Regular Paper

Max-Shift BM and Max-Shift Horspool: Practical Fast Exact String Matching Algorithms

Mohammed Sahli[1,a) Tetsuo Shibuya[2]

Received: September 8, 2011, Accepted: January 13, 2012

Abstract: Exact string matching is the problem of finding all occurrences of a pattern P in a text T. The problem is well-known and many sophisticated algorithms have been proposed. Some fast exact string matching algorithms have been described since the 80 s (e.g., the Boyer-Moore algorithm and its simplified version the Boyer-Moore-Horspool algorithm). They have been regarded as the standard benchmarks for the practical exact string search literature. In this paper, we propose two algorithms MSBM (Max-Shift BM) and MSH (Max-Shift BMH) both based on the combination of the bad-character rule of the right-most character used in the Boyer-Moore-Horspool algorithm, the extended bad-character rule and the good-suffix rule used in the Gusfield algorithm, which is a modification of the Boyer-Moore algorithm. Only a small extra space and preprocessing time are needed with respect to the BM and BMH algorithms. Nonetheless, empirical results on different data (DNA, Protein and Text Web) with different pattern lengths show that both MSBM and MSH are very fast in practice. MSBM algorithm usually won against other algorithms.

Keywords: exact string matching, information retrieval, Boyer-Moore algorithm, linear time, sublinear time

1. Introduction

Exact string matching problem consists in finding all occurrences of a string P (generally called Pattern) of length m within a larger string T (called Text) of length n. There are different ways to scan the text T in order to find occurrences of P. According to the survey of Charras and Lecroq [11], we can classify exact string-matching algorithms into four categories *From Left-to-Right*, *From Right-to-Left*, *In a specific order* and *In any order*. However, the last category (i.e., In any order) can also be considered as a subcategory of the second one (i.e., From Right-to-Left). We redescribe and summarize these categories as follows:

Forward Orientation (Left-to-Right): One way is to scan the text by moving forward and comparing characters of the pattern P to characters of the text T. Several algorithms use this idea such as: the Brute Force algorithm, Knuth-Morris-Pratt algorithm [12] which was the first linear-time algorithm that tackles this problem, the Apostolico-Crochemore algorithm [16] and the Forward Dawg Matching algorithm [17] that builds a suffix automaton of the pattern as a scanning strategy.

Backward Orientation (Right-to-Left): The idea is very simple: instead of comparing characters by moving from left to right (Forward Orientation), we can compare them starting from the opposite side (Right-to-Left). This category has led to creating very sophisticated and fast algorithms after the publication of the algorithm of Boyer and Moore in 1977 [8]. The original Boyer-Moore algorithm uses two shift functions: the bad-character rule and the good-suffix rule. Since that time, several variants of this algorithm and simplifications have been introduced such as Horspool algorithm [7], Tuned Boyer-Moore of Hume and Sunday, Quick Search by Sunday [5], [6], Turbo-BM by Crochemore et al. [3], [5], the Smith algorithm [4] and Raita algorithm [13]. Please refer to [11] for further information.

Specific Orientation: Such as partitioning the pattern into two parts as shown in the algorithms of Colussi [18] and Galil-Giancarlo [19]. The partitioning is done by first searching for the right part of the pattern from left to right and then if no mismatch occurs the search is done for the left part. There is also another strategy in which the pattern character positions are sorted according to their frequency and their leading shift respectively [11].

In this paper, we introduce new variants of BM algorithm which have not been reported previously. In order to accelerate the search, another rule called the *extended bad-character rule* [14] was added in addition to the *bad-character rule* and the *good-suffix rule* [8]. By taking the maximum of these three rules, we created new variants of BM and Horspool algorithms (we call them MSBM and MSH algorithms respectively). The extended bad-character rule has not been evaluated rigorously in the literature (according to our knowledge). The fact that we chose the extended bad-character rule is because it allows larger shifts than the normal bad-character rule especially when we deal with small alphabet size such as DNA and protein. Yet, relying only on the extended bad-character rule will not lead to an important enhancement [14]. Experiments showed that incorporating these rules all together gives a significant improvement upon previous algorithms.

[1] Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–0033, Japan.
[2] Human Genome Center, Institute of Medical Science, The University of Tokyo, Minato, Tokyo 108–8639, Japan.
[a)] mohammed@hgc.jp

The rest of this paper is organized as follows. Section 2 shows a brief description of Boyer-Moore algorithm and related work is given in Section 3. Section 4 describes our contribution by introducing the Multi-Shift strategy and illustrating it by a small example. In Section 5, two new variants of Boyer-Moore algorithm and Horspool algorithm respectively are given respectively by applying the Max-Shift rule. Then, Experimental results and comparison are shown in Section 6.

