Experiments on string matching in memory structures

Thierry Lecroq
LIR (Laboratoire d'Informatique de Rouen) and
ABISS (Atelier de Biologie Informatique Statistique et Socio-Linguistique),
Université de Rouen, Faculté des Sciences et des Techniques, 76128 Mont-Saint-Aignan Cedex, France,
(email: Thierry.Lecroq@dir.univ-rouen.fr)

Abstract

Various string matching algorithms have been designed and some experimental works on string matching over bounded alphabets have been performed but string matching over unbounded 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 bounded and use heuristics on symbols.

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

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 which assumed that the alphabet is bounded 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 shift length for each symbol.

When a short sequence of numbers has to be searched in a larger one or when a sequence of records has to be found in a flat file then the underlying alphabet is unbounded. Also with the new Unicode Standard [4], the characters are encoded on 16 bits which could enable to have 65536 different characters. No study had been done in practice when searching over an unbounded alphabet. Obviously this case can reduce to a search over a bounded alphabet (namely bytes) 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 symbols (and using algorithms over unbounded alphabets which are known to be slower) or decomposing the word and the text into bytes (and using algorithms over bounded

alphabets but lengthening the word and the text) ? This study tries to answer this question from a practival viewpoint.

We performed experiments with different alphabets: short integers on two bytes, real numbers on four bytes, real numbers on eight bytes and a structure on 32 bytes. For each alphabet, we performed various searches with string matching algorithms over unbounded alphabets (Boyer-Moore algorithm [5] with only the matching shift and Reverse Factor algorithm [6] and [7]) and with string matching algorithms over bounded alphabets (Boyer-Moore algorithm, Horspool algorithm [8], Quick Search algorithm [9], Tuned Boyer-Moore algorithm [10]).

We first recall briefly the algorithms before giving the results of the different experiments.

THE ALGORITHMS

Exact string matching consists in finding all the occurrences of a word $x = x[0, \ldots, m-1]$ of length m in a text $y = y[0, \ldots, n-1]$ 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 uses a window on the text, this window has the same size than the word, it then checks if there is a match between the word and the text symbols in the window (this specific work is called an *attempt*) then it *shifts* the window and the word to the right and repeats the same process again until the right end of the word goes beyond the right end of the text.

The various string matching algorithms 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 [11], 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.

All the algorithms that have been tested are variants of the Boyer-Moore algorithm. Six algorithms have been tested: two for the unbounded case (the Boyer-Moore algorithm (BM) [5] and the Reverse Factor algorithm (RF) [6] and [7]) and four for the bounded case (the Boyer-Moore algorithm, the Horspool algorithm (HOR) [8], the Quick Search algorithm (QS) [9] and the Tuned Boyer-Moore algorithm (TBM) [10]).

The five first algorithms have already been presented in [3]. We will only emphasize here on the improvement made, if any.

We modified all the original algorithms to add a fast loop (*ufast* according to the terminology

of [10]) which consists in checking if the rightmost symbol of the window leeds to a match before checking the remaining symbols of the window. This implies to add some virtual symbols at the end of the text: $y[n, \ldots, n+m-1] = (x[m-1])^m$.

Unbounded alphabets

In the case where the alphabet is unbounded, a preprocessing phase on the word is allowed but no information can be stored for each individual symbol using a reasonnable size. A compact structure to store an information only for the symbols which appear in the word x, in $O(m)$ space complexity, would be too slow to access.

Boyer-Moore

In this case the Boyer-Moore algorithm can only use the matching shift. Indeed the occurrence shift require to store a shift length for all the symbol of the alphabet which is not possible.

Reverse Factor

In this case the suffix automaton of the reverse of the word x is stored in the following way:

- the transitions from the start state are stored in a table and accessed via a hashing function; This function is very simple and consists in taking the integer value of the symbol (or a part of the symbol in the case of structures) to multiply it by a prime number and then taking the modulus by the size of the table. We use open hashing so the collisions are handled by using linked lists. Experimentally the best results were obtained with a table of size $3m$.
- 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).

Bounded alphabets

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 unbounded alphabet reduces in searching for a word x' of length $m' = pm$ in a text y' of length $n' = pn$ over a bounded alphabet Σ'. When an occurrence of x' is found at a position j in y' we still have to check if j is a multiple of p.

