On Improving the Average Case of the Boyer-Moore String Matching Algorithm*

ZHU RUI FENG** and TADAO TAKAOKA**

It is shown how to modify the Boyer-Moore string matching algorithm so that the number of characters actually inspected and the running time decrease sharply as the length of pattern gets longer.

1. Introduction

The pattern matching problem is to find all occurrences of a pattern in a text, or to decide that none exists.

A well known efficient algorithm is proposed in [2] (the BM algorithm). Unlike the Knuth-Morris-Pratt algorithm and the 'brute-force' algorithm, it compares the pattern with the text from the right end of the pattern. The performance of this algorithm is quite good in the average case, where it performs in $O(n/m)$ time. Here n is the length of text, while m is the length of the pattern.

Several variations were proposed to improve the number of character comparisons in the worst case of the BM algorithm. They are described by: Knuth [3] which achieved linear time in the worst case but lost the linear time in the preprocessing; [4] proposed by Galil which made the worst cost bounded to $14n$; and [1] recently published by Apostolico and Giancarlo which had the worst case bounded by $2n$ (known as AG algorithm).

In this paper, we present a new method to improve the average performance of the BM algorithm, which we call BM′ algorithm. The basic idea is to utilize two characters for a precomputed table instead of one character as in the original BM algorithm. Whenever a mismatch occurs, we can slide the pattern to the right a longer distance than in the original version. Our computer experiments have shown that as the pattern increases in length, the average number of characters inspected and the running time of the BM′ algorithm are much less than the original BM algorithm and [5], a recently proposed algorithm. Our solution preserves all good properties of the original BM algorithm and all worst case improvements over BM algorithm mentioned above can also be used for it.

The organization of this paper is as follows: In section 1, we review the Boyer-Moore algorithm briefly. In section 2 and 3 we introduce our variation of the BM algorithm, and in section 4 we present the results of computer experiments.

2. The Boyer-Moore Algorithm

The Boyer-Moore algorithm solves the pattern matching problem by repeatedly positioning the pattern over the text and attempting to match it. For each positioning that occurs, the algorithm starts matching the pattern against the text from the right end of the pattern. If no mismatch occurs, then the pattern has been found, otherwise the algorithm computes a shift that is an amount by which the pattern will be moved to the right before a new matching attempt is undertaken.

The algorithm is in Fig. 1 we assume that the input pattern (text) is stored in the array pattern[1:m] (text[1:n]).

```
Procedure BM;
  *p(q) points to the current characters of the pattern (text)*
  *D[ch], DD'[i] are the auxiliary shift functions.*
begin q:=m;
    repeat p:=m;
       while (p>0) and (pattern[p]=text[q]) do
       begin p:=p-1; q:=q-1 end;
       if p≠0 then q:=q+max{D[text[q]], DD'[p]}
    until (p=0) or (q>n)
end.
```

Fig. 1 The original Boyer-Moore Algorithm.

Here D and DD' are often referred to as SHIFT FUNCTIONS.

The DD' SHIFT FUNCTION is based on the idea that when the pattern is moved right, it has to (1) match over all the characters previously matched, and (2) bring a different character over the character of the text that caused the mismatch. The definition of the DD' SHIFT FUNCTION is

$$DD'[p]=\min\{s+m-p \mid (s\geq 1) \text{ and } (s\geq p \text{ or pattern}[p-s]\neq \text{pattern}[p]) \text{ and } (s\geq i \text{ or pattern}[i-s]=\text{pattern}[i],\ p<i\leq m)\}.$$

Secondly, the D SHIFT FUNCTION uses the fact

*This research was supported in part by a grant under the Monbusho International Scientific Research Program from the Japanese Government (No. 62045007)