In this paper, we consider the following terms :

- T: refers to the text.
- P: refers to the pattern,
- $n = |T|$ size of the text,
- $m = |P|$ size of the pattern.
- $T[i]$: is the i-th character in the text T.
- $T[i \dots j]$: is the substring of the text T starting from the i-th character and ending at j-th character.
- S_i or $S[i \dots m-1]$: is the i-th pattern suffix.
- P_i or $P[0 \dots i]$: is the i-th pattern prefix.

All indices start at position 0 and end at position $n-1$ (of T) or $m-1$ (of P).

2. Brief Description of Boyer-Moore Algorithm

The Boyer-Moore algorithm has been the standard benchmark for the practical string search literature and the most efficient string-matching algorithm in usual applications [5]. The characters of the pattern are scanned by the algorithm from right to left beginning with the rightmost one. When there is a match of the whole pattern or a mismatch, it uses two pre-computed functions to shift the window to the right. These two shift functions are called the good-suffix rule and the bad-character rule: "Assume that a mismatch occurs between the character $P[i] = a$ of the pattern and the character $T[i+j] = b$ of the text during an attempt at position j. Then, $P[i+1 \dots m-1] = T[i+j+1 \dots j+m-1]$ and $P[i] \neq T[i+j]$. The bad-character shift consists in aligning the text character $T[i+j]$ with its rightmost occurrence in $P[0 \dots m-2]$. If $T[i+j]$ does not occur in the pattern P, no occurrence of P in T can include $T[i+j]$, and the left end of the window is aligned with the character immediately after $T[i+j]$, namely $T[i+j+1]$. The good-suffix shift consists in aligning the segment $T[i+j+1 \dots j+m-1] = P[i+1 \dots m-1]$ with its rightmost occurrence in P which is preceded by a character different from $P[i]$. If there exists no such segment, the shift consists in aligning the longest suffix of $T[i+j+1 \dots j+m-1]$ with a matching prefix of P. Note that the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the good-suffix shift and bad-character shift" as explained in Ref. [8].

3. Related Work

Since Quick Search algorithm [6], [11] scans the text with the bad-character rule; it is slightly similar to the Horspool algorithm whereas the only difference is the choice of rightmost character. Horspool [7] uses the text character (say $T[k+m-1]$) that corresponds to the rightmost character of the pattern (say $P[m-1]$) for calculation. Sunday [6] noticed that the text character $T[k+m]$ just next to the rightmost text character can be involved too and then it could become useful for the bad-character shift. Another improvement was done by Smith [4] when he noticed that Quick Search algorithm gives sometimes shorter shift than that of the rightmost text character, he suggested the maximum of the two shifts.

The preprocessing algorithm for the bad-character algorithm runs in $O(m)$ time and requires $O(|\Sigma|)$ space [7], [8], [11]. The extended bad-character rule was mentioned in Ref. [14] by Gusfield. He discussed the matter of using the extended bad-character rule in the Boyer-Moore algorithm. In our paper, however, for creating our algorithm MSH we are combining the benefit of two rules: the extended bad-character and the simple bad-character of the rightmost character of the pattern (as in Horspool algorithm). In addition, the good-suffix rule will be used with latter rules in order to create our second algorithm MSBM. Several experiments and comparisons have been done for exact string algorithms, for further information on the subject we refer to Refs. [1], [2], [5], [11].

4. The Multi-Shift Algorithm

4.1 The Extended Bad-character Rule

The extended bad-character rule can be defined as follows: "When a mismatch occurs at position i of P and the mismatched character in T is a, then shift P to the right so that the closest a to the left of position i in P is below the mismatched a in T" [14].

4.2 Description of the Overall Algorithm

Boyer and Moore proposed to use two shift functions: the good-suffix rule and the bad-character rule [8], whereas Horspool suggested a simplified variant by using only the bad-character rule [7]. In the Horspool algorithm, computing shifts by the rightmost character of the pattern led to an efficient algorithm in practice. However, relying on the rightmost character of the pattern alone does not yield larger shifts in all cases especially when we deal with small alphabet size such as DNA or Protein data. For instance, consider the case when a mismatch occurs between the character $P[i]$ of the pattern and the character $T[k+i]$ of the text during an iteration k. Thus, suffix $P[i+1 \dots m-1]$ of the pattern and factor $T[k+i+1 \dots k+m-1]$ of the text are matched. Let $d_{m-1} = \mathit{BadShift}(T[k+m-1])$ and $d_i = \mathit{BadShift}(T[k+i])$, that is, the bad-character rule of the rightmost character $P[m-1] = T[k+m-1]$ and that of the i-th position of P where the mismatch occurred respectively. There are two cases here: if $d_i > d_{m-1}$ we shift by d_i, otherwise we shift by d_{m-1}. To sum up, we shift by $max(d_{m-1}, d_i)$.

