Experimental on results on string matching over infinite alphabets

Thierry Lecroq
LIR (Laboratoire d'Informatique de Rouen), Université de Rouen,
Faculté des Sciences et des Techniques, 76128 Mont-Saint-Aignan Cedex, France,
(e-mail: lecroq@dir.univ-rouen.fr)

Abstract

Various string matching algorithms have been designed and some experimental works on string matching over finite alphabets have been performed but string matching over infinite alphabets has been little investigated. We present here experimental results where symbols are taken among potentially infinite sets such as integers, reals or composed structures. These results show that in most cases it is better to decompose each symbols into a sequence of bytes and use algorithms which assume that the alphabet is finite which enable to use heuristics on characters.

INTRODUCTION

The exact string matching problem consists in finding one or more generally all the occurences of a word x in a text y. Both x and y are built over an alphabet Σ. Numerous algorithms have been designed to solve this problem (see [1] and [2]). All these algorithms can be divided in two categories: the algorithms that assumed that the alphabet is finite and the others. The faster algorithms in practice are of the former category (see [3]) because they use heuristics based on the alphabet. However they need to store a particular value for each symbol.

No study had been done in practice when searching over a potentially infinite alphabet. Obviously this case can reduced to a search over a finite alphabet (namely characters) decomposing both the word and the pattern into sequences of bytes, subsequently dealing with longer entities. The question which arises is whether it is faster to search with the actual symbol (and using algorithms over infinite alphabets which are known to be slower) or decomposing the word and the text in bytes (and using algorithms over finite alphabets but lengthening the word and the text) ? This study tries to answer this question from a practival viewpoint.

We perform experiments with four different alphabets: short integers on two bytes, reals on four bytes, reals on eight bytes and a structure on 32 bytes. For each alphabet, we performed various searches with string matching algorithms over infinite alphabets (Brute Force algorithm, Brute Force algorithm with a guard, Knuth-Morris-Pratt algorithm [4], Boyer-Moore algorithm [5] only with matching shift and Reverse Factor algorithm [6] and [7]) and with string matching algorithms over finite alphabets (Scan for First Character algorithm [8], Boyer-Moore algorithm, Horspool algorithm [8], Quick Search algorithm [9], Tuned Boyer-Moore algorithm [10] and Reverse Factor algorithm).

```
BRUTE-FORCE
1  for j ← 0 to n − m
2      do   if y[j, j + m − 1] = x
3              then OUTPUT(j)
```

Figure 1: The Brute Force algorithm.

THE ALGORITHMS

Exact string matching consists in finding all the occurrences of a word $x = x[0, m-1]$ of length m in a text $y = y[0, n-1]$ of length n. Both x and y are built over the same alphabet Σ. We are interested here, in the problem where the word x is given first, which allows some preprocessing phase. The word can then be searched in various texts.

A string matching algorithm works as follows: it first aligns the left end of the word with the left end of the text, it then checks if there is a match between the word and the text (this specific work is called an *attempt*) then it *shifts* the word to the right and repeats the same processus again until the right end of the word goes beyond the right end of the text. We will associate each attempt with the text position j where x is aligned with $y[j, j+m-1]$.

The various string matching algorithm differ both in the way they perform the symbol comparisons during the attempts and in the way they compute the shifts.

The Knuth-Morris-Pratt algorithm [4], which was the first discovered linear time string matching algorithm, performs symbol comparisons from left to right and computes the shifts with specific borders of the prefixes of the word x.

The Boyer-Moore algorithm [5] (and its variants), which is known to be very fast in practice, performs the symbol comparisons from right to left and computes the shifts with suffixes of the word x.

Infinite alphabets

In the case where the alphabet is potentially infinite, a preprocessing phase on the word is allowed but no information can be retained for each individual symbol.

Brute Force

The Brute Force algorithm (BF for short) can perform the symbol comparisons in any order. It is characterized by shifts of length 1 after each attempt. It does not need any preprocessing phase. The Brute Force algorithm is shown Figure 1. It has a quadratic worst case time complexity and uses constant extra space.

Brute Force with a guard

In practice, the test in line 2 in the Brute Force algorithm of Figure 1 can be very expensive and will most of the time lead to a mismatch. One solution is to use Raita's trick (see [11]), which Smith (see [12]) showed to be due to compiler effects. This trick consists in saving the first symbol of x in a guard and to checked if there is a full match only if there is a match between this guard and its corresponding symbol in the text during each attempt. The Brute

```
BRUTE-FORCE-WITH-A-GUARD
1  firstsymb ← x[0]
2  for j ← 0 to n − m
3      do    if y[j] = firstsymb andif y[j + 1, j + m − 1] = x[1, m − 1]
4               then OUTPUT(j)
```

Figure 2: The Brute Force algorithm with a guard.

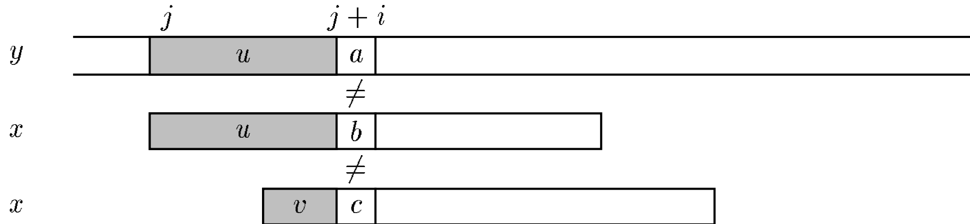

Figure 3: Shift in the Knuth-Morris-Pratt algorithm (v suffix of u).

