOM2) where the factor oracle of x^R is implemented in quadratic space with a transition matrix.
  - The Backward Suffix Oracle Matching [2] algorithm (BSOM2) where the factor oracle of x^R is implemented in quadratic space with a transition matrix.

- The Backward Enhanced Suffix Array Matching algorithm (BESAM0) This is an original algorithm. It is a version of RF0 using the enhanced suffix array instead of the suffix automaton of x^R. The enhanced suffix array consists here of a permutation array, an array of longest common prefixes (LCP) and a `childtab` array implemented using $3m$ integers [1]. The permutation is computing with `libdivsufsort` which is a library developed by Yuta Mori and that provides a very fast and lightweight suffix sorting algorithm. It is based on the Ko–Aluru algorithm [18]. It is available at the following URL http://homepage3.nifty.com/wpage/software/libdivsufsort.html.
- The Backward Enhanced Suffix Array Matching algorithm (BESAM1). It is a version of RF1 using the enhanced suffix array instead of the suffix automaton of x^R.

• For short patterns, four algorithms using bitwise operations:

  - The Backward Nondeterministic Dawg Matching algorithm [21] (BNDM).
  - The Simplified Backward Nondeterministic Dawg Matching algorithm [23] (SBNDM).
  - The Simplified Backward Nondeterministic Dawg Matching algorithm which main loop starts with a test and loop-unrolling [15] (SBNDM2).
  - The Fast Average Optimal Shift Or algorithm [13] (FAOSO). It consist in considering sparse q-grams of x and unrolling u shifts, thus $q(\lceil m/q \rceil + u) \leq w$ should holds where w is the number of bits of a machine word. We use the following algorithm to determine the values of u and q.

```
1   if m ≤ 5
2     then (u, q) ← (4, 2)
3     else  if m ≤ 10
4             then if σ ≤ 4
5                     then (u, q) ← (4, 2)
6                     else  (u, q) ← (4, 4)
7             else  if m ≤ 20
8                     then if σ ≤ 4
9                             then (u, q) ← (3, 4)
10                            else  (u, q) ← (3, 5)
11                    else  if m ≤ 25
12                            then if σ ≤ 4
13                                    then (u, q) ← (2, 4)
14                                    else  (u, q) ← (1, 6)
15                            else  if σ ≤ 4
16                                    then (u, q) ← (1, 5)
17                                    else  (u, q) ← (1, 6)
```

In [13], it is said that the fastest way to compute $\lfloor \log_2(i) \rfloor$ with i an integer consists in casting i to real number and then extract the exponent from the

standardized floating point representation. We rather use a computation involving masking operations due to Brendan McKay and implemented in `Nauty`[1].

These algorithms have been coded in C in an homogeneous way to keep the comparison significant. The programs have been compiled using `gcc` with the full optimization option `-O3`. The machine we used has an Intel Pentium processor at 1300MHz running Linux Red Hat version 2.4.20-8. The running times for the search of 100 patterns have been measured using the `clock` function.

4.2 Data

We give experimental results of the running times for the above algorithms for different types of text: binary alphabet, alphabet of size 4, alphabet of size 8, alphabet of size 20, genome and natural language (English). We consider short patterns (odd length within 5 and 31) and long patterns (length power of two from 2^5 to 2^{10}). For each length we executed the search for 100 patterns randomly chosen from the text.

4.2.1 Binary alphabet

The alphabet is $\Sigma = \{\mathtt{A}, \mathtt{B}\}$. The text is composed of 4,000,000 characters and was randomly built. The distribution is uniform, with 49.977 per cent of `A` and 50.023 per cent of `B`.

4.2.2 Alphabet of size 4

The alphabet is $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$. The text is composed of 4,000,000 characters and was randomly built. The distribution is uniform, with 25.003 per cent of `A`, 25.02 per cent of `B`, 24.981 per cent of `C` and 24.996 per cent of `D`.

4.2.3 Alphabet of size 8

The alphabet is $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}\}$. The text is composed of 4,000,000 characters and was randomly built. The distribution is uniform, with 12.497 per cent of `A`, 12.52 per cent of `B`, 12.487 per cent of `C`, 12.506 per cent of `D`, 12.497 per cent of `E`, 12.493 per cent of `F`, 12.518 per cent of `G` and 12.482 per cent of `H`.

4.2.4 Alphabet of size 20

The alphabet is $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{J}, \mathtt{K}, \mathtt{L}, \mathtt{M}, \mathtt{N}, \mathtt{O}, \mathtt{P}\}$. The text is composed of 4,000,000 characters and was randomly built. The distribution is uniform (see Table 1).

[1] http://cs.anu.edu.au/~bdm/nauty/

Table 1: Character distribution for the text on a twenty letter alphabet.

A	B	C	D	E	F	G	H	I	J
5.001	5.001	4.986	5.001	5.014	5.003	5.001	4.994	5.011	5.017
K	L	M	N	O	P	Q	R	S	T
4.986	5.014	4.983	4.994	5.004	5.009	4.991	5.001	4.989	5.000

Table 2: Character distribution for the text in natural language.

LF	CR	␣	!	"	#	$	%	'	(	)	*
2.632	2.632	17.331	0.001	0.002	0.001	0.116	0.362	0.037	0.623	0.622	0.109
,	-	.	/	0	1	2	3	4	5	6	7
2.023	0.407	0.398	0.066	0.840	1.103	0.599	0.339	0.312	0.335	0.280	0.283
8	9	:	;	<	=	>	?	@	A	B	C
0.367	0.897	0.888	0.541	$\simeq 0.000$	0.009	$\simeq 0.000$	$\simeq 0.000$	$\simeq 0.000$	0.636	0.200	0.549
D	E	F	G	H	I	J	K	L	M	N	O
0.313	0.364	0.230	0.252	0.110	0.409	0.114	0.097	0.290	0.312	0.436	0.279
P	Q	R	S	T	U	V	W	X	Y	Z	[
0.389	0.009	0.228	0.465	0.315	0.260	0.059	0.136	0.007	0.047	0.034	0.023
\	]	^	_	`	a	b	c	d	e	f	g
$\simeq 0.000$	0.023	0.005	$\simeq 0.000$	0.005	5.949	0.901	2.195	2.061	6.590	1.045	1.066
h	i	j	k	l	m	n	o	p	q	r	s
1.521	4.819	0.072	0.378	2.964	1.916	4.832	4.468	1.345	0.054	4.522	3.575
t	u	v	w	x	y	z	{	}	~		
4.636	1.896	0.652	0.535	0.197	0.933	0.124	$\simeq 0.000$	$\simeq 0.000$	$\simeq 0.000$		

4.2.5 Natural language

We used the file `world192.txt` (The CIA world fact book) of the Large Canterbury Corpus[2]. The alphabet is composed of 94 different characters (see distribution in Table 2). The text is composed of 2,473,400 characters.

4.2.6 Genome

A genome is a DNA sequence composed of the four nucleotides, also called base pairs or bases: Adenine, Cytosine, Guanine and Thymine. The alphabet is thus $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$. 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` of the Large Canterbury Corpus. It is composed of 24.621 per cent of Adenine, 25.421 per cent of Cytosine, 25.364 per cent of Guanine and 24.594 per cent of Thymine.

4.3 Results

The results for short patterns (length less than 32) are presented in tables 3 to 8 and in figures 13 to 18. The results for long patterns (length more than 32) are presented in tables 9 to 14.

4.3.1 Short Patterns

Binary alphabet For lengths 5 and 7 `FAOSO` is the fastest algorithm, for length 9 it is `RFO` and for lengths 11 to 31 it is `BNDM2`.