The solution which consists in making shifts always of length which are multiples of p and then avoid to check j does not yield to fast running times. This is due to the fact that the shifts would not be significantly longer and the cost of their adjustment would not be amortized by the fact that the offset is not checked. Another aspect which is more difficult to analyse is that, in general, due to the byte distribution, the length of the shifts are probably greater when the window is not aligned with positions that are not multiples of p.

In this case we are able to use heuristics on the alphabet Σ'.

Boyer-Moore

In the case of a bounded alphabet the two shift functions (the occurrence and the matching shift) can be used.

Quick Search

Our version of the Quick Search [9] algorithm uses Raita's trick [12] which consists in saving the first symbol of the word in a variable and checking if the leftmost symbol of the window matches this variable before checking the remaining unchecked symbols of the window.

Tuned Boyer-Moore

The Tuned Boyer-Moore algorithm [10] is a practical variant of the Boyer-Moore algorithm. It only uses the occurrence shift. Its fast loop consists in appplying three shifts in a row (implying that the shift for $x[m-1]$ is equal to 0) before checking for a full match. It also store in a variable the minimal shift to apply when $x[m-1]$ has been matched in the rightmost position of the window. Our version also uses Raita's trick.

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 were randomly built. There is then a dependancy between the nature of the actual symbols and the distribution of bytes. For each word length, we searched for hundred words randomly chosen in the texts (these words were found at most seven times in the text and at least one). Two approaches were made: the first one consists in dealing with the actual symbols and the second one in decomposing the actual symbols into sequences of bytes. The time given in the tables below are the times in seconds for searching hundred words. These times include the preprocessing times which are shown in parentheses.

The target machine is a Hyunday SPARC HWS-S310 running SunOS 4.1.3. The compiler is `cc`.

More complete results can be found in Reference [13].

Results

Short integers on two bytes

The text is composed of 200000 integers, represented on two bytes, ranging from -32767 to 32761. Each individual symbol is composed of 2 bytes, then the length of the text when considered over a bounded alphabet is 400000 bytes. In each position the average frequency value for one symbol is $200000/256 = 781.25$. Overall it is $400000/256 = 1562.5$. The frequency

byte	1	2	overall
symbols	256	44	256
minimum	678	0	678
maximum	1564	12500	14064
$1/q$	248.25	32.00	86.01

Table 1: Frequency distribution for short integers.

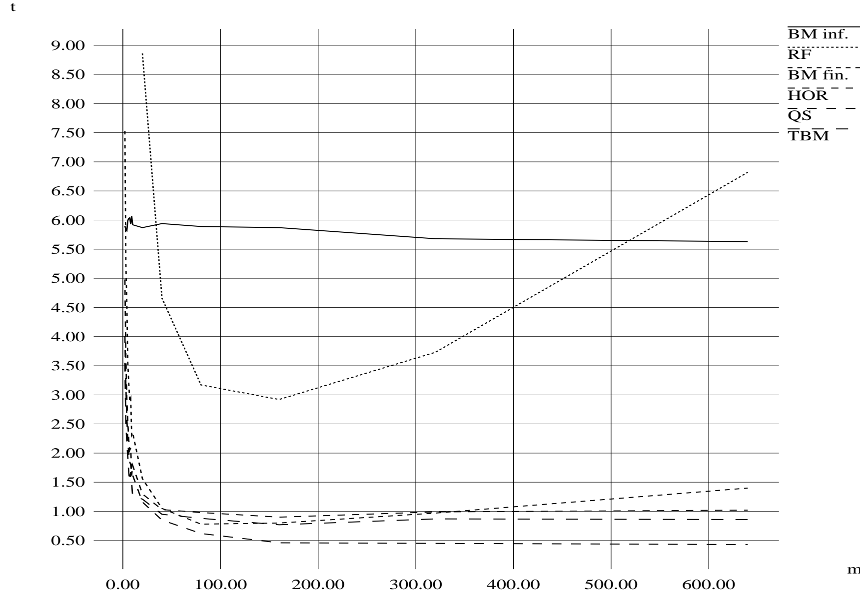

Figure 1: Running times for short integers.

distribution for the different positions is shown in Table 1. For each byte, which are numbered from left to right (the most significant byte on the left), it gives:

- the number of different symbols occuring in the text;
- the number of occurrences for the least frequent symbol;
- the number of occurrences for the most frequent symbol;
- $1/q$ where q denotes the probability $\sum_{c\in\Sigma} p^2(c)$ that two randomly chosen characters match. $0 \leq p(c) \leq 1$ denotes the probabiblity for symbol c with $\sum_{c\in\Sigma} p(c) = 1$.