Note: Readers should notice that d_i refers to the shift by the rightmost character of the prefix $P[0 \dots i]$ (i.e., $P[i]$).

4.3 Example

Consider the text Y = GCATCGCGGAGAGTATACAGTACG and the pattern X = GCGGAGAG. The example was taken from Ref. [11], but it is modified from its original form. **Table 1** shows that we gain two iterations by applying the Max-Shift rule in this small example. It could be concluded that if the Max-Shift rule is applied on a large text, more iterations will be skipped and therefore makes much contribution to the overall speed.

Table 1 Bad-character rule vs. extended bad-character rule vs. Max-Shift rule.

Step	Bad-character rule = d_{m-1}	Extended Bad-character rule = d_i	Max-Shift rule = $max(d_i, d_{m-1})$
1	GCATCG**CG**GAGAGAATACAGTACG GCGGAG**AG** *(shift by 2)*	GCATCG**CG**GAGAGTATACAGTACG GCGGAG**AG** *(shift by 5)*	GCATCG**CG**GAGAGAATACAGTACG GCGGAG**AG** *(shift by 5)*
2	GCATCGCGG**A**GAGAATACAGTACG GCGGAGA**G** *(shift by 1)*	GCATC**GCGGAGAG**TATACAGTACG **GCGGAGAG** *(shift by 1)*	GCATC**GCGGAGAG**AATACAGTACG **GCGGAGAG** *(shift by 2)*
3	GCATCGC**GGAG**AGAATACAGTACG GCGG**AGAG** *(shift by 2)*	GCATCGCGGAGAG**A**ATACAGTACG GCGGAGA**G** *(shift by 1)*	GCATCGCGGAGAGA**A**TACAGTACG GCGGAGA**G** *(shift by 1)*
4	GCATC**GCGGAGAG**AATACAGTACG **GCGGAGAG** *(shift by 2)*	GCATCGCGGAGAGA**A**TACAGTACG GCGGAGA**G** *(shift by 1)*	GCATCGCGGAGAGAA**T**ACAGTACG GCGGAGA**G** *(shift by 8)*
5	GCATCGCGGAGAGA**A**TACAGTACG GCGGAGA**G** *(shift by 1)*	GCATCGCGGAGAGAA**T**ACAGTACG GCGGAGA**G** *(shift by 8)*	GCATCGCGGAGAGAATACAGTA**CG** *(shift by 5)* G**C**GGAG**AG**
6	GCATCGCGGAGAGAA**T**ACAGTACG GCGGAGA**G** *(shift by 8)*	GCATCGCGGAGAGAATACAGTA**CG** *(shift by 5)* GCGGAG**AG**	
7	GCATCGCGGAGAGAATACAGTA**CG** *(shift by 2)* GCGGAG**AG**		

```
Algorithm: BC-Shift( P, ASIZE, Bc[ ] )
Output: Integer Bc[ASIZE];
String P;          //Pattern
Integer m = |P|;   //Pattern Size
Integer ASIZE;     //Alphabet Size
Begin
  from i = 0 to ASIZE-1 do    //Initialization
      Bc[i] = m;

  from i = 0 to m-2 do        //Computation
      Bc[P[i]] = m-1-i;
End.
```

```
Algorithm: Ext-Shift( P, ASIZE, MsBc[ ][ ] )
Output: Integer MsBc[m][ASIZE];
String P;
Integer m = |P|;
Integer ASIZE;
Begin
  from i = 0 to m-1 do            //Initialization
    from j = 0 to ASIZE-1 do
        MsBc[i][j] = m-i;
  from i = 0 to m-2 do            //Computation
    from j = 0 to m-2-i do
        MsBc[i][P[j]] = m-1-i-j;
End.
```

```
Procedure suffixes(X, suff[])
Integer f,g,i;
Integer m = |X|;
Begin
  suff[m-1] = m;
  g = m-1;
  for i = m-2 downto 0
  begin
    if i > g and suff[i+m-1-f] < i-g then
        suff[i] = suff[i+m-1-f];
    else
      begin
        if i < g then g = i;
        f = i;
        while g >= 0 and X[g] = X[g+m-1-f] do
            g = g-1;
        suff[i] = f-g;
      end
  end
End.
```

```
Procedure preBmGs(X, bmGs[])
Integer m = |X|;
Integer i,j,suff[m];
Begin
  suffixes( X, suff );
  for i = 0 to m-1 do
      bmGs[i] = m;
  j = 0;
  for i = m-1 downto -1 do
      if i = -1 or suff[i] = i+1
      begin
        while j < m-1-i do
          begin
            if bmGs[j] = m then
                bmGs[j] = m-1-i;
            j = j+1;
          end
      end
  for i = 0 to m-2 do
      bmGs[m-1-suff[i]] = m-1-i;
End.
```