[2] `http://www.data-compression.info/Corpora/CanterburyCorpus/`

Table 3: Results for short patterns on a binary alphabet

	5	7	9	11	13	15	17	19	21	23	25	27	29	31
BF	41.29	25.84	21.54	20.21	20.26	20.10	20.25	20.50	20.16	20.08	20.08	20.09	20.05	20.07
BM	26.90	9.28	4.10	3.12	2.97	2.77	2.67	2.44	2.33	2.24	2.21	2.03	1.94	2.00
BMfast	28.07	10.01	5.36	3.48	2.94	2.72	2.72	2.48	2.36	2.26	2.25	2.08	1.96	2.06
BM1	29.53	8.81	4.42	2.95	2.44	2.15	2.17	2.04	1.94	1.87	1.83	1.71	1.60	1.66
BM2	28.59	9.60	4.51	3.18	2.62	2.41	2.40	2.22	2.13	2.06	2.00	1.86	1.76	1.81
BM2fast	26.03	8.93	4.15	2.98	2.89	2.65	2.59	2.35	2.26	2.16	2.15	2.00	1.87	1.96
HOR	28.44	13.35	8.68	9.28	8.82	9.05	8.68	8.72	8.39	8.18	9.00	8.43	8.21	8.95
TBM	27.72	12.25	7.74	6.85	6.12	6.65	6.40	6.37	6.14	6.00	6.58	6.22	5.99	6.59
QS	34.60	18.62	12.75	12.11	12.13	12.04	11.69	11.53	11.25	11.65	11.40	11.72	11.62	11.74
SSABS	30.40	10.86	7.25	6.45	5.93	6.24	6.09	5.88	5.75	5.99	5.86	6.02	5.87	5.98
PAMA	30.37	10.02	5.02	3.95	3.46	3.19	3.13	2.88	2.77	2.66	2.61	2.41	2.27	2.37
ZT	30.07	10.10	5.51	4.02	3.50	3.18	3.21	2.91	2.80	2.70	2.57	2.55	2.45	2.45
BR	34.39	13.97	10.59	9.12	9.56	9.23	9.23	8.79	8.95	9.24	8.74	8.99	9.61	9.12
Smith	34.03	15.47	12.05	11.61	11.12	11.31	10.99	10.98	10.46	10.49	11.17	10.68	10.37	11.27
Raita	29.29	11.18	7.46	6.28	5.85	6.18	6.12	6.15	5.85	5.78	6.38	6.02	5.82	6.42
Skip	33.98	9.46	10.48	11.29	11.65	11.58	11.11	10.95	11.01	10.95	10.92	10.89	10.89	10.81
FS	28.34	9.80	5.08	3.05	2.56	2.25	2.31	2.13	2.02	1.96	1.91	1.77	1.68	1.73
RF1	25.52	9.45	3.16	2.73	2.64	2.25	2.10	1.86	1.68	1.57	1.46	1.39	1.33	1.25
RF0	26.88	10.14	2.58	2.59	2.48	2.12	1.91	1.67	1.51	1.40	1.31	1.23	1.17	1.12
BOM	27.71	11.29	5.50	4.61	3.92	3.42	3.19	2.84	2.61	2.40	2.24	2.11	2.00	1.87
BSOM	27.89	11.03	5.65	4.74	4.16	3.59	3.32	2.98	2.77	2.55	2.39	2.25	2.14	2.01
BOM2	24.78	10.12	4.49	3.00	2.27	1.85	1.78	1.64	1.37	1.24	1.16	1.07	1.01	0.96
BSOM2	28.16	10.01	5.57	3.36	3.03	2.69	2.43	2.14	1.99	1.83	1.65	1.53	1.46	1.38
BESAM0	40.69	18.35	12.44	11.05	9.80	8.92	7.82	7.04	6.50	6.12	5.75	5.44	5.14	4.88
BESAM1	40.85	13.61	10.65	9.99	8.65	8.03	7.07	6.43	5.98	5.65	5.36	5.08	4.80	4.55
BNDM	25.13	9.28	3.98	2.53	2.28	2.03	1.82	1.57	1.43	1.31	1.21	1.13	1.08	1.01
SBNDM	31.53	10.34	4.52	2.74	1.90	1.60	1.37	1.22	1.12	0.99	0.92	0.86	0.79	0.73
SBNDM2	28.39	9.38	3.66	2.20	1.58	1.30	1.13	0.96	0.91	0.84	0.75	0.71	0.65	0.62
FAOSO	10.48	4.70	3.74	3.90	4.84	4.93	1.15	1.13	2.49	2.47	1.57	2.93	2.97	2.19

Among the Boyer-Moore algorithms, as expected BM1 is almost always the fastest, except for lengths 5 and 9 where our new algorithm BM2fast overpasses it. In case of a binary alphabet the strong matching shift behaves exactly like the best matching shift thus the occurrence shift is useless.

Among the Boyer-Moore variants, for lengths 5, 7 and 9, TBM, Skip and PAMA are respectively the fastest. Then for lengths within 11 and 31 FS is the fastest.

Among the algorithms based on an index structure BOM2 is almost always the fastest except for length 7 where it is RF1 and lengths 9 and 11 where it is RF0.

Among the algorithms using bitwise operations, for lengths 5 and 7 FAOSO is the fastest and for lengths 9 to 31 it is BNDM2.

For a binary alphabet our new algorithms are never the fastest.

Alphabet of size 4 For length 5 our new algorithm BM2fast is the fastest algorithm, for lengths 9 to 31 it is BNDM2.

Among the Boyer-Moore algorithms, our new algorithm BM2fast is always the fastest.

Among the Boyer-Moore variants, for lengths 5 and 7, TBM is the fastest, for lengths 9 and 11 it is FS, then for lengths within 11 and 31 ZT is the fastest.

Among the algorithms based on an index structure BOM2 is almost always the fastest except for length 5 where it is RF0.

Among the algorithms using bitwise operations, for length 5 FAOSO is the fastest, and for lengths 7 to 31 it is BNDM2.

For an alphabet of size 4, our new algorithm BM2fast is the fastest for very short patterns.

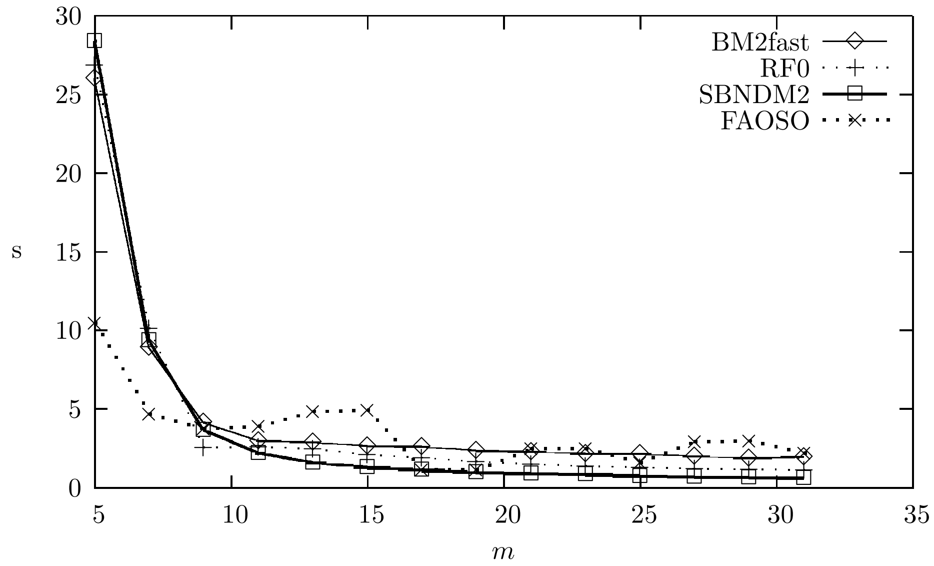

Figure 13: Results for short patterns on a binary alphabet.