Force string matching algorithm with a guard (BFG for short) is depicted Figure 2. It has still a quadratic worst case time complexity but it runs three times faster than the Brute Force algorithm (see Experiments).

Knuth-Morris-Pratt

The Knuth-Morris-Pratt algorithm [4] (KMP for short) was the first discovered linear time string matching algorithm. The symbol comparisons are done from left to right. It uses a precomputed table *next* defined as follows:

$$next[i] = \begin{cases} \text{length of the longest border of } x[0, i-1] \text{ followed by a symbol different of } x[i] \\ -1 \text{ if there exists no such border} \end{cases}$$

During an attempt j where x is aligned with $y[j, j+m-1]$, when a prefix $x[0, i-1] = u$ is matched and a mismatch occurred with $x[i] = b$ and $y[j+i] = a$, then $next[i] = v$ is the longest prefix of x which can reasonably match a suffix of u after a shift, the word is then shifted by $i - next[i]$ (see Figure 3).

The values of the table *next* depend only on the word x and can be computed in $O(m)$ time in a preprocessing phase.

The Knuth-Morris-Pratt algorithm is given Figure 4. It finds all the occurrences of a word of length m in a text of length n in $O(m+n)$ time and needs $O(m)$ extra space.

Boyer-Moore

The Boyer-Moore algorithm [5] performs the symbol comparisons from right to left. It usually performs the shifts with two functions: the *matching shift* which depends only on the word x and the *occurrence shift* which depends on the alphabet. In the case of an infinite alphabet we will be able to use only the matching shift. It uses a precomputed table *delta*.

```
KNUTH-MORRIS-PRATT
1  i ← 0
2  j ← 0
3  while j < n
4     do    while i > −1 andif y[j] ≠ x[i]
5              do    i ← next[i]
6           i ← i + 1
7           j ← j + 1
8           if i = m
9              then OUTPUT(j − m + 1)
10                  i ← next[i]
```

Figure 4: The Knuth-Morris-Pratt algorithm.

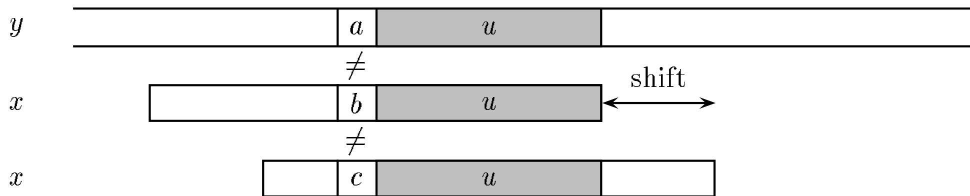

Figure 5: matching shift, u reappears preceded by a character different from b.

Let us defined two conditions:

$$cond_1(i,s): \text{ for each } k \text{ such that } i < k < m \text{ then } s \geq k \text{ or } x[k-s] = x[k]$$

and

$$cond_2(i,s): \text{ if } s < i \text{ then } x[i-s] \neq x[j]$$

Then the table *delta* is defined as follows: for $0 \leq i < m$

$$delta[i+1] = \min\{s > 0 \mid cond_1(i,s) \text{ and } cond_2(i,s) \text{ hold}\}$$

and $delta[0] = per(x)$.

The values of the table *delta* depends only on the word x and can be computed in $O(m)$ time in a preprocessing phase.

During an attempt j where the word x is aligned with $y[j, j+m-1]$, when a suffix $x[i+1, m-1] = u$ is matched and a mismatch occurs between $x[i] = b$ and $y[i+j] = a$. The matching shift consists in aligning the segment $y[j+i+1, j+m-1] = u$ with its rightmost occurrence in x that is preceded by a symbol different from $x[i] = b$ (see Figure 5). If there exists no such segment, the shift consists in aligning the longest suffix v of u with a matching prefix of x (see Figure 6). The word is then shifted by $delta[i+1]$ positions to the right.

In the algorithm presented in Figure 7 we add a fast loop (lines 5 and 6) which only shifts the word by 1 position to the right until there is a match with the last symbol of x (to the contrary of the fast loop of the fast algorithm presented in [5] which shifts the word with the occurrence shift of the text symbol). This implies to set $y[n]$ to $x[m-1]$ before the searching

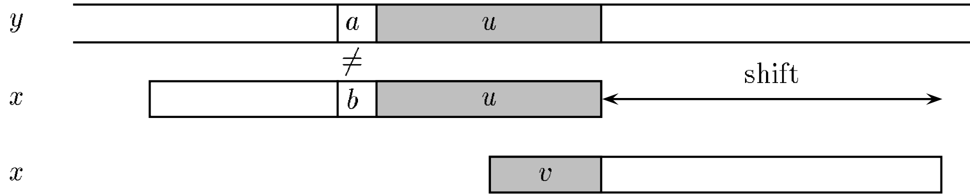

Figure 6: matching shift, only a prefix of u reappears in x.

```
BOYER-MOORE
1  lastsymb ← x[m − 1]
2  y[n] ← lastsymb
3  j ← m − 1
4  while j < n
5     do   while y[j] ≠ lastsymb
6             do    j ← j + 1
7          i ← m − 2
8          while i ≥ 0 andif y[j − m + 1 + i] = x[i]
9             do    i ← i − 1
10         if i < 0 andif j < n
11            then OUTPUT(j − m + 1)
12         j ← j + delta[i + 1]
```

Figure 7: The Boyer-Moore algorithm.

phase. When $x[m-1]$ is found at a position j in y, in order to avoid a test $j < n$ each time this symbol is found, the test is postponed before reporting a full match. The exceeding work resulting when $y[n]$ is matched is fully amortized by avoiding the test after all the preceeding matches.