**Department of Information Science, Ibaraki University, Hitachi, 316 Japan.

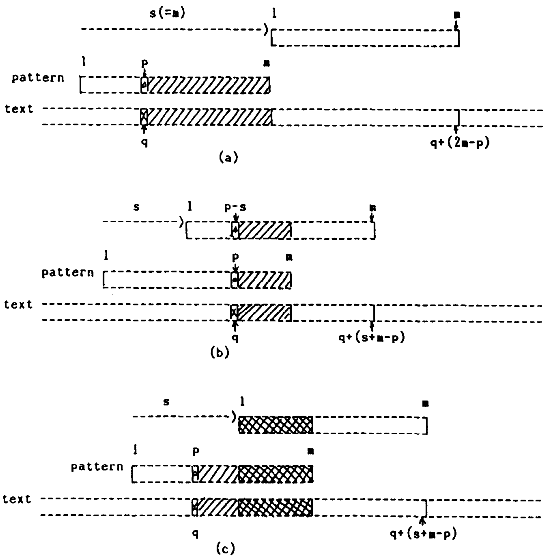

Fig. 2 The illustration of DD' SHIFT FUNCTION (a) $s=m$, (b) $s<p$, (c) $s \geq p$

that we must bring over ch=text[q] (the character that caused the mismatch), the first occurence of ch in the pattern will match it. Thus the D SHIFT FUNCTION is defined by

D[ch]=min $\{s \mid s=m$ or $(0<s<m$ and pattern$[m-s]$ =ch$)\}$.

Both SHIFT FUNCTIONS can be obtained from precomputed tables based solely on the pattern and the alphabet used. The DD' SHIFT FUNCTION requires a table of length equal to the pattern, while the D SHIFT FUNCTION requires a table of size equal to the alphabet size. Given the two values of the two SHIFT FUNCTIONS, the BM algorithm chooses the larger one. Our idea is to improve the D SHIFT FUNCTION to make the BM algorithm more powerful.

3. Our variation

We modified the D SHIFT FUNCTION to operate on a two-dimensional array for observing two characters instead of one character. The new definition of the D SHIFT FUNCTION, called D' SHIFT FUNCTION, is the following:

$$D'[\text{ch1}, \text{ch2}] = \min\{s \mid (s=m) \text{ or } (s=m-1 \text{ and ch2=pattern}[1]) \text{ or } (0<s<m-1 \text{ and pattern}[m-s-1]=\text{ch1 and pattern}[m-s]=\text{ch2})\}$$

For example: pattern: *d j e a*
D' SHIFT FUNCTION:

ch1 \ ch2	a	b	c	d	e	f	g	h	i	j
a	4	4	4	3	4	4	4	4	4	4
b	4	4	4	3	4	4	4	4	4	4
c	4	4	4	3	4	4	4	4	4	4
d	4	4	4	3	4	4	4	4	4	2
e	0	4	4	3	4	4	4	4	4	4
f	4	4	4	3	4	4	4	4	4	4
g	4	4	4	3	4	4	4	4	4	4
h	4	4	4	3	4	4	4	4	4	4
i	4	4	4	3	4	4	4	4	4	4
j	4	4	4	3	1	4	4	4	4	4

Fig. 3 An example of the D' SHIFT FUNCTION Table.

The computation of the D' SHIFT FUNCTION is:

```
Procedure D';
begin for each ch1 of Σ do
for each ch2 of Σ do if ch2=pattern[1] then d'[ch1, ch2]:=m-1
  else d'[ch1, ch2]:=m;
for j:=2 to m do
D'[pattern[j-1], pattern[j]]:=m-j
end.
```

Fig. 4 The algorithm for computing the D' SHIFT FUNCTION.

Note: Σ is the alphabet.

The algorithm is as follows:

```
Procedure BM';
1: begin q:=m;
2:     repeat p:=m;
3:        while (p>0) and (pattern[p]=text[q]) do
4:        begin p:=p-1; q:=q-1 end;
5:        if p≠0 then q:=q+max {D'[text[q-1], text[q]], DD'[p]}
6:     until (p=0) or (q>n)
7: end.
```

*Note: At line 6 if $p=0$ the algorithm stops, and only one pattern is found. We can easily modify the algorithm to find all occurrences of pattern in the text by adding a loop between line 1 and line 7. In section 4 of the computer experiments, all algorithms were written to find all occurrences of pattern in the text.