Table 4: Results for short patterns on an alphabet of size 4

	5	7	9	11	13	15	17	19	21	23	25	27	29	31
BF	19.98	18.88	19.50	19.76	19.47	19.38	19.37	19.37	19.35	19.36	19.43	19.12	18.92	18.88
BM	3.25	2.19	2.03	1.78	1.76	1.66	1.58	1.58	1.55	1.58	1.55	1.51	1.48	1.41
BMfast	2.66	1.97	1.81	1.60	1.56	1.47	1.42	1.39	1.39	1.39	1.36	1.35	1.32	1.25
BM1	3.22	2.70	2.57	2.29	2.18	2.17	1.96	1.95	1.89	1.85	1.88	1.85	1.84	1.76
BM2	2.67	1.88	1.67	1.46	1.38	1.27	1.23	1.17	1.16	1.12	1.09	1.06	1.03	1.01
BM2fast	2.47	1.74	1.56	1.34	1.27	1.19	1.13	1.09	1.08	1.04	1.00	1.00	0.96	0.94
HOR	3.87	3.20	3.01	2.74	2.85	2.68	2.64	2.72	2.81	2.94	2.83	2.90	2.81	2.76
TBM	2.56	1.80	1.72	1.54	1.58	1.49	1.46	1.51	1.52	1.57	1.53	1.56	1.52	1.50
QS	6.00	5.95	5.67	5.31	5.43	5.21	5.36	5.21	5.54	5.46	5.63	5.35	5.37	5.48
SSABS	2.70	2.16	2.06	1.90	1.92	1.83	1.89	1.84	1.95	1.92	1.97	1.90	1.90	1.93
PAMA	3.43	2.76	2.43	2.13	2.03	1.86	1.78	1.71	1.69	1.64	1.58	1.52	1.49	1.45
ZT	3.07	2.17	1.79	1.50	1.35	1.23	1.15	1.08	1.04	1.02	0.97	0.96	0.93	0.91
BR	4.70	4.10	3.48	3.05	2.80	2.58	2.48	2.38	2.34	2.31	2.22	2.15	2.10	2.12
Smith	6.18	5.72	5.31	4.91	5.03	4.79	4.84	4.82	4.95	5.04	5.06	5.01	4.89	4.88
Raita	3.03	2.27	2.11	1.90	1.97	1.83	1.82	1.86	1.87	1.94	1.89	1.94	1.97	1.93
Skip	5.59	6.11	5.79	5.59	5.48	5.48	5.43	5.41	5.41	5.36	5.35	5.32	5.30	5.30
FS	2.83	1.84	1.70	1.50	1.49	1.39	1.33	1.31	1.31	1.31	1.29	1.25	1.24	1.18
RF1	3.24	2.39	1.89	1.56	1.37	1.20	1.10	0.99	0.93	0.86	0.80	0.76	0.73	0.69
RF0	2.90	2.15	1.80	1.46	1.30	1.16	1.05	0.96	0.89	0.82	0.77	0.73	0.69	0.66
BOM	4.90	4.22	3.52	3.01	2.75	2.48	2.30	2.13	2.00	1.87	1.78	1.68	1.60	1.52
BSOM	4.34	3.94	3.31	2.83	2.54	2.31	2.13	1.97	1.84	1.73	1.62	1.56	1.50	1.45
BOM2	2.88	1.94	1.61	1.33	1.16	1.03	0.95	0.86	0.80	0.75	0.69	0.66	0.64	0.61
BSOM2	3.01	2.36	1.90	1.59	1.42	1.27	1.15	1.07	0.99	0.93	0.86	0.81	0.77	0.74
BESAM0	11.38	8.49	7.13	6.14	5.49	5.04	4.63	4.28	3.96	3.75	3.50	3.34	3.16	3.03
BESAM1	10.83	8.00	6.77	5.82	5.27	4.85	4.42	4.11	3.90	3.63	3.45	3.29	3.10	3.01
BNDM	2.96	2.05	1.67	1.36	1.17	1.05	0.95	0.85	0.78	0.73	0.69	0.67	0.63	0.58
SBNDM	4.36	2.28	1.68	1.38	1.15	1.01	0.89	0.80	0.71	0.68	0.62	0.58	0.56	0.5
SBNDM2	2.51	1.53	1.26	1.01	0.86	0.74	0.67	0.61	0.55	0.54	0.49	0.49	0.46	0.45
FAOSO	2.43	1.72	1.63	1.35	1.54	1.54	0.53	0.52	0.63	0.63	0.60	1.36	1.34	1.31

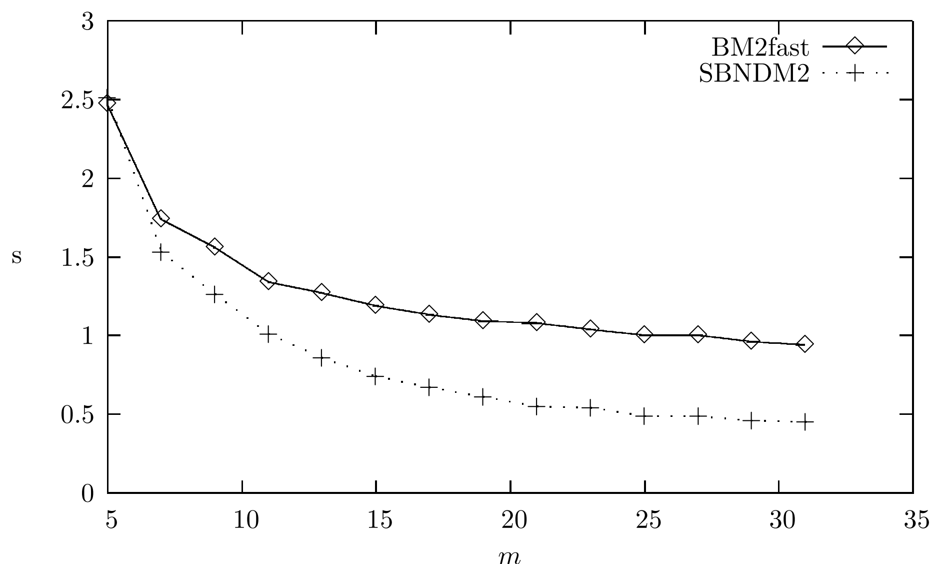

Figure 14: Results for short patterns on an alphabet of size 4.

Alphabet of size 8 For length 5 `TBM` is the fastest, for length 7 our new algorithm `BM2fast` is the fastest algorithm, for lengths 9 to 31 it is `BNDM2`.

Among the Boyer-Moore algorithms, our new algorithm `BM2fast` is always the fastest.

Among the Boyer-Moore variants, for lengths 5 to 23, `TBM` is the fastest, for lengths 21 to 31 it is `ZT`.

Among the algorithms based on an index structure `BOM2` is almost always the fastest except for length 5 where it is `RFO`.

Among the algorithms using bitwise operations, for length 5 `FAOSO` is the fastest and for lengths 7 to 31 it is `BNDM2`.

For an alphabet of size 8, our new algorithm `BM2fast` is the fastest for very short patterns.

Alphabet of size 20 For length 5 our new algorithm`BM2fast` is the fastest, for length 7 `TBM` is the fastest algorithm, for lengths 9 to 31 it is `BNDM2`.

Among the Boyer-Moore algorithms, our new algorithm `BM2fast` and `BMfast` are the fastest.

Among the Boyer-Moore variants, `TBM`, `FS` and also `SSABS` are the fastest.

Among the algorithms based on an index structure `BOM2` is almost always the fastest.

Among the algorithms using bitwise operations, `BNDM2` is always the fastest.

For an alphabet of size 20, our new algorithm `BM2fast` is the fastest for very short patterns.

Table 5: Results for short patterns on an alphabet of size 8

	5	7	9	11	13	15	17	19	21	23	25	27	29	31
BF	18.62	18.96	19.22	19.17	19.11	19.12	19.10	19.15	19.29	19.14	19.16	19.15	19.13	19.15
BM	1.57	1.17	1.06	0.96	0.89	0.85	0.84	0.79	0.79	0.76	0.76	0.76	0.77	0.77
BMfast	1.25	0.96	0.82	0.77	0.73	0.69	0.64	0.66	0.65	0.64	0.64	0.63	0.63	0.62
BM1	2.53	2.21	2.21	2.05	2.05	1.97	1.92	1.86	1.86	1.72	1.81	1.73	1.74	1.62
BM2	1.37	0.98	0.88	0.80	0.75	0.70	0.67	0.64	0.63	0.61	0.63	0.60	0.61	0.60
BM2fast	1.12	0.81	0.76	0.67	0.65	0.61	0.62	0.57	0.58	0.55	0.55	0.55	0.52	0.52
HOR	1.96	1.54	1.43	1.28	1.20	1.11	1.08	1.06	1.05	1.04	1.02	1.02	1.05	1.01
TBM	1.10	0.85	0.73	0.72	0.65	0.61	0.64	0.62	0.61	0.60	0.62	0.60	0.61	0.61
QS	4.60	3.90	3.44	3.23	3.03	2.92	2.86	2.76	2.73	2.71	2.65	2.65	2.72	2.73
SSABS	1.23	0.96	0.88	0.84	0.80	0.77	0.73	0.72	0.71	0.73	0.74	0.72	0.72	0.74
PAMA	2.23	1.66	1.49	1.34	1.21	1.14	1.09	1.03	1.00	0.95	0.94	0.93	0.92	0.92
ZT	1.83	1.45	1.13	1.01	0.85	0.75	0.71	0.66	0.61	0.63	0.57	0.58	0.57	0.55
BR	3.60	2.98	2.51	2.17	1.94	1.72	1.58	1.46	1.35	1.27	1.18	1.12	1.06	1.03
Smith	4.79	3.93	3.39	3.10	2.89	2.72	2.65	2.54	2.47	2.42	2.44	2.41	2.45	2.43
Raita	1.61	1.18	1.05	0.93	0.90	0.84	0.85	0.78	0.79	0.79	0.78	0.79	0.78	0.79
Skip	3.82	3.38	3.20	3.09	3.02	2.99	2.92	2.89	2.88	2.87	2.81	2.82	2.80	2.79
FS	1.29	0.91	0.83	0.74	0.71	0.68	0.69	0.63	0.63	0.63	0.64	0.63	0.62	0.62
RF0	1.90	1.45	1.13	0.98	0.84	0.76	0.69	0.61	0.59	0.58	0.58	0.56	0.55	0.50
RF1	1.92	1.48	1.24	1.04	0.89	0.80	0.72	0.70	0.64	0.59	0.55	0.54	0.55	0.52
BOM	3.36	3.00	2.70	2.49	2.25	2.14	1.97	1.87	1.76	1.66	1.55	1.47	1.39	1.35
BSOM	2.96	2.67	2.47	2.20	2.01	1.88	1.75	1.62	1.48	1.37	1.27	1.25	1.17	1.11
BOM2	1.92	1.31	1.06	0.87	0.75	0.68	0.63	0.57	0.56	0.55	0.51	0.48	0.54	0.47
BSOM2	2.03	1.52	1.23	1.00	0.88	0.76	0.69	0.64	0.62	0.58	0.55	0.55	0.54	0.52
BESAM0	7.39	6.12	5.22	4.65	4.21	3.87	3.60	3.36	3.12	3.00	2.83	2.70	2.55	2.44
BESAM1	7.52	6.10	5.23	4.65	4.17	3.86	3.59	3.35	3.15	2.97	2.83	2.68	2.56	2.48
BNDM	1.92	1.42	1.13	0.92	0.82	0.72	0.64	0.62	0.58	0.55	0.51	0.50	0.49	0.50
SBNDM	2.40	1.75	1.41	1.15	0.93	0.82	0.75	0.66	0.60	0.58	0.54	0.52	0.49	0.48
SBNDM2	1.75	0.90	0.68	0.60	0.52	0.50	0.46	0.42	0.45	0.41	0.42	0.41	0.39	0.41
FAOSO	1.73	1.50	0.85	0.68	0.70	0.65	0.66	0.66	1.21	1.21	1.07	1.13	1.11	1.16