Reverse Factor

The Reverse Factor algorithm [6] and [7] performs the symbol comparisons from right to left and uses the smallest suffix automaton (or dawg) of the reverse word to compute the shifts.

The smallest suffix automaton of a word $w \in \Sigma^*$ is a deterministic finite automaton $\mathcal{A} = (S, s_0, F, \delta)$ where S is a set of states, $s_0 \in S$ is the start state, $F \subseteq S$ is the set of final states and $\delta : S \times \Sigma \to S$ is the transition function. The language accepted by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{u \in \Sigma^* \mid \exists v \in \Sigma^* \text{ such that } w = vu\}$.

The preprocessing phase of the Reverse Factor algorithm consists in computing the smallest suffix automaton for the reverse word x^R. It is linear in time and space in the length of the word.

During each attempt of the searching phase the Reverse Factor algorithm parses the symbols of the text from right to left with the automaton $\mathcal{A}$ starting with state s_0. It goes until there are no more transitions defined for the current symbol from the current state of the automaton. At this point it is easy to know what is the length of the longest prefix of the word which has

```
Reverse-Factor
 1  j ← m − 1
 2  y[n, n + m − 1] = x[m − 1]^m
 3  while j < n
 4      do  while δ(s0, y[j]) = UNDEFINED
 5              do  j ← j + m
 6          state ← δ(s0, y[j])
 7          if state ∈ F
 8              then shift ← m − 1
 9              else shift ← m
10          i ← 1
11          while i < m andif δ(state, y[j − i]) = UNDEFINED
12              do  state ← δ(state, y[j − i])
13                  if state ∈ T
14                      then shift ← m − 1 − i
15                  i ← i + 1
16          if i ≥ m andif j < n
17              then Output(j − m + 1)
18                  shift ← per(x)
19          j ← j + shift
```

Figure 8: The Reverse Factor algorithm.

been matched: it corresponds to the length of the path taken in $\mathcal{A}$ from the start state s_0 to the last final state encountered. Knowing this length it is trivial to compute the right shift to perform.

The Reverse Factor is shown in Figure 8. A fast loop is added to find a transition defined from the start state, this implies to set all the symbols of $y[n, n+m-1]$ to $x[m-1]$. And as in the Boyer-Moore algorithm, when one of this symbol is matched, the test $j < n$ is postponed before reporting a full match.

The Reverse Factor algorithm has a quadratic worst case time complexity but it performs very well in practice (see [3]).

Finite alphabet

Now we consider each symbol of the alphabet Σ as a sequence of p bytes. Searching for a word x of length m in a text y of length n over an infinite alphabet reduces in searching for a word x' of length $m' = pm$ in a text y' of length $n' = pn$ over a finite alphabet Σ'. When an occurrence of x' is found at a position j in y' we still have to check that j is a multiple of p. We are then able to use heuristics on the symbols of the alphabet.

```
SCAN-FOR-FIRST-CHARACTER
1  firstsymb ← x'[0]
2  j ← 0
3  while j ≤ n' − m'
4      do    i ← first index k such that y'[k] = firstsymb in y'[j, n' − 1]
                   or -1 if no such k exists
5      if i = −1
6          then j ← n'
7          else  if y'[j + 1, j + m' − 1] = x'[1, m' − 1] andif j is a multiple of p
8                    then OUTPUT(j)
9                j ← j + 1
```

Figure 9: The Scan for First Character algorithm.

Scan for First Character

The variant presented below is an adaptation to find all occurrences of the word x' in y' of the algorithm called Scan for First Character (SFC) in [8]. It consists in introducing a fast loop with a specific instruction to find the first occurrence of the first symbol of x' in the remaining part of y' and then in checking for a full match.

The SFC algorithm is given in Figure 9.

Boyer-Moore

In the case of a finite alphabet we can use the occurrence shift: it consists after a mismatch to align with the symbol of the text which caused the mismatch a matched symbol in the word or to align the word just after this text symbol if it does not appear in the word.

The values of the occurrence shift are stored in table *delta2* defined as follows:
for each $c \in \Sigma'$

$$delta2[c] = \begin{cases} \min\{j \mid 0 < j < m' - 1 \text{ and } x'[m' - j] = c\} \text{ if } c \text{ appears in } x' \\ m' \text{ otherwise} \end{cases}$$

When a mismatch occurs the Boyer-Moore applies the maximum between the matching shift and the occurrence shift. The fast loop computes the shift with the occurrence shift of the text symbol aligned with the rightmost word symbol. This implies to set all the symbols of $y'[n', n' + m' - 1]$ to $x'[m' - 1]$.

The Boyer-Moore algorithm is given Figure 10. It finds the first occurrence of a non-periodic word of length m' in a text of length n' performing $3n'$ symbol comparisons [13] and it needs an extra space in $O(m' + |\Sigma'|)$.