The results are shown Table 2 and Figure 1.

m	BM	RF	m'	BM	HOR	QS	TBM
2	5.90 (0.00)	79.62 (0.01)	4	7.53 (0.00)	4.98 (0.00)	4.01 (0.01)	**3.55** (0.00)
3	5.84 (0.00)	56.86 (0.01)	6	5.25 (0.01)	3.64 (0.01)	3.15 (0.01)	**2.51** (0.01)
4	5.82 (0.01)	43.27 (0.03)	8	4.57 (0.01)	2.93 (0.00)	2.61 (0.00)	**2.21** (0.03)
5	5.99 (0.00)	34.08 (0.01)	10	3.55 (0.01)	2.53 (0.00)	2.20 (0.00)	**1.93** (0.00)
6	6.03 (0.00)	28.15 (0.03)	12	3.28 (0.03)	2.22 (0.01)	1.99 (0.01)	**1.67** (0.00)
7	6.04 (0.00)	24.02 (0.01)	14	2.89 (0.03)	2.05 (0.00)	1.85 (0.01)	**1.52** (0.00)
8	5.93 (0.01)	22.01 (0.03)	16	2.94 (0.02)	2.09 (0.00)	1.80 (0.01)	**1.67** (0.00)
9	6.07 (0.00)	19.06 (0.02)	18	2.31 (0.02)	1.62 (0.01)	1.65 (0.00)	**1.48** (0.00)
10	5.92 (0.00)	17.87 (0.02)	20	2.35 (0.04)	1.79 (0.00)	1.62 (0.00)	**1.26** (0.02)
20	5.87 (0.01)	8.86 (0.06)	40	1.56 (0.01)	1.30 (0.01)	**1.16** (0.02)	1.21 (0.02)
40	5.94 (0.01)	4.66 (0.20)	80	1.05 (0.01)	1.03 (0.00)	**0.85** (0.01)	0.95 (0.01)
80	5.89 (0.00)	3.17 (0.16)	160	0.78 (0.03)	0.98 (0.03)	**0.62** (0.01)	0.88 (0.01)
160	5.87 (0.01)	2.92 (0.55)	320	0.80 (0.07)	0.90 (0.03)	**0.46** (0.02)	0.77 (0.03)
320	5.68 (0.09)	3.73 (0.88)	640	0.97 (0.21)	0.99 (0.02)	**0.45** (0.02)	0.87 (0.04)
640	5.63 (0.08)	6.82 (1.80)	1280	1.40 (0.24)	1.02 (0.06)	**0.43** (0.07)	0.86 (0.04)

Table 2: Running times for short integers (preprocessing times are in parentheses).

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 1.66 for words of length 2, and the best is 13.09 for words of lenth 640.

Real numbers on 4 bytes

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 $800000/256 = 3125$.

TBM is always the faster algorithm meaning that in this case also it is worth decomposing. See Reference [13] for complete results.

Real numbers on 8 bytes

Each individual symbol is composed of 8 bytes, then the length of the text when considered over a bounded 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 3.

The results are shown Table 4.

The text has a special structure. The single symbol occurring at each seventh byte within an actual symbol is the same that the one occurring at each eighth position (see Table 3) which explained that the use of the occurrence shift gives poor results for the algorithms HOR, QS and TBM. Thus the BM algorithm is always the best one. But for very small word (length 2

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
$1/q$	2.00	48.07	255.71	254.40	113.98	2.28	1.00	1.00	8.10

Table 3: Frequency distribution for real numbers on 8 bytes.