4.4 Computation

The basic bad-character rule of the Horspool algorithm and the extended bad-character rule are given below. Since the *BC-Shift* (Bad-Character-Shift) algorithm computes shifts of the rightmost character of the pattern P, *Ext-Shift* generalizes this idea by computing the shifts of the rightmost characters of all prefixes of the pattern. Hence, we can conclude that the Ext-Shift algorithm is a generalization of the *BC-Shift* algorithm. The matrix *MsBc* is a

```
Algorithm MSH( String P, String T)
Integer m = |P|;              //Pattern size
Integer n = |T|;              //Text size
Integer j,di,dm;              //Counter and helpers
Integer MsBc[m][ASIZE]; //Max-Shift Matrix

Begin
  //Preprocessing
  ExtShift(P, ASIZE, MsBc);
  //Searching
  j = 0;
  while j <= n-m do
  begin
    if P[m-1] = T[j+m-1] then
      begin
        i = m-2;
        while i >= 0 and T[j+i] = P[i] do i = i-1;
        if i < 0 then
          begin
            OUTPUT(j);
            j = j+MsBc[0][T[j+m-1]];
          end
        else
          begin
            di = MsBc[m-1-i][T[j+i]];
            dm = MsBc[0][T[j+m-1]];
            j = j+MAX(di,dm);
          end
      end
    else    j = j+MsBc[0][T[j+m-1]];
  end
End.
```

```
Algorithm MSBM( String P, String T )
Integer m = |P|, n = |T|;
Integer j,di,dm;
Char c;                       //Auxiliary Character
Integer bmGs[m];              //Suffixes Array
Integer MsBc[m][ASIZE]; //Max-Shift Matrix
Begin
  preBmGs(P,bmGs);    //Preprocessing
  ExtShift(P, ASIZE, MsBc);
  //Searching
  j = 0;
  while j <= n-m do
    begin
      c = T[j+m-1];
      if P[m-1] = c then
        begin
          i = m-2
          while i >= 0 and T[j+i] = P[i] do    i = i-1;
          if i < 0 then
            begin
              OUTPUT(j);
              j = j+max(bmGs[0],MsBc[0][c]);
            end
          else
            begin
              di = MsBc[m-1-i][T[j+i]];
              dm = MsBc[0][c];
              j = j+max(bmGs[i],dm,di);
            end
        end
      else j = j+max(bmGs[m-1],MsBc[0][c]);
    end
End.
```

vector of m arrays of Bc.

The good-suffix algorithm is also given below in order to let everything ready for programmers. The following implementations are taken from Ref. [11]. The procedure *preBmGs*() computes the good-suffix rules which will be used in the MSBM algorithm.

5. MSH and MSBM Algorithms

5.1 MSH Algorithm

By combining the bad-character rule and the extended bad-character rule in the Horspool algorithm, we create a new variant of it named Max-Shift-Horspool (MSH) algorithm. For every mismatch i, we shift by the maximum of the bad-character rule of the rightmost character of P and the bad-character rule of the rightmost character of the prefix P_i of P. The modified part of Horspool algorithm consists in changing the shift instructions to adapt this idea.

In case the rightmost character $P[m-1]$ of the pattern P is not equal to its corresponding character $c = T[j+m-1]$ in the text T, we shift by the bad-character rule of the rightmost character of P which is represented by $MsBc[0][T[j+m-1]]$. The same thing will be applied in case we find an occurrence of P in T. In this case, we also shift by the bad-character rule of the rightmost character of P (i.e., $MsBc[0][T[j+m-1]]$). Otherwise, if the comparison stops at $i \geq 0$, we shift by the maximum of the bad-character rule of the rightmost character of P (i.e., $MsBc[0][T[j+m-1]]$) and the bad-character rule of the rightmost character of the prefix $P_i = P[0 \ldots i]$ (i.e., $MsBc[m-1-i][T[j+i]]$).

Note: Each prefix P_i of $P (0 \leq i < m)$ is characterized by the array $MsBc[m-1-i]$.