Horspool

The Horspool algorithm [8] uses only the occurrence shift of the text symbol aligned with the rightmost word symbol to compute the shifts. The Horspool algorithm is depicted Figure 11.

```
Boyer-Moore
1  lastsymb ← x'[m' − 1]
2  y[n', n' + m' − 1] = lastsymb^{m'}
3  shift ← delta[0]
4  j ← m' − 1
5  while j < n'
6     do    while y'[j] ≠ lastsymb
7              do    j ← j + delta2[y'[j]]
8           i ← m' − 2
9           while j ≥ 0 andif y'[j − m' + 1 + i] = x'[i]
10             do    i ← i − 1
11          if i < 0 andif j − m' + 1 is a multiple of p andif j < n'
12             then Output(j − m' + 1)
13                   j ← j + shift
14             else  j ← j + max(delta[i + 1], delta2[y'[j − m' + 1 + i]] − m' + i + 1)
```

Figure 10: The Boyer-Moore algorithm.

It uses a fast loop to detect a match with the last symbol of the word before testing for a full match. This implies to set all the symbols of $y'[n', n' + m' - 1]$ to $x'[m' - 1]$.

The Horspool algorithm has a quadratic worst case time complexity but it can be shown that the average number of comparisons for one text symbol is between $1/|\Sigma'|$ and $2/(|\Sigma'| + 1)$. It needs an extra space in $O(|\Sigma'|)$.

Quick Search

The Quick Search [9] algorithm scans the symbols from left to right and computes the shifts with the occurrence shift of the text symbol just after the text symbol aligned with the rightmost word symbol. It uses a table *delta3* defined as follows:

$$delta3[c] = \begin{cases} \min\{j + 1 \mid 0 < j < m \text{ and } x'[m' - j] = c\} & \text{if } c \text{ appears in } x' \\ m' + 1 & \text{otherwise} \end{cases}$$

It uses a fast loop to detect a match with the last symbol of the word before testing for a full match. This implies to set all the symbols of $y'[n', n' + m' - 1]$ to $x'[m' - 1]$. When a match is detected with $x'[m' - 1]$ it uses a guard to test if there is a match with $x'[0]$ before testing for a full match.

The Quick Search algorithm is shown Figure 12. It has a quadratic worst case time complexity and needs an extra space in $O(|\Sigma'|)$.

Tuned Boyer-Moore

The Tuned Boyer-Moore algorithm [10] is a practical variant of the Boyer-Moore algorithm. It computes the shifts with the occurrence shift of the text symbol aligned with the rightmost word symbol. It uses a table *delta4* defined as follows:

Horspool

```
1  lastsymb ← x'[m' − 1]
2  y[n', n' + m' − 1] = lastsymb^{m'}
3  shift ← delta2[lastsymb]
4  j ← m' − 1
5  while j < n'
6      do    while y'[j] ≠ lastsymb
7                do    j ← delta2[y'[j]]
8            if x'[0, m' − 2] = y'[j − m' + 1, j − 1] andif j − m' + 1 is a multiple of p andif j < n'
9                then Output(j − m' + 1)
10           j ← j + shift
```

Figure 11: The Horspool algorithm.

Quick-Search

```
1  firstsymb ← x'[0]
2  lastsymb ← x'[m' − 1]
2  y[n', n' + m' − 1] = lastsymb^{m'}
2  j ← m' − 1
4  while j < n'
6      do    while y'[j] ≠ lastsymb
7                do    j ← delta3[y'[j + 1]]
5            if y'[j − m' + 1] = firstsymb andif x'[1, m' − 2] = y'[j − m' + 2, j − 1] andif
                     j − m' + 1 is a multiple of p andif j < n'
7                then Output(j − m' + 1)
8            j ← j + delta3[y'[j + 1]]
```

Figure 12: The Quick Search algorithm.

```
TUNED-BOYER-MOORE
1  firstsymb ← x'[0]
2  i ← m' − 2
3  while i ≥ 0 andif x'[i] ≠ x'[m' − 1]
4      do   i ← i − 1
5  shift ← m' − 1 − i
6  lastsymb ← x'[m' − 1]
7  y[n', n' + m' − 1] = lastsymb^{m'}
8  j ← m' − 1
9  while j < n'
10     do   k ← delta4[y'[j]]
11          while k ≠ 0
12              do   j ← j + k
13                   k ← delta4[y'[j]]
14                   j ← j + k
15                   k ← delta4[y'[j]]
16                   j ← j + k
17                   k ← delta4[y'[j]]
18          if y'[j − m' + 1] = firstsymb andif x'[1, m' − 2] = y'[j − m' + 2, j − 1] andif
                j − m' + 1 is a multiple of p andif j < n'
19              then OUTPUT(j − m' + 1)
20          j ← j + shift
```

Figure 13: The Tuned Boyer-Moore algorithm.

$$delta4[c] = \begin{cases} \min\{j \mid 0 < j < m' \text{ and } x'[m'-j] = c\} \text{ if } c \text{ appears in } x' \\ m' \text{ otherwise} \end{cases}$$

such that $delta4[x'[m'-1]] = 0$ which enabled the Tuned Boyer-Moore algorithm to perform three shifts in a row before testing if there is a match with $x'[m'-1]$ (or equivalently to test if the last applied shift was of null length). The Tuned Boyer-Moore stores in a variable *shift* the length of the shift to apply when a match with $x'[m'-1]$ occurred. All this implies to set all the symbols of $y'[n', n'+m'-1]$ to $x'[m'-1]$.

The Tuned Boyer-Moore is presented in Figure 13. It has a quadratic worst case time complexity and needs an extra space in $O(|\Sigma'|)$.