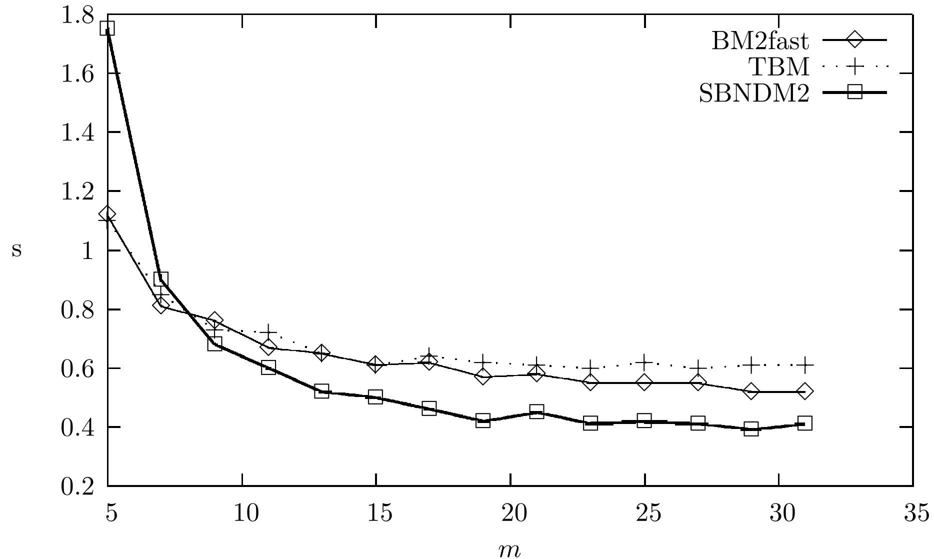

Figure 15: Results for short patterns on an alphabet of size 8.

Table 6: Results for short patterns on an alphabet of size 20

	5	7	9	11	13	15	17	19	21	23	25	27	29	31
BF	19.01	19.06	19.02	18.74	18.68	18.65	18.61	18.56	18.52	18.57	18.53	18.56	18.52	18.52
BM	1.07	0.86	0.70	0.63	0.56	0.54	0.51	0.53	0.47	0.48	0.47	0.49	0.46	0.47
BMfast	0.72	0.56	0.51	0.46	0.47	0.43	0.45	0.44	0.43	0.45	0.42	0.43	0.38	0.43
BM1	2.32	2.17	2.09	1.99	1.93	1.88	1.78	1.77	1.68	1.68	1.69	1.66	1.65	1.60
BM2	0.87	0.66	0.58	0.53	0.49	0.47	0.46	0.45	0.46	0.46	0.42	0.42	0.42	0.45
BM2fast	0.67	0.57	0.51	0.46	0.45	0.44	0.43	0.44	0.41	0.41	0.42	0.43	0.40	0.40
HOR	1.25	0.95	0.79	0.69	0.64	0.58	0.59	0.55	0.51	0.51	0.51	0.51	0.50	0.52
TBM	0.69	0.54	0.49	0.49	0.46	0.44	0.44	0.44	0.42	0.42	0.43	0.42	0.43	0.42
QS	3.83	3.01	2.54	2.24	2.04	1.83	1.71	1.58	1.53	1.44	1.41	1.36	1.31	1.28
SSABS	0.71	0.61	0.53	0.51	0.52	0.47	0.46	0.48	0.45	0.45	0.44	0.41	0.43	0.43
PAMA	1.64	1.22	1.00	0.88	0.80	0.68	0.67	0.62	0.58	0.56	0.56	0.55	0.55	0.51
ZT	1.64	1.18	0.94	0.77	0.71	0.64	0.56	0.56	0.53	0.51	0.51	0.52	0.50	0.50
BR	3.20	2.56	2.12	1.81	1.58	1.39	1.26	1.17	1.06	0.99	0.91	0.87	0.82	0.78
Smith	4.24	3.24	2.69	2.32	2.06	1.86	1.72	1.59	1.49	1.42	1.35	1.29	1.20	1.18
Raita	1.10	0.81	0.67	0.60	0.55	0.53	0.50	0.47	0.51	0.47	0.49	0.46	0.48	0.47
Skip	1.97	1.80	1.65	1.55	1.44	1.41	1.39	1.36	1.34	1.29	1.27	1.24	1.24	1.21
FS	0.76	0.58	0.52	0.50	0.47	0.46	0.46	0.44	0.40	0.42	0.41	0.42	0.41	0.42
RF1	1.28	1.05	0.93	0.79	0.70	0.62	0.56	0.55	0.52	0.51	0.48	0.44	0.44	0.42
RF0	1.20	1.01	0.87	0.75	0.66	0.59	0.56	0.53	0.51	0.50	0.46	0.44	0.43	0.43
BOM	2.40	2.19	2.03	1.95	1.95	1.88	1.75	1.64	1.51	1.46	1.37	1.34	1.23	1.23
BSOM	2.32	2.11	1.98	1.84	1.81	1.81	1.69	1.62	1.50	1.48	1.39	1.34	1.29	1.24
BOM2	1.18	0.97	0.85	0.74	0.66	0.58	0.55	0.49	0.50	0.45	0.48	0.46	0.41	0.44
BSOM2	1.23	1.02	0.87	0.74	0.68	0.61	0.56	0.51	0.49	0.50	0.46	0.46	0.43	0.39
BESAM0	6.03	4.99	4.33	3.92	3.62	3.38	3.13	2.96	2.79	2.63	2.52	2.38	2.28	2.23
BESAM1	6.40	5.24	4.49	4.14	3.79	3.48	3.20	3.03	2.88	2.73	2.57	2.48	2.39	2.29
BNDM	1.20	1.02	0.88	0.74	0.70	0.60	0.56	0.51	0.51	0.45	0.44	0.44	0.42	0.41
SBNDM	1.47	1.13	0.83	0.78	0.69	0.54	0.56	0.52	0.49	0.47	0.44	0.43	0.39	0.37
SBNDM2	1.35	0.67	0.48	0.44	0.41	0.40	0.35	0.37	0.33	0.33	0.31	0.32	0.28	0.31
FAOSO	1.46	1.03	0.76	0.50	0.51	0.49	0.49	0.49	1.07	1.07	1.09	1.06	1.08	1.07