5.2 MSBM Algorithm

We apply the same adaptation to the Boyer-Moore algorithm to get a new variant called Max-Shift-Boyer-Moore (MSBM) algorithm. Since the Boyer-Moore algorithm uses two shift functions: the bad-character rule and the good-suffix rule, we will combine them with the extended bad-character rule, then we shift by the maximum of all of them.

It is known that Horspool algorithm is a simplified version of the original Boyer-Moore algorithm that consists in using the bad-character rule alone. However, by experiments, we noticed that combining all shifting rules gives significant results compared to BM and Horspool algorithms. This combination has proven a very reasonable speed increase over the overall algorithm.

In the case where the rightmost character $P[m-1]$ of the pat-

Table 2 Different data used for the test.

Alphabet size	Type of data	Size of Data (MB)
Small = 7	DNA (FASTA file) [15]	493.3
Medium = 27	Protein (Fasta file) [10]	234.6
Big = 65536	Text Web (Wikipedia XML file) [9]	148.6

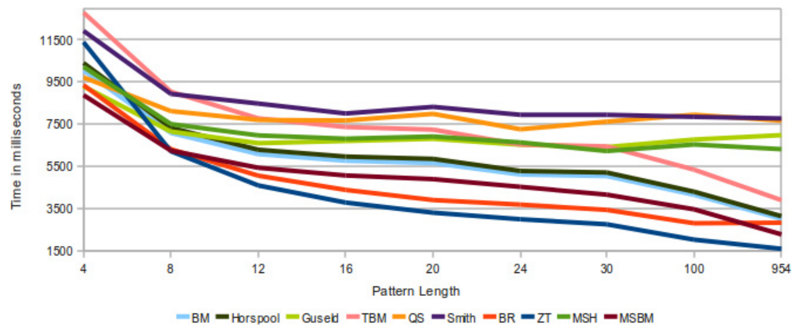


Fig. 1 Searching times in milliseconds for DNA.

tern P is not equal to its corresponding character $c = T[j+m-1]$ in the text T, we shift by the maximum of the good-suffix rule $bmGs[m-1]$ and the bad-character rule of the rightmost character of P which is represented by $MsBc[0][c]$. The same thing could be done when an occurrence of P is found in T; we shift by the maximum of the bad-character rule of the rightmost character of P (i.e., $MsBc[0][c]$) and the good-suffix rule $bmGs[0]$. If the comparison stops at $i \geq 0$, in this case we shift by the maximum of three values: 1) the bad-character rule of the rightmost character of P (i.e., $MsBc[0][c]$), 2) the bad-character rule of the rightmost character of the prefix $P_i = P[0 \ldots i]$ (i.e., $MsBc[m-1-i][T[j+i]]$), and 3) the good-suffix rule $bmGs[i]$. Note that each suffix S_i of $P(0 \leq i < m)$ is characterized by the array $bmGs[i]$.

6. Experimental Result and Comparison

6.1 Data and Algorithms Chosen

Smith, Zhu-Takaoka [20], Berry-Ravindran [21], BM, Horspool, Turbo-BM and Quick Search were the chosen algorithms for making a clear comparison against MSH and MSBM algorithms. In spite of the absence of the extended bad-character rule implementation in Ref. [14] we attempted to adapt it in the Boyer-Moor algorithm as it was mentioned by Gusfield (we will refer to it as the "Gusfield Algorithm"). All the mentioned algorithms were implemented in C language. The pattern P and the text T are loaded into memory before computation and timing started. Concerning the various data chosen for the test, we considered three kinds of data depending on the size of the alphabet.

The different algorithms were executed using Intel(R) Core(TM) i7 CPU speed 2.93 GHz, 12 GB RAM, and Kernel Linux 2.6.32-64-generic operating system. All algorithms were compiled by GCC compiler of the Qt Creator 1.3.1 (based on Qt 4.6.2 "64 bit"). BM, Horspool (BMH), Turbo-BM (TBM) and Quick Search (QS) algorithms were implemented as described in Ref. [11] with some modifications we did for generalizing algorithms' input and output and also to speed-up some parts of their implementations.

Pattern length ranges from 4 to 954 characters for protein data and DNA data, and from 4 to 200 characters for text web data.

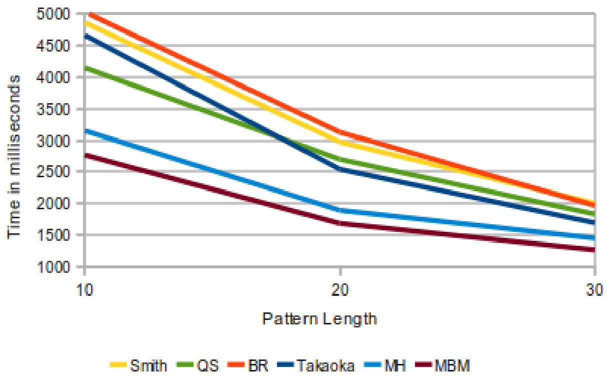


Fig. 2 Execution time when the pattern is not found in a DNA file.