m	BM	RF	m'	BM	HOR	QS	TBM
2	**8.51** (0.00)	86.00 (0.00)	16	21.30 (0.01)	32.75 (0.01)	21.24 (0.01)	29.68 (0.00)
3	**8.62** (0.00)	62.59 (0.00)	24	8.99 (0.00)	32.90 (0.00)	22.48 (0.02)	28.79 (0.00)
4	8.49 (0.00)	48.70 (0.01)	32	**8.18** (0.05)	34.01 (0.01)	19.95 (0.03)	30.11 (0.01)
5	8.50 (0.00)	41.89 (0.02)	40	**7.27** (0.01)	34.42 (0.00)	19.62 (0.00)	28.88 (0.01)
6	8.62 (0.00)	32.96 (0.01)	48	**8.00** (0.02)	34.40 (0.01)	22.57 (0.01)	29.30 (0.03)
7	8.73 (0.00)	28.61 (0.03)	56	**7.85** (0.04)	34.59 (0.02)	21.14 (0.02)	30.05 (0.00)
8	8.52 (0.00)	25.33 (0.01)	64	**6.54** (0.07)	32.22 (0.01)	21.92 (0.03)	30.65 (0.04)
9	8.56 (0.00)	22.95 (0.02)	72	**5.94** (0.04)	34.12 (0.01)	23.64 (0.00)	30.39 (0.01)
10	8.59 (0.00)	20.61 (0.04)	80	**6.83** (0.01)	32.04 (0.01)	20.83 (0.02)	29.08 (0.01)
20	8.42 (0.00)	10.72 (0.06)	160	**4.34** (0.02)	32.79 (0.01)	18.73 (0.01)	27.94 (0.02)
40	8.46 (0.00)	6.49 (0.15)	320	**3.50** (0.11)	30.92 (0.02)	19.18 (0.02)	27.15 (0.03)
80	8.45 (0.00)	4.87 (0.31)	640	**3.11** (0.15)	34.78 (0.02)	21.08 (0.02)	29.00 (0.04)
160	8.58 (0.00)	5.02 (1.10)	1280	**3.18** (0.33)	31.73 (0.06)	19.54 (0.08)	28.86 (0.08)
320	8.56 (0.00)	6.97 (3.04)	2560	**3.60** (0.68)	33.27 (0.07)	19.96 (0.04)	29.33 (0.07)
640	8.71 (0.00)	18.66 (13.45)	5120	**4.42** (1.23)	33.01 (0.20)	19.99 (0.22)	30.54 (0.17)

Table 4: Running times for doubles (preprocessing times are in parentheses).

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 1.97 for words of length 640.

Structures

Each individual symbol is composed of 32 bytes as follows: a short integer on 2 bytes, a real number on 4 bytes, a real number on 8 bytes and a string on 18 bytes. When two symbols are compared, the comparisons is done component by component, beginning with the short integer, then the real number and finally the string. The length of the text when considered over a bounded 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 best results are always reached by the TBM algorithm meaning that even when the text has a special structure on some positions if there are enough positions with a "good" distribution, the use of heuristic on the alphabet is efficient. See Reference [13] for complete results.

CONCLUSION

Unbounded alphabet

In this case BM algorithm run in a time independent from the word length. This is due to the fact that for almost all the attempts the rightmost symbol of the window leads to a mismatch thus yielding to a shift of length one. RF algorithm runs faster with long words up to length of approximately 200. For longer words the cost of the construction of the suffix automaton becomes too expensive (the preprocessing time becomes too large with respect to the searching time).

Bounded alphabet

For the TBM there are only slight differences between doing two, three or four consecutive shifts in the fast loop, except for the real numbers on 8 bytes where doing only two shifts is always better but still slower than the BM algorithm. We presented here the results for the algorithm suggested in [10] with three consecutive shifts.

In some cases the QS algorithm is faster than the HOR algorithm though the latter only fetches the rightmost symbol of the window while the QS algorithm also fetches the following symbol to compute the shift. This is due to the byte distribution, in some cases the shifts are longer when computed with the rightmost symbol of the window and in other cases they are longer when computed with the symbol immediately the the right of the window.

General conclusion

We have performed experiments of string matching algorithms with four different kinds of unbounded alphabets. In almost all cases, but for very small words on a very special byte frequency distribution, it is more efficient to decompose symbols of a unbounded alphabet into sequences of bytes. Doing this, we have to search for a longer word (in term of number of symbols) in a longer text but we can use heuristics on the alphabet which is bounded. This enables to save a lot of byte comparisons and therefore to save time. Thus the assumption that, in practice, we always can reduce to a bounded alphabet seems well founded.

The source codes and data sets used to produce the results given in this article are available at the following URL: `http://www.dir.univ-rouen.fr/~lecroq/esmms.tar.gz`.

Acknowledgements

I thank the anonymous referees for their useful comments.

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] *The Unicode Standard, Version 2.0.* Addison-Wesley, 1996.

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

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

[13] T. Lecroq. Experimental results on string matching over infinite alphabets. Technical Report LIR 97.01, Université de Rouen, France, 1997.