Fig. 5 BM′ Algorithm.

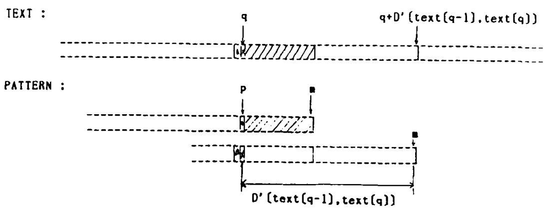

Fig. 6 An illustration of the way the pattern matching works with the D' SHIFT FUNCTION.

4. Another implementation of the D' SHIFT FUNCTION

If the alphabet size $|\Sigma|$ is large, for example, the Japanese characters of hiragana, and the pattern length is comparatively short, the D' array, which requires an

```
pattern : a b c e d c d        m=7, c=26

      |-------------------------------|
  0   |  c e             3            |
      |-------------------------------|
  1   |  a b             5            |
      |-------------------------------|
  2   |                               |
      |-------------------------------|
  3   |                               |
      |-------------------------------|
  4   |                               |
      |-------------------------------|
  5   |                               |
      |-------------------------------|
  6   |                               |
      |-------------------------------|
  7   |  b c          4               |
      |-------------------------------|
  8   |                               |
      |-------------------------------|
  9   |  e d            2             |
      |-------------------------------|
  10  |  d c            1             |
      |-------------------------------|
  11  |                               |
      |-------------------------------|
  12  |                               |
      |-------------------------------|
  13  |  c d            0             |
      |-------------------------------|
```

Fig. 7 An example of the D' SHIFT FUNCTION when implemented by a Hash Table.

array of size equal to $|\Sigma|$, becomes a large sparse matrix. In such cases, we can realize the D' SHIFT FUNCTION by using a hash table instead of the two dimensional array.

For example, the hash function may be defined as follows:

$$H(\text{ch1}, \text{ch2}) = \{[\text{ord}(\text{ch1}) - \text{ord}(`a')] * c + \text{ord}(\text{ch2}) - \text{ord}(`a')\} \text{ MOD } 2 * m$$

In the case of collision, We use another hash function defined as:

$$G(H(\text{CH1}, \text{CH2})) = [H(\text{CH1}, \text{CH2}) + m] \text{ MOD } 2 * m$$

Note: c is a constant.

The space complexity is $O(m)$, and the average case of time complexity of table look up is $O(1)$.

5. The computer experiment

We have examined the costs of the original BM algorithm, [5] algorithm and our algorithm (BM′ algorithm), with pattern of length 8 to 400 and the source string consisting of random text from a given

Table 1 The computer test of the number of references to the text.

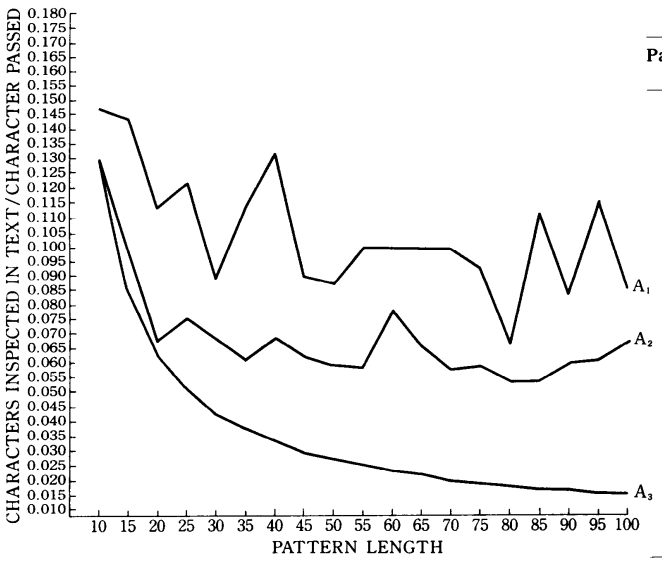

Table 1′ The computer test of the number of references to the text. (10 Characters, set[‘A’ . . ‘J’]) (The length of Text is 500,000)