EXPERIMENTS

The algorithms presented above have been implemented in C in a homogeneous way such as to keep their comparison significant. The texts used are composed of 200000 symbols and where randomly built. For each word length, we searched for a hundred words randomly chosen in the texts. The time given in the tables below are the times in seconds for searching a hundred words.

byte	1	2	overall
symbols	256	44	256
minimum	678	0	678
maximum	1564	12500	14064
mean deviation	74.28	1293.78	1366.76

Table 1: Frequency distribution for short integers.

Infinite alphabets

For the Reverse Factor algorithm, in the case of an infinite alphabet, the automaton is implemented as follows:

- the transitions from the start state are stored in a table and accessed via a hashing function;
- the transitions from all the other states are stored in linked lists (it is expected that there will be only one transition defined from these states).

Finite alphabet

For the Reverse Factor algorithm, in the case of a finite alphabet, the automaton is implemented in table of size $O(m' \times |\Sigma'|)$.

The algorithm BM2 is a variant of the BM algorithm where only the matching is used when the last character of the word is matched. The line 13 of the BM algorithm of Figure 10 is replaced by: $j \leftarrow j + delta[i+1]$.

Results

Short integers

Each individual symbol is composed of 2 bytes, then the length of the text when considered over a finite alphabet is 400000. In each position the average frequency value for one symbol is $200000/256 = 781.25$. Overall it is $400000/256 = 1562.5$. The frequency distribution for the different positions is shown in Table 1. It gives:

- the number of different symbols occuring in the text;
- the number of occurrences for the less frequent symbol;
- the number of occurrences for the most frequent symbol;
- the mean deviation.

The results are shown Table 2 and Figure 14. Table 3 shows the average lengths of the shifts for algorithms BM, BM2, HOR, QS and TBM.

The two best algorithms are TBM and QS meaning that in this case, even with a small comparison cost for the actual symbols (short integers on two bytes), it is worth decomposing.

The worst speed-up gained by decomposing is 1.33 for words of length 2, and the best is 7.9 for words of lenth 640.

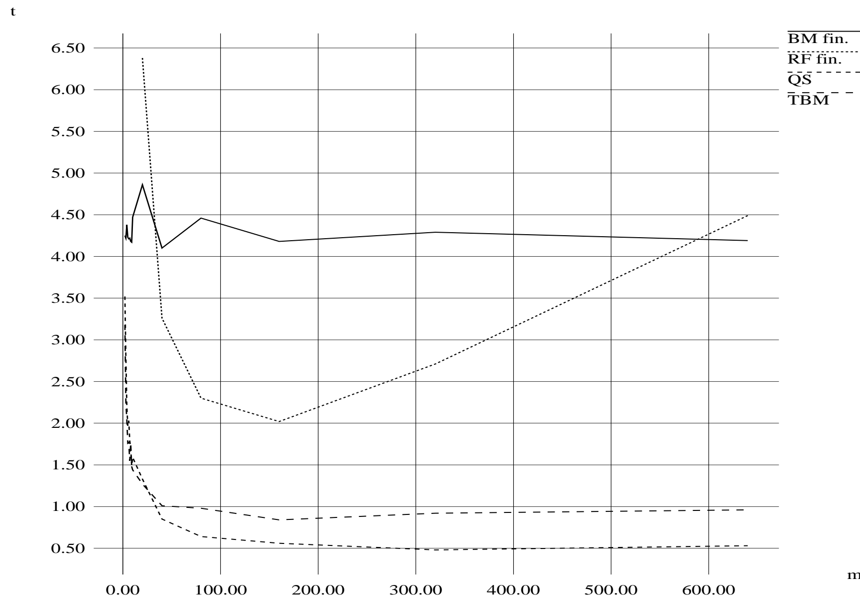

Figure 14: Running times for short integers.

m	BF	BFG	KMP	BM	RF	m'	SFC	BM	BM2	HOR	QS	TBM	RF
2	25.85	7.48	18.34	4.25	63.33	4	6.50	7.27	6.90	4.75	3.52	**3.18**	4.25
3	25.94	7.01	17.93	4.22	44.05	6	6.70	4.34	3.97	3.43	2.79	**2.24**	3.51
4	27.13	7.06	18.94	4.38	32.33	8	6.36	4.22	3.69	2.91	2.20	**2.04**	2.83
5	27.79	7.09	18.03	4.23	25.41	10	6.55	3.28	3.23	2.10	1.97	**1.76**	2.66
6	27.83	7.07	18.18	4.21	20.81	12	6.66	3.08	3.00	2.28	2.03	**1.77**	2.59
7	28.60	7.08	18.05	4.22	17.90	14	6.45	2.72	2.52	1.89	1.79	**1.55**	2.23
8	27.29	6.94	18.18	4.18	15.76	16	6.40	2.46	2.17	1.98	**1.70**	1.76	2.05
9	27.36	7.13	18.07	4.17	13.51	18	6.35	1.86	2.25	1.72	1.49	**1.48**	2.14
10	27.08	7.02	18.21	4.47	12.64	20	6.74	2.03	2.01	1.74	1.59	**1.44**	2.13
20	27.59	6.98	18.00	4.86	6.38	40	7.23	1.78	1.70	1.59	1.33	**1.27**	1.78
40	27.18	7.02	18.06	4.10	3.26	80	7.01	1.17	1.44	1.48	**0.85**	1.01	1.58
80	27.80	7.04	18.16	4.46	2.30	160	6.88	0.86	1.36	1.30	**0.64**	0.98	2.19
160	28.07	6.79	18.33	4.18	2.02	320	7.29	0.89	1.36	1.24	**0.56**	0.84	3.23
320	27.97	7.62	18.22	4.29	2.71	640	6.55	1.08	1.53	1.27	**0.48**	0.92	5.80
640	27.14	7.29	18.34	4.19	4.49	1280	6.79	1.49	1.83	1.19	**0.53**	0.96	10.90