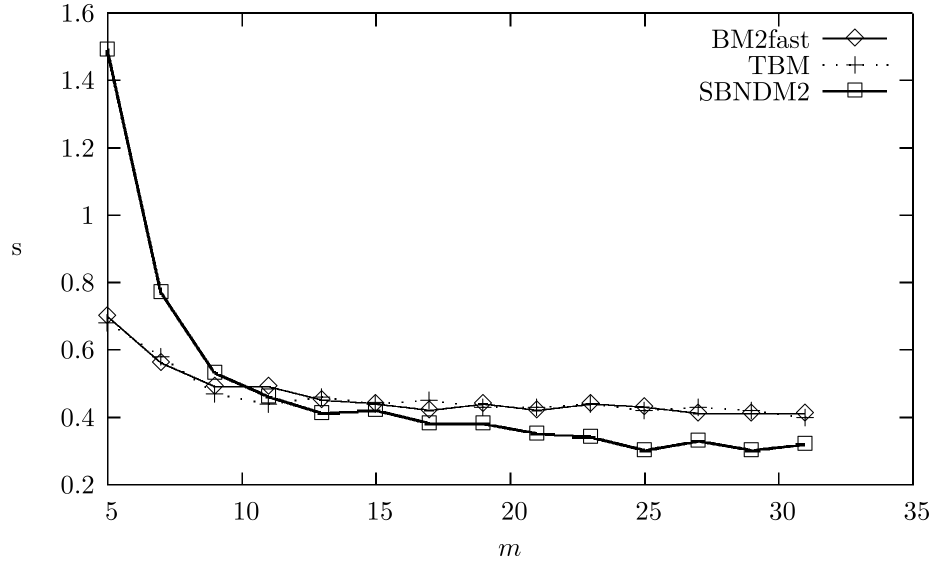

Figure 16: Results for short patterns on an alphabet of size 20.

Table 7: Results for short patterns on the *E. coli* genome

	5	7	9	11	13	15	17	19	21	23	25	27	29	31
BF	22.75	22.16	22.74	22.52	22.89	22.55	22.47	22.50	22.44	22.09	22.04	22.04	22.06	22.03
BM	3.86	2.47	2.17	2.10	1.99	1.96	1.88	1.92	1.89	1.78	1.82	1.74	1.74	1.76
BMfast	3.59	2.23	1.95	1.84	1.79	1.78	1.63	1.72	1.67	1.59	1.57	1.56	1.55	1.54
BM1	4.02	3.14	2.83	2.69	2.58	2.49	2.38	2.25	2.30	2.19	2.23	2.10	2.13	2.16
BM2	3.26	2.13	1.82	1.68	1.61	1.55	1.44	1.43	1.37	1.29	1.27	1.23	1.20	1.19
BM2fast	2.82	2.01	1.75	1.60	1.45	1.41	1.28	1.27	1.24	1.17	1.13	1.11	1.10	1.06
HOR	4.40	3.65	3.23	3.25	3.29	3.33	3.24	3.30	3.36	3.23	3.23	3.23	3.27	3.27
TBM	3.11	2.11	1.83	1.82	1.81	1.79	1.79	1.84	1.90	1.80	1.80	1.82	1.82	1.82
QS	7.29	6.70	6.47	6.43	6.20	6.30	5.95	6.38	6.23	6.00	6.06	6.23	6.14	6.01
SSABS	3.37	2.36	2.23	2.29	2.19	2.27	2.19	2.31	2.27	2.19	2.20	2.29	2.25	2.21
PAMA	4.37	3.04	2.63	2.49	2.34	2.30	2.13	2.12	1.99	1.92	1.86	1.78	1.79	1.73
ZT	3.45	2.45	2.05	1.72	1.53	1.41	1.32	1.25	1.19	1.16	1.12	1.09	1.05	1.03
BR	5.73	4.58	4.03	3.55	3.17	2.96	2.83	2.81	2.70	2.64	2.54	2.53	2.43	2.33
Smith	7.56	6.57	5.83	5.80	5.73	5.78	5.46	5.72	5.78	5.53	5.45	5.51	5.61	5.60
Raita	3.91	2.61	2.36	2.30	2.28	2.29	2.22	2.24	2.39	2.29	2.24	2.25	2.29	2.31
Skip	6.96	6.83	6.93	6.66	6.57	6.53	6.39	6.32	6.29	6.22	6.30	6.42	6.37	6.17
FS	3.16	2.11	1.83	1.80	1.69	1.67	1.54	1.57	1.54	1.52	1.54	1.52	1.54	1.51
RF1	4.06	2.65	2.19	1.80	1.57	1.38	1.27	1.14	1.06	0.98	0.93	0.88	0.84	0.79
RFO	3.80	2.50	2.07	1.72	1.49	1.33	1.21	1.10	1.02	0.96	0.88	0.84	0.80	0.77
BOM	5.94	4.48	3.94	3.51	3.25	2.95	2.72	2.57	2.31	2.20	2.09	2.01	1.85	1.76
BSOM	5.57	4.31	3.69	3.28	3.02	2.76	2.53	2.37	2.19	2.08	1.94	1.86	1.75	1.67
BOM2	3.37	2.31	1.88	1.54	1.34	1.18	1.08	1.01	0.92	0.87	0.81	0.76	0.73	0.70
BSOM2	3.53	2.69	2.20	1.83	1.61	1.45	1.32	1.22	1.13	1.06	0.98	0.93	0.89	0.84
BESAMO	12.13	9.80	8.09	7.03	6.32	5.67	5.25	4.87	4.56	4.25	4.00	3.78	3.64	3.45
BESAM1	11.52	9.36	7.65	6.66	5.99	5.49	5.03	4.72	4.41	4.14	3.92	3.71	3.55	3.41
BNDM	3.45	2.39	1.95	1.57	1.36	1.20	1.07	0.99	0.90	0.83	0.79	0.75	0.71	0.70
SBNDM	4.79	2.73	1.99	1.55	1.39	1.22	1.08	0.99	0.87	0.80	0.77	0.74	0.69	0.66
SBNDM2	3.86	1.87	1.37	1.13	0.96	0.92	0.81	0.73	0.68	0.64	0.65	0.56	0.56	0.56
FAOSO	2.59	2.52	1.30	1.54	1.70	1.62	1.61	1.60	2.27	2.33	1.59	1.54	1.60	1.40

***E. coli* genome** For lengths 5 to 9 `FAOSO` is the fastest algorithm, for lengths 11 to 31 it is `BNDM2`.

Among the Boyer-Moore algorithms, our new algorithm `BM2fast` is always the fastest.

Among the Boyer-Moore variants, for length 5 to 9 `TBM` is the fastest, for lengths 11 to 31 it is `ZT`.

Among the algorithms based on an index structure `BOM2` is always the fastest.

For the *E. coli* genome our new algorithms are never the fastest.

English text `SSABS` is the fastest for length 5. Our new algorithm `BM2fast` is the fastest for length 7, 9, 11 and 15. `TBM` is the fastest for lengths 7, 11, 13 and 15. `BNDM2` is the fastest for lengths 11, 15 and 17 to 31.

Among the Boyer-Moore algorithms, our new algorithm `BM2fast` and `BMfast` are the fastest.

Among the Boyer-Moore variants, `TBM`, `FS`, `SSABS` and also `Raita` are the fastest.

Among the algorithms based on an index structure `BOM2` is almost always the fastest.

Among the algorithms using bitwise operations, `BNDM2` is always the fastest except for length 5 where it is `FAOSO`.

For the English text our new algorithm `BM2fast` is almost always the fastest for patterns of length within 7 and 15 which is the typical case when searching for words in natural language texts.

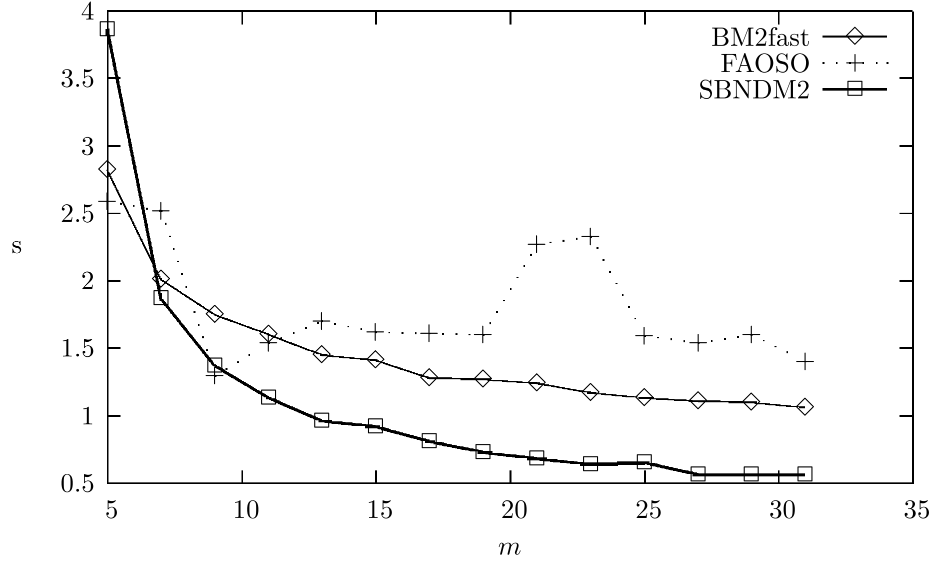

Figure 17: Results for short patterns on the *E. coli* genome.