In fact, we are dealing with a reasonable and big size of data that requires time to be uploaded and scanned.

The speed of our machine, in which we underwent the algorithms, is very fast. Time was measured in millisecond unit, which is a significant measurement unit for such machine. As query patterns, we selected randomly 100 substrings from each of the target text data; the mean of execution times for every pattern is shown for each experiment. The execution includes the preprocessing phase and searching phase. The preprocessing time was very negligible for all algorithms except ZT and BR algorithms that required a lot of execution time when dealing with a big alphabet size.

6.2 Experiments on Small Alphabet Size (DNA Data)

The result of the execution time for DNA data is shown in **Fig. 1** and **Fig. 2** MSBM achieved the best time when the pattern length $\leq$ 8 and the second best time when it is long enough ($\geq$ 954). ZT was the best algorithm when the pattern length $\geq$ 12 BM and Horspool algorithms achieved almost the same execution times. BM was slightly faster than Horspool algorithm. TBM was the slowest algorithm when the pattern length $\leq$ 12 while Smith algorithm was the slowest when the pattern length $>$ 12 nucleotides. Concerning the case when the pattern is not found in the data file, our algorithms MSBM and MSH achieved the best execution time as shown in Fig. 2.

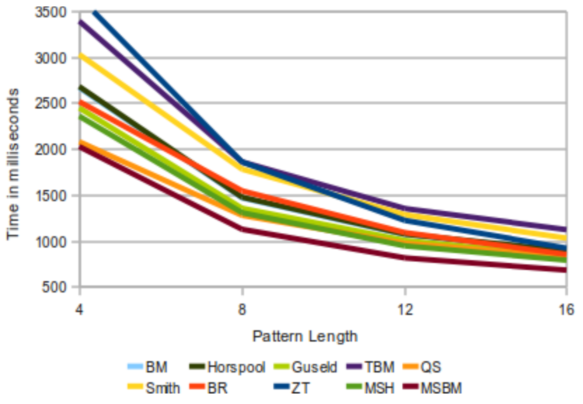


Fig. 3 Searching times in milliseconds for the Protein file (Pattern length is 4 to 16).

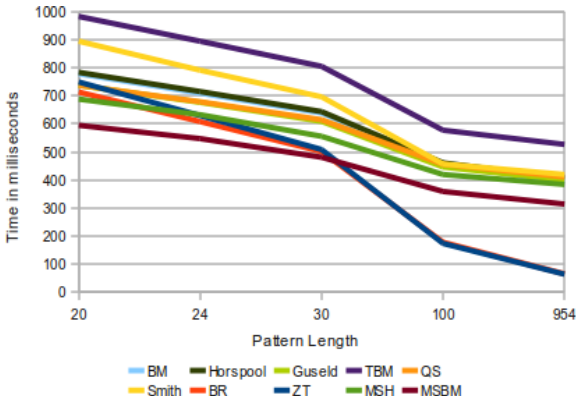


Fig. 4 Searching times in milliseconds for the Protein file (Pattern length is 20 to 954).

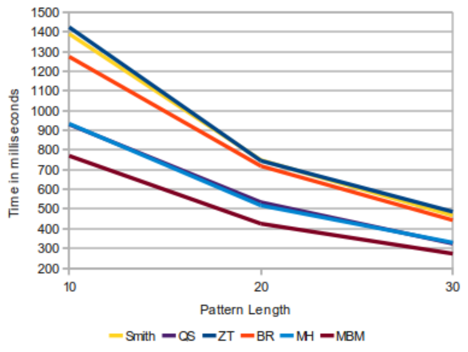


Fig. 5 Execution time when the pattern is not found in a Protein file.

6.3 Experiments on Medium Alphabet Size (Protein Data)

Fig. 3, **Fig. 4** and **Fig. 5** show the results for the search in a protein data file. Our algorithm MSBM usually won against other algorithms. When the pattern length is 4 characters, MSBM and QS achieved almost the same time and ZT algorithm won when the pattern length ≥ 100. BM and Horspool algorithms again achieved almost the same execution time, while TBM was the slowest algorithm compared to other algorithms. Besides, MSBM achieved the best execution time when the pattern is not found in the protein data file, while MSH and QS algorithms achieved the second execution time as shown in Fig. 5.