Table 2: Running times for short integers.

m'	BM	BM2	HOR	QS	TBM
4	3.090	3.090	3.000	4.000	3.000
6	5.000	5.000	5.000	6.000	4.980
8	7.050	7.050	7.000	7.970	6.720
10	8.880	8.860	8.850	9.910	8.480
12	10.680	10.670	10.670	11.660	10.080
14	12.330	12.280	12.280	13.140	11.420
16	14.790	14.720	14.690	14.800	12.290
18	16.020	15.820	15.820	16.080	14.890
20	17.670	17.610	17.590	17.440	16.180
40	33.950	32.110	32.050	31.150	28.410
80	61.580	50.730	50.370	54.750	43.940
160	102.440	72.310	71.380	88.430	56.460
320	146.630	90.900	89.890	142.830	68.250
640	189.100	105.020	95.450	222.760	72.310
1280	213.860	115.390	110.300	317.850	85.750

Table 3: Shift lengths for short integers.

byte	1	2	3	4	overall
symbols	17	256	256	256	256
minimum	0	467	688	548	1778
maximum	75121	1170	870	2383	77592
mean deviation	1507.28	257.24	22.92	161.92	1517.17

Table 4: Frequency distribution for floats.

Float

Each individual symbol is composed of 4 bytes, then the length of the text when considered over a finite alphabet is 800000. In each position the average frequency value for one symbol is $200000/256 = 781.25$. Overall it is $800000/256 = 3125$. The frequency distribution for the different positions is shown in Table 4.

The results are shown Table 5. Table 6 shows the average lengths of the shifts for algorithms BM, BM2, HOR, QS and TBM.

TBM is always the faster algorithm meaning that in this case also it is worth decomposing.

The worst speed-up gained by decomposing is 2.61 for words of length 2, and the best is 20.27 for words of length 320.

Double

Each individual symbol is composed of 8 bytes, then the length of the text when considered over a finite alphabet is 1600000. In each position the average frequency value for one symbol is $200000/256 = 781.25$. Overall it is $1600000/256 = 6250$. The frequency distribution for the different positions is shown in Table 7.

The results are shown Table 8. Table 9 shows the average lengths of the shifts for algorithms BM, BM2, HOR, QS and TBM.

The text has a special structure. The single symbol occurring at each seventh byte within

m	BF	BFG	KMP	BM	RF	m'	SFC	BM	BM2	HOR	QS	TBM	RF
2	32.01	11.46	23.11	8.78	97.39	8	26.45	6.97	6.73	4.93	4.67	**3.36**	4.96
3	31.99	10.94	23.14	8.71	69.06	12	25.97	4.91	4.75	3.46	3.69	**2.46**	3.93
4	31.52	11.09	22.76	8.77	52.99	16	25.90	3.90	3.78	3.06	3.33	**2.24**	3.40
5	31.87	11.21	22.78	8.65	43.25	20	25.88	3.40	3.32	2.45	2.99	**2.01**	2.85
6	31.26	11.44	23.25	8.60	35.88	24	25.76	2.95	2.80	2.29	2.64	**1.83**	2.56
7	31.26	11.00	22.69	9.13	31.86	28	25.72	2.60	2.65	2.07	2.55	**1.67**	2.46
8	31.12	11.20	22.67	8.82	28.41	32	26.21	2.47	2.44	1.93	2.68	**1.78**	2.56
9	31.30	11.00	22.96	8.82	25.18	36	25.60	2.26	2.23	1.70	2.32	**1.55**	2.28
10	30.94	11.48	22.83	8.95	22.95	40	25.24	2.09	2.06	1.57	2.22	**1.37**	2.13
20	31.18	11.28	22.37	8.54	13.59	80	25.54	1.28	1.25	0.93	1.72	**0.76**	1.65
40	30.89	11.23	22.59	9.20	8.99	160	25.53	0.93	0.75	0.62	1.49	**0.48**	1.80
80	30.96	11.01	22.30	8.68	6.95	320	25.19	0.81	0.70	0.45	1.19	**0.42**	2.77
160	30.81	10.87	22.27	8.49	6.56	640	25.30	0.75	0.68	0.58	1.23	**0.41**	5.51
320	30.89	10.87	22.31	8.72	10.54	1280	24.96	0.88	0.91	0.44	1.06	**0.43**	10.09
640	30.89	11.20	22.39	8.98	24.28	2560	25.06	1.39	1.47	0.67	1.26	**0.65**	19.48

Table 5: Running times for floats.