Pattern Length	Characters Inspected in Text/Character passed A1	A2	A3
8	0.171510	0.158544	0.163784
10	0.147646	0.132936	0.129654
15	0.144722	0.103778	0.085090
20	0.114640	0.068120	0.063944
25	0.122714	0.076760	0.051780
30	0.089712	0.069620	0.043972
35	0.114574	0.062560	0.038412
40	0.132870	0.069972	0.034726
45	0.090170	0.063896	0.030796
50	0.088656	0.060918	0.028112
55	0.107184	0.059700	0.026438
60	0.104504	0.079970	0.024598
65	0.107970	0.067868	0.023384
70	0.101972	0.059234	0.021986
75	0.093776	0.060362	0.020838
80	0.068366	0.055844	0.019244
85	0.113972	0.055928	0.018366
90	0.085382	0.061976	0.018618
95	0.116378	0.062876	0.017756
100	0.087256	0.068120	0.017488

Table 2 The computer test of executing time.

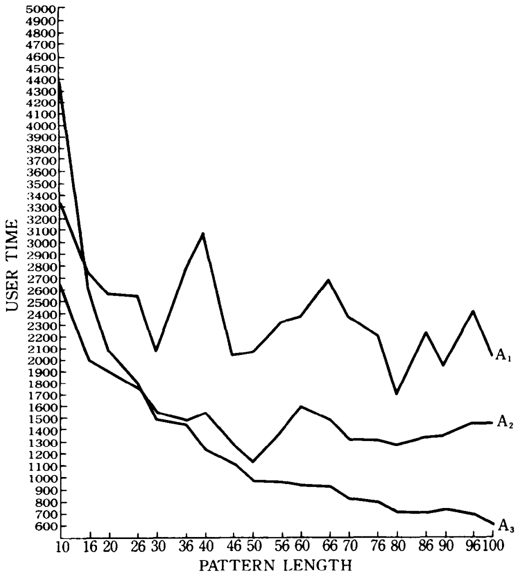

Table 2′ The computer test of executing time.
(10 Characters, set['A' . . 'J'])
(The length of Text is 1,000,000)

Pattern Length	User Time A1	A2	A3
8	3910	3140	5260
10	3320	2650	4360
16	2750	2000	2590
20	2560	1900	2090
26	2540	1780	1790
30	2060	1540	1480
36	2730	1470	1440
40	3070	1550	1230
46	2040	1270	1110
50	2070	1130	970
56	2310	1380	960
60	2360	1570	930
66	2680	1470	910
70	2380	1310	820
76	2210	1300	790
80	1680	1270	700
86	2220	1330	700
90	1940	1340	730
96	2400	1430	670
100	2030	1440	610

Table 3 The computer test of executing time.

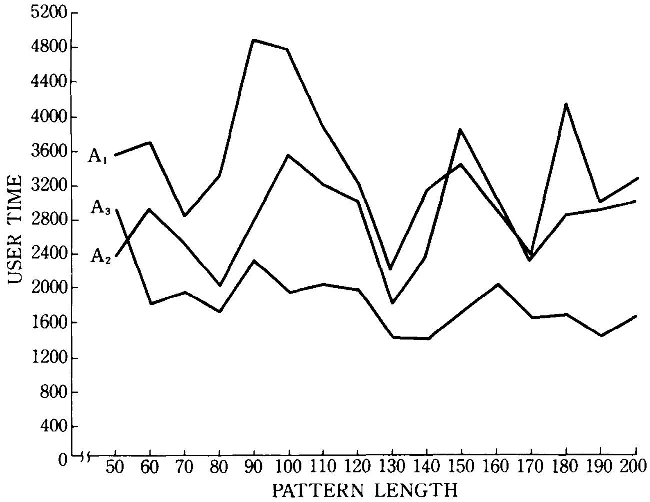

Table 3′ The computer test of executing time.
(5 Characters, set['A' . . 'E'])
(The length of Text is 1,000,000)