Table 8: Results for short patterns on an English text

	5	7	9	11	13	15	17	19	21	23	25	27	29	31
BF	11.99	11.63	11.67	11.54	11.57	11.55	11.51	11.51	11.54	11.51	11.51	11.50	11.52	11.51
BM	1.00	0.58	0.44	0.39	0.36	0.32	0.31	0.31	0.29	0.29	0.29	0.29	0.27	0.28
BMfast	0.70	0.42	0.37	0.31	0.31	0.28	0.28	0.27	0.26	0.26	0.26	0.27	0.25	0.25
BM1	1.60	1.35	1.32	1.27	1.20	1.20	1.19	1.15	1.14	1.12	1.09	1.09	1.07	1.08
BM2	0.87	0.48	0.41	0.34	0.32	0.29	0.30	0.28	0.27	0.28	0.27	0.28	0.26	0.26
BM2fast	0.68	0.40	0.35	0.31	0.29	0.28	0.27	0.27	0.27	0.26	0.26	0.26	0.25	0.25
HOR	1.11	0.67	0.49	0.46	0.41	0.36	0.35	0.36	0.35	0.30	0.33	0.32	0.31	0.30
TBM	0.81	0.40	0.36	0.30	0.29	0.28	0.27	0.26	0.26	0.27	0.25	0.26	0.25	0.25
QS	2.49	1.89	1.54	1.31	1.18	1.06	0.98	0.90	0.85	0.80	0.79	0.74	0.72	0.68
SSABS	0.66	0.41	0.37	0.33	0.31	0.29	0.28	0.28	0.28	0.26	0.27	0.27	0.27	0.26
PAMA	1.31	0.81	0.66	0.56	0.49	0.43	0.41	0.38	0.35	0.35	0.34	0.33	0.33	0.32
ZT	1.22	0.81	0.64	0.54	0.46	0.42	0.39	0.36	0.35	0.34	0.33	0.33	0.33	0.32
BR	2.30	1.70	1.38	1.16	1.03	0.91	0.82	0.75	0.70	0.64	0.61	0.58	0.54	0.51
Smith	2.78	2.03	1.73	1.40	1.24	1.11	1.01	0.93	0.86	0.82	0.76	0.74	0.69	0.67
Raita	0.80	0.58	0.45	0.39	0.35	0.33	0.33	0.30	0.30	0.28	0.30	0.29	0.28	0.27
Skip	1.63	1.24	1.10	1.02	1.00	0.97	0.90	0.90	0.88	0.85	0.88	0.85	0.84	0.81
FS	0.69	0.43	0.36	0.32	0.31	0.29	0.28	0.28	0.26	0.27	0.27	0.27	0.25	0.25
RF1	0.99	0.63	0.57	0.48	0.42	0.39	0.37	0.34	0.32	0.32	0.30	0.30	0.28	0.28
RF0	1.24	0.65	0.61	0.47	0.41	0.38	0.37	0.33	0.32	0.33	0.31	0.30	0.29	0.27
BOM	1.72	1.40	1.25	1.19	1.18	1.12	1.08	1.02	0.98	0.93	0.90	0.85	0.83	0.81
BSOM	1.58	1.26	1.16	1.10	1.08	1.03	0.99	0.94	0.91	0.88	0.83	0.80	0.79	0.75
BOM2	0.92	0.66	0.56	0.46	0.40	0.38	0.34	0.33	0.31	0.31	0.29	0.28	0.28	0.27
BSOM2	0.95	0.66	0.56	0.46	0.43	0.39	0.35	0.33	0.33	0.31	0.30	0.29	0.29	0.26
BESAM0	3.98	3.18	2.72	2.44	2.24	2.07	1.94	1.82	1.72	1.65	1.56	1.54	1.48	1.44
BESAM1	4.22	3.27	2.88	2.57	2.36	2.20	2.07	1.95	1.85	1.75	1.67	1.58	1.53	1.44
BNDM	0.96	0.67	0.52	0.48	0.41	0.38	0.35	0.33	0.32	0.30	0.29	0.28	0.28	0.27
SBNDM	1.37	0.75	0.60	0.50	0.45	0.42	0.38	0.34	0.33	0.26	0.31	0.29	0.27	0.31
SBNDM2	1.30	0.53	0.41	0.30	0.30	0.28	0.26	0.22	0.22	0.23	0.20	0.21	0.18	0.21
FAOSO	1.03	0.66	0.49	0.33	0.31	0.31	0.32	0.33	0.66	0.69	0.68	0.69	0.68	0.68

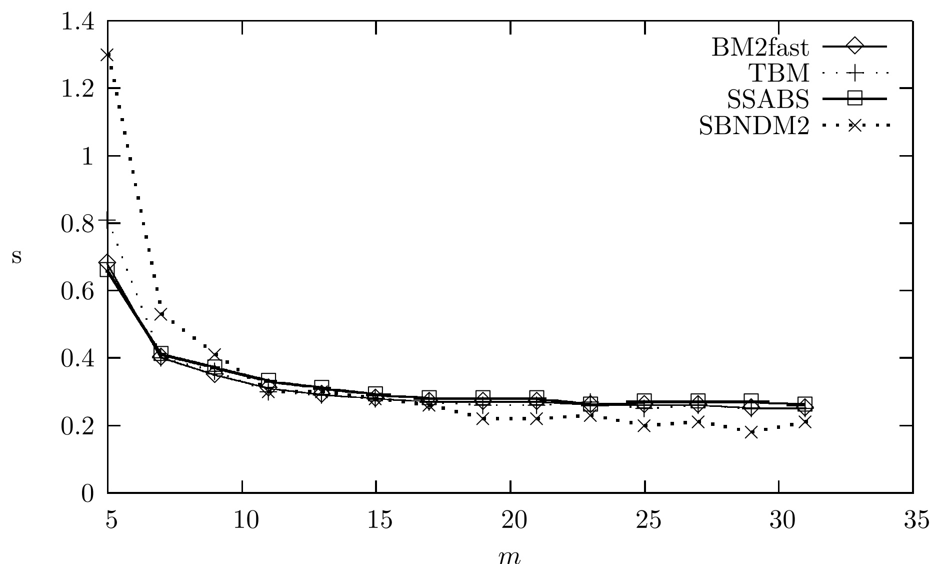

Figure 18: Results for short patterns on an English text.

4.3.2 Long Patterns

Binary alphabet For lengths 32 to 512 `BOM2` is the fastest algorithm, for length 1024 it is `BOM`.

Among the Boyer-Moore algorithms, as expected `BM1` is always the fastest.

Among the Boyer-Moore variants, `FS` is always the fastest.

Alphabet of size 4 For lengths 32 to 512 `BOM2` is the fastest algorithm, for length 1024 it is `BOM` and `BSOM`.

Among the Boyer-Moore algorithms, our new algorithm `BM2fast` is always the fastest.

Among the Boyer-Moore variants, `ZT` is always the fastest. `PAMA` is also the fastest for length 1024.

Alphabet of size 8 For lengths 32 to 512 `BOM2` is the fastest algorithm, for length 1024 it is `BSOM2`.

Among the Boyer-Moore algorithms, our new algorithm `BM2fast` is always the fastest except for length 1024 where it is our new algorithm `BM2`.

Among the Boyer-Moore variants, `ZT` is always the fastest.

Alphabet of size 20 For length 32 our new algorithms `BM2` and `BM2fast` are the fastest algorithms, for lengths 64 to 512 it is `BOM2` (with `RF1` for length 128) and for length 1024 it is `BSOM2`.