m'	BM	BM2	HOR	QS	TBM
2	7.000	7.000	7.000	7.750	7.000
3	11.000	11.000	11.000	10.290	11.000
4	15.000	15.000	15.000	12.510	15.000
5	18.980	18.970	18.970	14.550	18.740
6	22.000	22.000	22.000	16.130	22.000
7	25.990	25.990	25.990	17.570	25.920
8	29.040	29.040	29.040	18.590	29.000
9	32.860	32.860	32.860	20.030	32.590
10	36.020	36.020	36.020	21.000	35.970
20	66.860	66.770	66.720	28.390	66.410
40	115.200	115.050	115.070	36.360	114.640
80	178.320	178.030	178.040	51.810	177.360
160	234.680	234.130	233.820	56.200	232.970
320	266.030	265.740	264.950	71.020	264.130
640	271.370	269.970	268.320	73.410	268.100

Table 6: Shift lengths for floats.

byte	1	2	3	4	5	6	7	8	overall
symbols	5	215	256	256	256	4	1	1	256
minimum	0	0	704	692	174	0	0	0	1635
maximum	100152	6371	863	1344	9667	124889	200000	200000	536705
mean deviation	1549.95	1076.77	21.34	40.28	488.26	1537.84	1556.14	1556.14	6530.00

Table 7: Frequency distribution for doubles.

m	BF	BFG	KMP	BM	RF	m'	SFC	BM	BM2	HOR	QS	TBM	RF
2	32.53	11.55	23.28	**8.46**	70.35	16	47.83	20.27	40.48	25.96	16.94	25.08	25.01
3	31.79	11.46	22.45	**8.53**	49.77	24	48.12	8.56	38.31	25.75	16.81	24.43	17.51
4	31.68	10.82	23.39	8.27	38.11	32	47.59	**7.98**	36.62	25.95	16.56	24.23	13.62
5	32.27	10.75	22.72	8.44	32.90	40	46.89	**7.16**	35.97	25.42	16.00	23.89	11.10
6	32.16	10.80	23.21	8.41	25.02	48	46.52	**7.14**	35.09	25.34	16.24	23.68	9.22
7	32.64	10.86	22.85	8.48	22.29	56	45.93	**7.06**	33.17	25.76	16.03	23.64	8.22
8	31.62	10.67	23.02	8.49	19.72	64	45.59	**5.29**	33.27	25.17	16.15	23.77	7.32
9	31.86	10.95	22.79	8.60	17.51	72	45.74	**4.97**	33.77	26.44	16.69	24.58	6.59
10	32.03	10.64	23.24	8.48	16.07	80	46.08	**5.48**	33.27	25.24	15.78	23.40	6.23
20	31.71	10.60	23.26	8.37	8.70	160	45.78	**3.59**	31.50	23.83	15.08	22.48	4.21
40	32.49	11.06	22.38	8.50	5.48	320	45.88	**2.85**	27.63	23.34	14.95	22.07	3.88
80	31.24	10.72	22.60	8.37	4.34	640	46.02	**2.83**	26.85	25.51	16.51	23.35	5.74
160	31.96	10.96	22.51	8.35	4.02	1280	45.89	**2.78**	24.78	23.35	15.11	21.72	10.54
320	31.47	10.61	22.62	8.34	5.98	2560	45.90	**3.15**	24.81	24.48	15.59	23.13	20.30
640	31.29	10.62	22.56	8.47	16.70	5120	46.30	**4.18**	22.56	24.90	16.67	23.98	39.48

Table 8: Running times for doubles.

an actual symbol is the same that the one occurring at each eighth position (see Table 7) which explained that the use of the occurrence shift gives poor results for the algorithms HOR, QS and TBM (see Table 9). Thus the BM algorithm is always the best one. But for very small word (length 2 and 3) it is more efficient to perform the search with the actual alphabet and for longer words it is better to decompose.

The best speed-up gained by decomposing is 2.33 for words of length 20.

Structure

Each individual symbol is composed of 32 bytes as follows: a short integer on 2 bytes, a float on 4 bytes, a double on 8 bytes and a string on 16 bytes. When two symbols are compared, the comparisons is done component by component, beginning with the short integer, then the float, then the double and finally the string. The length of the text when considered over a finite alphabet is 6400000. In each position the average frequency value for one symbol is $200000/256 = 781.25$. Overall it is $6400000/256 = 25000$. The frequency distribution for the different positions is shown in Table 10.

The results are shown Table 11. Table 12 shows the average lengths of the shifts for algorithms BM, BM2, HOR, QS and TBM.

The best results are always reached by the TBM algorithm meaning that even when the text has a special structure on some positions (as shown by Table 10) if there are enough positions with a "good" distribution, the use of heuristic on the alphabet is efficient.

The worst speed-up gained by decomposing is 1.28 for words of length 2, and the best is 2.92 for words of length 20.

m'	BM	BM2	HOR	QS	TBM
2	14.880	10.610	6.680	10.710	4.080
3	21.680	11.340	7.250	12.850	4.100
4	28.120	12.300	7.180	14.060	3.600
5	34.570	12.240	7.390	14.770	3.620
6	39.280	12.440	7.790	14.360	3.990
7	44.240	13.250	7.780	14.520	3.990
8	51.090	13.740	7.830	14.180	4.080
9	56.410	13.440	7.430	13.920	3.700
10	59.800	13.940	8.170	14.740	4.330
20	99.010	15.910	10.540	16.080	6.110
40	148.930	23.070	9.950	16.950	5.330
80	195.240	21.320	11.250	15.520	6.460
160	232.790	28.370	12.480	20.340	7.240
320	239.610	27.240	11.120	18.930	6.300
640	247.680	31.660	12.260	18.470	7.170

Table 9: Shift lengths for doubles.