6.4 Experiments on Big Alphabet Size (Text Web Data)

We considered a big alphabet size (65,535 Unicode letters). All the algorithms did not behave unexpectedly except Zhu-Takaoka

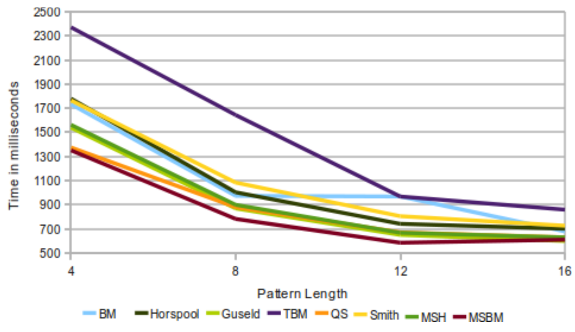


Fig. 6 Searching times in ms for the Text Web Data (Pattern length: 4 to 16).

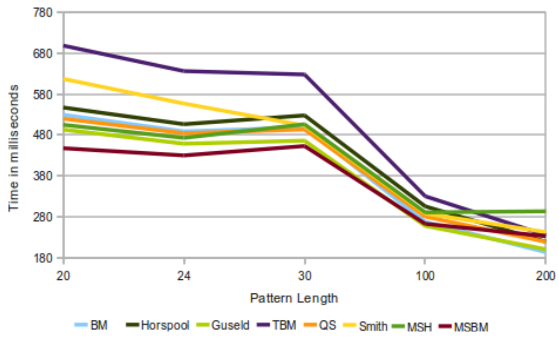


Fig. 7 Searching times in ms for the Text Web Data (Pattern length: 20 to 200).

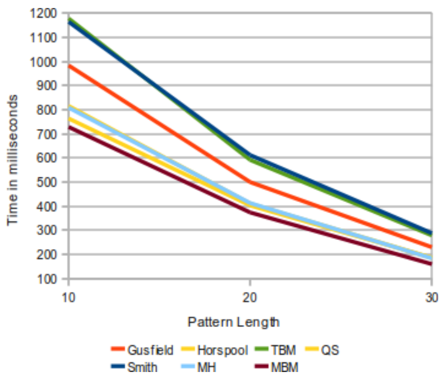


Fig. 8 Execution time when the pattern is not found in the text file.

and Berry-Ravindran algorithms that could not fit in our memory because of space limitation. This is due to their preprocessing phase that runs in $O(m + \Sigma^2)$ time and space [11], [20], [21] such that Σ is the alphabet size and m is the pattern length. Therefore, we compared with other algorithms and the results for the text web test are shown in **Fig. 6**, **Fig. 7** and **Fig. 8**. Our algorithm MSBM was the winner in most cases. However, it was slightly slower for the long pattern length of 200 characters. There were only a very few differences between BM, Gusfield, QS, Horspool and MSH when the pattern length ≥ 27. However, TBM was again the slowest algorithm in most cases. On the other hand, MSBM algorithm showed great performance and won against all the other algorithms when the pattern is not found in the text file as shown in Fig. 8.

7. Discussion

MSBM and MSH algorithms are variants of BM and Horspool algorithms respectively. The searching time of MSBM and MSH is also in $O(m \cdot n)$ time complexity, while the preprocessing phase is executed in $O(m \cdot |\Sigma|)$ time and space complexity. We may say that the space complexity of our algorithms represents their weaknesses although it is better than many other variants of BM such as Zhu-Takaoka and Berry-Ravindran algorithms, The best performance is $O(n/m)$ of both algorithms.

8. Conclusion

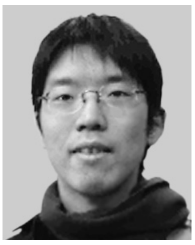
Replacing the bad-character rule strategy by its generalized version and applying the maximum of shift values (the Max-Shift rule) gives a significant improvement over Boyer-Moore algorithm and its different variants. According to our experiments, MSBM algorithm surprisingly worked well in most cases and the preprocessing phase did not affect the speed of the overall algorithm.

Acknowledgments This work was partially supported by the Grant-in-Aid from the Ministry of Education, Culture, Sports, Science and Technology of Japan. We are grateful to Pr. Satoru Miyano (the head of Laboratory of DNA Sequence Analysis and Laboratory of Sequence Analysis, Human Genome Center, The University of Tokyo) for his additional support in publishing this work.