Table 9: Results for long patterns on a binary alphabet

	32	64	128	256	512	1024
BF	20.19	20.22	20.20	20.21	20.22	20.15
BM	1.96	1.44	1.26	1.10	0.92	0.80
BMfast	2.03	1.49	1.28	1.12	0.95	0.82
BM1	1.60	1.21	1.03	0.93	0.78	0.67
BM2	1.80	1.34	1.15	1.05	0.93	0.88
BM2fast	1.92	1.42	1.23	1.11	0.97	0.90
HOR	8.75	8.68	8.39	8.71	8.56	8.61
TBM	6.24	6.18	6.02	6.24	6.14	6.19
QS	11.05	11.28	11.28	11.27	10.95	11.31
SSABS	5.50	5.69	5.76	5.76	5.62	5.78
PAMA	2.32	1.72	1.44	1.30	1.10	0.99
ZT	2.42	1.83	1.54	1.34	1.13	0.99
BR	8.84	9.31	9.40	9.25	8.61	9.41
Smith	11.19	10.95	10.72	11.06	11.00	10.97
Raita	6.28	6.19	6.11	6.34	6.17	6.29
Skip	10.57	10.56	11.20	10.83	10.64	10.55
FS	1.69	1.26	1.07	0.99	0.81	0.71
RF1	1.23	0.78	0.77	0.46	0.39	0.67
RF0	1.08	0.68	0.69	0.42	0.35	0.65
BOM	1.82	1.05	0.93	0.54	0.27	0.15
BSOM	1.93	1.17	1.02	0.55	0.29	0.16
BOM2	0.92	0.60	0.65	0.38	0.21	0.19
BSOM2	1.32	0.75	0.77	0.41	0.24	0.20
BESAM0	4.69	2.72	1.91	1.13	0.66	0.41
BESAM1	4.40	2.65	1.95	1.14	0.67	0.41

Table 10: Results for long patterns on an alphabet of size 4

	32	64	128	256	512	1024
BF	19.42	19.43	19.80	19.17	19.20	19.17
BM	1.46	1.21	1.15	0.99	0.96	0.81
BMfast	1.28	1.06	1.02	0.88	0.83	0.72
BM1	1.74	1.42	1.29	1.10	1.08	0.89
BM2	1.00	0.79	0.75	0.66	0.57	0.57
BM2fast	0.92	0.73	0.72	0.62	0.55	0.55
HOR	2.79	2.62	2.81	2.66	2.84	2.92
TBM	1.64	1.44	1.56	1.47	1.54	1.55
QS	5.28	5.10	5.52	5.44	5.37	5.25
SSABS	1.95	1.83	2.00	1.95	1.93	1.88
PAMA	1.46	1.11	0.98	0.85	0.71	0.65
ZT	0.89	0.78	0.78	0.74	0.70	0.65
BR	1.99	1.78	1.81	1.85	1.78	1.82
Smith	4.86	4.58	4.99	4.74	4.90	4.89
Raita	1.87	1.77	1.89	1.76	1.85	1.88
Skip	5.10	5.09	5.78	5.45	5.25	5.12
FS	1.19	0.99	0.94	0.83	0.77	0.69
RF1	0.67	0.49	0.57	0.33	0.25	0.49
RF0	0.64	0.48	0.50	0.32	0.25	0.48
BOM	1.49	0.90	0.85	0.45	0.25	0.14
BSOM	1.38	0.84	0.84	0.45	0.23	0.14
BOM2	0.58	0.42	0.43	0.27	0.15	0.15
BSOM2	0.71	0.48	0.52	0.29	0.17	0.16
BESAM0	2.83	1.74	1.41	0.82	0.49	0.32
BESAM1	2.79	1.75	1.41	0.83	0.48	0.32

Table 11: Results for long patterns on an alphabet of size 8

	32	64	128	256	512	1024
BF	18.24	19.17	19.11	19.17	18.85	18.78
BM	0.77	0.73	0.71	0.68	0.62	0.56
BMfast	0.63	0.60	0.57	0.57	0.51	0.49
BM1	1.70	1.46	1.33	1.34	1.04	0.98
BM2	0.58	0.55	0.54	0.50	0.45	0.46
BM2fast	0.54	0.50	0.53	0.47	0.42	0.48
HOR	1.04	0.98	1.03	1.05	1.02	1.03
TBM	0.61	0.61	0.60	0.62	0.62	0.62
QS	2.68	2.63	2.68	2.66	2.62	2.71
SSABS	0.72	0.72	0.71	0.71	0.69	0.74
PAMA	0.89	0.73	0.69	0.64	0.57	0.51
ZT	0.53	0.49	0.48	0.47	0.46	0.44
BR	0.97	0.65	0.71	0.68	0.64	0.69
Smith	2.42	2.37	2.38	2.42	2.40	2.40
Raita	0.76	0.79	0.73	0.74	0.77	0.78
Skip	2.77	2.69	3.37	2.94	2.77	2.73
FS	0.60	0.58	0.58	0.56	0.51	0.46
RF1	0.51	0.43	0.47	0.30	0.23	0.39
RF0	0.50	0.39	0.35	0.21	0.20	0.40
BOM	1.30	0.78	0.76	0.40	0.23	0.13
BSOM	1.07	0.68	0.73	0.39	0.22	0.14
BOM2	0.47	0.38	0.33	0.17	0.12	0.12
BSOM2	0.52	0.38	0.37	0.22	0.14	0.14
BESAM0	2.40	1.47	1.23	0.70	0.42	0.27
BESAM1	2.40	1.47	1.25	0.72	0.45	0.30

Table 12: Results for long patterns on an alphabet of size 20

	32	64	128	256	512	1024
BF	18.72	18.70	18.59	19.32	18.61	18.67
BM	0.46	0.46	0.43	0.44	0.45	0.40
BMfast	0.42	0.39	0.40	0.41	0.35	0.39
BM1	1.67	1.61	1.52	1.37	1.14	1.02
BM2	0.40	0.40	0.44	0.42	0.40	0.43
BM2fast	0.40	0.40	0.42	0.38	0.36	0.43
HOR	0.50	0.48	0.49	0.45	0.50	0.50
TBM	0.42	0.41	0.38	0.36	0.41	0.42
QS	1.25	1.07	1.08	1.13	1.11	1.11
SSABS	0.43	0.41	0.43	0.42	0.40	0.44
PAMA	0.54	0.51	0.49	0.47	0.41	0.44
ZT	0.50	0.39	0.50	0.30	0.21	0.18
BR	0.74	0.51	0.61	0.43	0.33	0.30
Smith	1.19	0.99	0.93	0.97	0.97	0.92
Raita	0.46	0.45	0.43	0.46	0.47	0.44
Skip	1.27	1.15	1.85	1.49	1.27	1.15
FS	0.44	0.39	0.43	0.40	0.37	0.38
RF1	0.42	0.37	0.26	0.20	0.22	0.41
RF0	0.41	0.36	0.28	0.17	0.16	0.39
BOM	1.14	0.69	0.73	0.37	0.21	0.15
BSOM	1.18	0.76	0.74	0.36	0.22	0.14
BOM2	0.42	0.33	0.26	0.14	0.07	0.10
BSOM2	0.44	0.34	0.27	0.17	0.11	0.09
BESAM0	2.18	1.34	1.17	0.68	0.43	0.26
BESAM1	2.24	1.36	1.21	0.66	0.43	0.29

Among the Boyer-Moore algorithms, our new algorithms BM2 and BM2fast are always the fastest.

Among the Boyer-Moore variants, TBM is the fastest for lengths 32 and 128 to 1024 while ZT and FS are the fastest for length 64.

***E. coli* genome** For lengths 32 to 512 BOM2 is the fastest algorithm, for length 1024 it is BSOM.

Among the Boyer-Moore algorithms, our new algorithm BM2fast is always the fastest.

Among the Boyer-Moore variants, ZT is always the fastest. PAMA is also fast for lengths 512 and 1024.