byte	1 to 18	19	20	21	22	23	24	25
symbols	44	256	44	19	256	256	256	5
minimum	0	678	0	0	479	674	539	0
maximum	12500	1564	12500	75272	1123	862	2437	100112
mean deviation	1293.78	74.33	1293.78	1507.29	259.58	23.71	162.08	1549.89

byte	26	27	28	29	30	31	32	overall
symbols	226	256	256	256	4	1	1	256
minimum	0	708	683	163	0	0	0	4180
maximum	6409	856	1306	9645	125080	200000	200000	779742
mean deviation	1073.89	22.68	42.27	486.96	1537.84	1556.14	1556.14	30430.62

Table 10: Frequency distribution for structures.

m	BF	BFG	KMP	BM	RF	m'	SFC	BM	BM2	HOR	QS	TBM	RF
2	37.91	14.38	25.78	12.68	116.96	64	135.00	25.28	33.71	12.38	14.10	**9.90**	26.83
3	37.61	14.66	26.10	12.87	82.50	96	134.72	15.99	30.83	10.69	10.91	**8.46**	18.93
4	37.15	14.85	26.04	12.48	62.09	128	134.77	13.98	30.54	8.92	9.88	**7.23**	14.93
5	36.93	14.81	25.64	12.60	50.12	160	134.70	13.12	29.05	8.29	8.83	**6.54**	12.55
6	37.50	14.52	26.00	12.44	42.51	192	134.70	11.87	26.57	7.66	8.08	**6.07**	11.10
7	37.38	14.35	25.96	13.01	36.46	224	135.00	10.25	23.99	7.31	7.60	**5.79**	10.12
8	37.82	14.63	26.13	12.62	32.96	256	135.51	10.79	24.21	6.85	7.45	**5.30**	9.66
9	36.62	14.25	25.47	12.77	29.46	288	135.09	9.40	24.33	6.70	6.98	**5.23**	9.34
10	36.53	14.38	25.32	12.27	26.50	320	135.57	8.70	23.59	6.29	6.71	**5.12**	8.91
20	36.57	14.29	25.38	12.21	14.80	640	135.32	6.90	24.62	5.30	5.19	**4.17**	9.52
40	37.22	14.29	25.44	12.53	9.86	1280	134.95	6.14	23.20	4.76	4.29	**3.89**	14.76
80	36.27	13.77	25.07	12.31	7.60	2560	135.31	6.26	21.07	4.56	3.94	**3.60**	27.20
160	36.06	13.98	24.68	11.98	8.08	5120	135.08	7.01	19.62	4.72	4.08	**3.72**	50.47
320	35.72	13.90	25.04	12.42	13.16	10240	136.08	9.06	22.37	5.19	4.70	**4.59**	94.00
640	36.45	13.78	25.07	12.44	29.44	20480	137.21	13.07	24.76	6.61	5.86	**5.55**	194.58

Table 11: Running times for structures.

m'	BM	BM2	HOR	QS	TBM
64	53.320	46.310	44.120	35.990	40.820
96	65.810	51.080	55.370	49.800	49.300
128	80.160	55.390	67.880	55.510	61.080
160	91.220	58.660	75.380	63.110	66.990
192	104.030	64.540	82.800	69.250	73.200
224	114.640	70.080	90.110	75.210	79.620
256	122.640	71.900	97.470	79.370	85.960
288	131.820	73.410	101.610	84.740	89.260
320	141.020	76.800	107.070	88.790	94.070
640	196.600	86.810	140.020	122.570	120.800
1280	243.700	99.660	167.100	162.620	143.000
2560	281.020	116.300	188.820	193.100	160.730
5120	302.470	133.560	195.790	204.970	165.830
10240	302.390	124.610	193.460	204.050	164.020
20480	302.690	128.160	189.740	208.750	161.000

Table 12: Shift lengths for structures.

CONCLUSION

Infinite alphabet

In this case BF, BFG, KMP and BM algorithms run in a time independent from the word length. Only RF algorithm runs faster with longer words up to length of approximately 200. For longer words the cost of the construction the suffix automaton becomes too expensive.

BFG is approximately 3 times faster than BF, and 2 times faster than KMP. BM algorithm is the most efficient algorithm except for word lengths between approximately 40 and 300 or 400 (depending on the alphabet) where RF is better.

Finite alphabet

The BM2 algorithm is sometimes better than the BM algorithm except in the case of the doubles.

For the TBM there are only slight differences between doing two, three or four consecutive shifts in the fast loop, except for the doubles where doing only two shifts is always better but still slower than the BM algorithm. We present the results for the algorithm suggested in [10] with three consecutive shifts.

General conclusion

References

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

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

[3] T. Lecroq. Experimental results on string matching algorithms. *Software–Practice and Experience*, 25(7):727–765, 1995.

[4] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

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

[6] T. Lecroq. A variation on the Boyer-Moore algorithm. *Theoret. Comput. Sci.*, 92(1):119–144, 1992.

[7] 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.

[8] R. N. Horspool. Practical fast searching in strings. *Software–Practice and Experience*, 10:501–506, 1980.

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

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

[11] T. Raita. Tuning the Boyer-Moore-Horspool string searching algorithm. *Software–Practice and Experience*, 22(10):879–884, 1992.

[12] P. D. Smith. On tuning the Boyer-Moore-Horspool string searching algorithms. *Software–Practice and Experience*, 24(4):435–436, 1994.

[13] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. *SIAM J. Comput.*, 23(5):1075–1091, 1994.