Reference

[1] Kalsi, P., Peltola, H. and Tarhio, J.: Comparison of Exact String Matching Algorithms for Biological Sequences, *Bioinformatics Research and Development* (*BIRD*), pp.417–426 (2008).
[2] Lecroq, T.: Experimental Results on String Matching Algorithms, *Software Practice and Experience*, Vol.25, No.7, pp.727–765 (1995).
[3] Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W. and Rytter, W.: Deux méthodes pour accélérer l'algorithme de Boyer-Moore, *Théorie des Automates et Applications, Actes des 2e Journes Franco-Belges*, Krob, D. (Ed.), Rouen, France, pp.45–63 (1991).
[4] Smith, P.D.: Experiments with a Very Fast Substring Search Algorithm, *Software–Practice and Experience* (*SPE*), Vol.21, No.10, pp.1065–1074 (1991).
[5] Hume, A. and Sunday, D.: Fast string searching, *Software: Practice and Experience*, Vol.21, No.11, pp.1221–1248 (1991).
[6] Sunday, D.M.: A very fast substring search algorithm, *Comm. ACM*, Vol.33, No.8, pp.132–142 (1990).
[7] Horspool, R.N.: Practical fast searching in strings, *Software: Practice and Experience*, Vol.10, pp.501–506 (1980).
[8] Boyer, R.S. and Moore, J.S.: A fast string searching algorithm, *Comm. ACM*, Vol.20, No.10, pp.762–772 (1977).
[9] Wikipedia; Benchmark (Text Web), file name: enwiki-latest-abstract8.xml, available from ⟨http://download.wikimedia.org/enwiki/latest/⟩.
[10] Universal Protein Resource; Benchmark (Protein Fasta file), file: uniprot_sprot.fasta.gz, available from ⟨ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/⟩.
[11] Charras, C. and Lecroq, T.: *Handbook of Exact String-Matching Algorithms* (2004).
[12] Knuth, D.E., Morris, J.H. and Pratt, V.R.: Fast pattern matching in strings, *SIAM J. Comput.*, Vol.6, pp.323–350 (1977).
[13] Raita, T.: Tuning the Boyer-Moore-Horspool String Searching Algorithm, *Software: Practice and Experience*, Vol.22, No.10, pp.879–884 (1992).
[14] Gusfield, D.: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* (1997).
[15] National Center for Biotechnology Information; Benchmark: (DNA Fasta File), file 4, available from ⟨ftp://ftp.ncbi.nih.gov/pub/TraceDB/anopheles_gambiae_s/⟩.
[16] Apostolico, A. and Crochemore, M.: Optimal canonization of all substrings of a string, *Information and Computation*, Vol.95, No.1, pp.76–95 (1991).
[17] Crochemore, M. and Rytter, W.: *Text Algorithms*, Oxford University Press (1994).
[18] Colussi, I.: Correctness and efficiency of the pattern matching algorithms, *SIAM Journal on Computing*, Vol.21, No.3, pp.407–437 (1992).
[19] Galil, Z. and Giancarlo, R.: On the exact complexity of string matching: Upper bounds, *Information and Computation*, Vol.95, No.2, pp.225–251 (1991).
[20] Zhu, R.F. and Takaoka, T.: On improving the average case of the Boyer-Moore string matching algorithm, *Journal of Information Processing*, Vol.10, No.3, pp.173–177 (1987).
[21] Berry, T., Ravindran, S.: A fast string matching algorithm and experimental results, *Proc. Prague Stringology Club Workshop '99, Collaborative Report DC-99-05*, Holub, J. and Simanek, M. (Eds.), pp.16–26, Czech Technical University, Prague, Czech Republic (1999).

Mohammed Sahli a doctoral student of The University of Tokyo received his Bachelor of Mathematics and Computer Science from The University Hassan II-Mohammedia in Casablanca 2006, and Master of Research in Computer Science from University Mohamed V-Soussi in Rabat 2008. His research interest is on algorithms in Artificial Intelligence and Operations Research. The subject of his Ph.D. thesis is Genome Assembly Problem. He likes programming and is interested in Robotics as well.



Tetsuo Shibuya received his Bachelor of Science, Master of Science, and Ph.D. of Science from The University of Tokyo, in 1995, 1997, and 2002 respectively. His research interest ison algorithms in computational biology. He was a researcher at IBM Tokyo Research Laboratory from 1997 to 2004. He was an assistant professor at The University of Tokyo from 2004 to 2009, and he is an associate professor at The University of Tokyo from 2009. He is the chair of SIGBIO, IPSJ. He is editor-in-chief of the TBIO. He is a member of JSBi (Japanese Society for Bioinformatics), IPSJ (Information Processing Society of Japan), and IEICE (Institute of Electronics, Information and Communication Engineers).