Table 13: Results for long patterns on the *E. coli* genome

	32	64	128	256	512	1024
BF	23.46	25.85	23.31	23.38	23.39	23.56
BM	1.77	1.55	1.46	1.32	1.09	1.02
BMfast	1.56	1.35	1.30	1.13	0.97	0.93
BM1	2.04	1.81	1.64	1.47	1.23	1.19
BM2	1.22	0.99	0.96	0.84	0.71	0.71
BM2fast	1.12	0.91	0.89	0.78	0.66	0.67
HOR	3.35	3.41	3.47	3.48	3.36	3.52
TBM	1.84	1.87	1.92	1.88	1.78	1.91
QS	6.26	6.37	8.68	9.05	7.23	6.38
SSABS	2.31	2.33	2.46	2.31	2.40	2.39
PAMA	1.77	1.40	1.28	1.07	0.87	0.83
ZT	1.11	0.98	0.99	0.94	0.86	0.78
BR	2.47	2.23	2.13	2.29	2.27	2.20
Smith	5.65	5.85	5.91	5.71	5.69	5.86
Raita	2.17	2.21	2.25	2.20	2.13	2.25
Skip	6.32	6.21	6.99	6.57	6.29	6.01
FS	1.37	1.20	1.17	1.03	0.91	0.86
RF1	0.82	0.60	0.69	0.40	0.31	0.53
RF0	0.80	0.62	0.64	0.39	0.30	0.52
BOM	1.81	1.11	1.03	0.57	0.29	0.18
BSOM	1.69	1.04	1.05	0.55	0.31	[0.17]
BOM2	[0.71]	[0.52]	[0.53]	[0.31]	[0.20]	0.18
BSOM2	0.87	0.59	0.64	0.36	0.22	0.20
BESAM0	3.50	2.11	1.69	1.01	0.59	0.38
BESAM1	3.48	2.16	1.75	1.05	0.61	0.38

Table 14: Results for long patterns on an English text

	32	64	128	256	512	1024
BF	11.92	11.91	11.90	11.92	11.98	11.99
BM	0.27	0.25	0.25	0.19	0.13	0.10
BMfast	0.24	0.24	0.24	0.17	0.13	0.09
BM1	1.10	1.02	0.98	0.89	0.81	0.80
BM2	0.25	0.24	0.27	0.23	0.20	0.24
BM2fast	0.26	0.22	0.27	0.22	0.22	0.25
HOR	0.31	0.26	0.27	0.20	0.15	0.11
TBM	0.25	0.23	0.23	0.18	0.13	0.09
QS	0.67	0.48	0.46	0.39	0.32	0.25
SSABS	0.26	0.24	0.24	0.18	0.13	0.09
PAMA	0.31	0.29	0.29	0.24	0.20	0.21
ZT	0.31	0.29	0.31	0.19	0.12	0.10
BR	0.51	0.35	0.41	0.30	0.19	0.12
Smith	0.65	0.44	0.47	0.38	0.27	0.21
Raita	0.28	0.24	0.26	0.20	0.14	0.10
Skip	0.80	0.77	1.16	0.96	0.85	0.79
FS	0.26	0.24	0.26	0.18	0.13	0.10
RF1	0.28	0.23	0.19	0.14	0.15	0.40
RF0	0.28	0.23	0.19	0.12	0.15	0.39
BOM	0.77	0.52	0.49	0.30	0.17	0.13
BSOM	0.72	0.51	0.49	0.29	0.17	0.10
BOM2	0.27	0.21	0.16	0.09	0.06	0.10
BSOM2	0.27	0.19	0.16	0.09	0.07	0.09
BESAM0	1.40	0.96	0.87	0.50	0.33	0.25
BESAM1	1.47	0.94	0.84	0.52	0.33	0.26

English text For lengths 32 BMfast is the fastest. For lengths 64 to 256 BSOM2 is the fastest. For lengths 128 to 512 BOM2 is the fastest. For length 1024 it is BMfast, TBM, SSABS and BSOM2.

Among the Boyer-Moore algorithms, our new algorithm BMfast is always the fastest except for length 64 where it is BM2fast.

Among the Boyer-Moore variants, TBM, SSABS and FS are the fastest.

5 General interpretations

Our new algorithm BM2fast is the best for short patterns (length 5 to 7) on alphabets of size 4 to 20 and for patterns of length within 5 to 15 on natural language.

The FAOSO algorithm is efficient for short patterns (length 5 to 7) on small

size alphabet (2 to 4).

The `TBM` algorithm is efficient for short patterns: length 5 on an alphabet of size 8, length 7 on an alphabet of size 20 and length within 7 and 15 on natural language.

The `BNDM2` is efficient in all the other cases for short patterns: length within 11 and 31 for a binary alphabet; length within 7 and 31 for an alphabet of size 4; length within 9 and 31 for an alphabet of size 8 to 20; length within 15 and 31 for a natural language;

For long patterns the `BOM2` algorithm is the most efficient algorithm except for very long patterns (length 1024) where the quadratic size of the implementation of the factor oracle may be a problem.

Though the suffix array is a very space-economical index structure and is very efficient when performing string matching on a fixed text its utilization as an index structure of the reverse pattern gives very poor performance.

6 CONCLUSION

In this paper we presented a simple algorithm for computing the best matching shift of the Boyer–Moore algorithm. We conducted experiments showing that in some cases the algorithm using this shift is the most efficient string matching algorithm. These cases are:

- alphabet of size 4 and pattern length around 5,
- alphabet of size 8 to 20 and pattern length within 5 and 7,
- natural language and pattern length within 7 and 15.

7 ACKNOWLEDGEMENTS

The authors are grateful to Jan Holub for providing the code of the `SBNDM2` algorithm and to Florent Hivert for pointing out the masking method for computing the log of an integer.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] C. Allauzen, Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartosek, editors, *Proceedings of 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.

[3] 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.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[5] D. Cantone and S. Faro. Fast-search: A new efficient variant of the boyer-moore string matching algorithm. In K. Jansen, M. Margraf, M. Mastrolilli, and J. D. P. Rolim, editors, *Proceedings of the 2nd International Workshop on Experimental and Efficient Algorithms*, number 2647 in Lecture Notes in Computer Science, pages 47–58, Ascona, Switzerland, 2003. Springer-Verlag, Berlin.

[6] C. Charras and T. Lecroq. *Handbook of exact string matching algorithms.* King's College London Publications, 2004.

[7] 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, New Jersey, 1998. Springer-Verlag, Berlin.

[8] 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.

[9] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte*. Vuibert, 2001. In French.

[10] M. Crochemore, C. Hancart, and T. Lecroq. A unifying look at the Apostolico-Giancarlo string matching algorithm. *Journal of Discrete Algorithms*, 1(1):37–52, 2003.

[11] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings.* Cambridge University Press, 2007.

[12] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[13] K. Fredriksson and S. Grabowski. Practical and optimal string matching. In *Proceedings of SPIRE'2005*, LNCS 3772, pages 374–385, 2005.

[14] D. Gusfield. *Algorithms on strings, trees and sequences.* Cambridge University Press, 1997.

[15] 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, 2005. http:://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf.

[16] R. N. Horspool. Practical fast searching in strings. *Software – Practice & Experience*, 10(6):501–506, 1980.

[17] A. Hume and D. M. Sunday. Fast string searching. *Software – Practice & Experience*, 21(11):1221–1248, 1991.

[18] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210, Morelia, Michocán, Mexico, 2003. Springer-Verlag, Berlin.

[19] T. Lecroq. A variation on the Boyer-Moore algorithm. *Theoretical Computer Science*, 92(1):119–144, 1992.

[20] S. Lu, F. Cao, and Y. Lu. Pama: a fast string matching algorithm. *International Journal on Foundations of Computer Science*, 17(2):357–378, 2006.

[21] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithms*, 5:4, 2000.

[22] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[23] Hannu Peltola and Jorma Tarhio. Alternative algorithms for bit-parallel string matching. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval*, volume 2857 of *Lecture Notes in Computer Science*, pages 80–94, Manaus, Brazil, 2003. Springer-Verlag, Berlin.

[24] T. Raita. Tuning the boyer-moore-horspool string searching algorithm. *Software – Practice & Experience*, 22(10):879–884, 1992.

[25] W. Rytter. A correct preprocessing algorithm for Boyer–Moore string searching. *SIAM Journal on Computing*, 9(3):509–512, 1980.

[26] S. S. Sheik, S. K. Aggarwal, A. Poddar, N. Balakrishnan, and K. Sekar. A fast pattern matching algorithm. *J. Chem. Inf. Comput. Sci.*, 44:1251–1256, 2004.

[27] P.D. Smith. Experiments with a very fast substring search algorithm. *Software – Practice & Experience*, 21(10):1065–1074, 1991.

[28] W. F. Smyth. *Computing Patterns in Strings.* Addison Wesley Longman, 2002.

[29] G. A. Stephen. *String searching algorithms*. World Scientific Press, 1994.

[30] D. M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[31] R. F. Zhu and T. Takaoka. On improving the average case of the Boyer–Moore string matching algorithm. *J. Inform. Process.*, 10(3):173–177, 1987.