Pattern Length	User Time A1	A2	A3
50	3550	2350	2950
60	3780	2980	1840
70	2820	2550	1940
80	3350	2060	1780
90	4810	2770	2320
100	4790	3560	1870
110	3920	3250	2050
120	3280	3050	1980
130	2090	1810	1460
140	3150	2310	1420
150	3470	3840	1710
160	2940	3080	2030
170	2320	2280	1670
180	4150	2830	1690
190	2980	2890	1430
200	3220	2910	1620

character set (5 and 10 character alphabet, that is alphabet Σ='A' . . 'E' and Σ='A' . . 'J', respectively).

In table 1 the number of references to text per character in text passed is plotted against the pattern length for each of the three algorithms, that is, the original BM algorithm A1, [5] algorithm A2 and our algorithm A3. Note that as the length of pattern gets longer, our algorithm A3 is much more efficient in terms of direct character comparisons than either of the other two algorithms. The reason the number of references per character in text passed decreases so rapidly as the pattern length increases is that for longer patterns the probability that the two characters just fetched occurs somewhere in the pattern is lower than the one character case, and therefore the distance the pattern can be moved forward (if a mismatch occurs) is longer. But this is not obvious in cases where the pattern is short. Note also that to what extent the probability decreases depends on the alphabet size.

Table 2 and 3 shows that as for the time spent in execution of the actual code used to implement the algorithm against the pattern length, it is noticeable that when the length of pattern is longer than 26 in Table 2 and 50 in Table 3 our algorithm A3 takes much less time than either of the other two algorithms. The

Table 4 The computer test of executing time.
(Alphabet Σ is '1' . . . '}')
(The length of Text is 1,000,000)

Alphabet size $\|\Sigma\|$	Pattern length	User Time A1	A2	A3′
15	100	1470	1000	970
15	200	1710	1050	650
15	400	1590	840	570
20	100	1120	730	930
20	200	1210	790	670
20	400	1390	930	510
30	100	1130	650	970
30	200	800	540	580
30	400	820	550	370
50	100	970	450	830
50	200	560	390	520
50	400	640	420	380

*Note: A3′ means the BM′ algorithm with the *D′* SHIFT FUNCTION implemented with a hash table instead of a two dimensional array.

D′ SHIFT FUNCTION in A3 is implemented with a two dimensional array.

Table 4 shows some experimental results when the *D′* SHIFT FUNCTION of the BM′ algorithm is implemented with a hash table instead of the two-dimensional array. The user time of the BM′ algorithm will decrease faster, as the pattern length increases, since the overhead time of the hash function is a constant.

References

1. APOSTOLICO, A. and GIANCARLO, R. The Boyer-Moore-Galil String Searching Strategies Revisited, *SIAM J. Comput.* **15**, 1 (Feb. 1986).
2. BOYER, R. S. and MOORE, J. S. A Fast String Matching Algorithm, *Comm. ACM* **20**, 10 (1977), 762–772.
3. KNUTH, D. E., MORRIS, J. H. and PRATT, V. R. Fast Pattern Matching in Strings, *SIAM J. Comput.* **6**, 2 (1977), 323–350.
4. GALIL, Z. On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm, *Comm. ACM* **22**, 9 (1979), 505–508.
5. SEMBA, I. An Efficient String Searching Algorithm, *Journal of Information Processing* **8**, 2 (1985).

(Received November 10, 1986; revised May 25, 1987)