An Efficient String Searching Algorithm

ICHIRO SEMBA*

The string searching problem is to find all occurrences of a pattern in a text or to determine that none exists. We measure the cost of the string searching algorithm by the number of comparisons performed between characters of the pattern and the characters of the text.

We present an efficient string searching algorithm based on the idea of Knuth, Boyer-Moore and Knuth-Morris-Pratt algorithms. It is shown that $\lfloor n/m \rfloor \leqq$ the cost $\leqq 2n$ (where n is the length of the text and m is the length of the pattern). The preprocessing time is proved to be linear.

Computer tests indicate that for large size character set the average cost of our algorithm is less than that of the Boyer-Moore algorithm.

1. Introduction

The string searching problem is to find all occurrences of a pattern in a text or to determine that none exists. In many information retrieval, artificial intelligence and text-editing applications it is necessary to solve this problem as quick as possible. We measure the cost of a string searching algorithm by the number of comparisons performed between characters of the pattern and the characters of the text.

In 1977, two efficient algorithms solving the problem were proposed. They are the Boyer-Moore algorithm (BM) [2] and the Knuth-Morris-Pratt algorithm (KMP) [5]. The KMP algorithm attempts to match from the left end of the pattern. On the other hand, the BM algorithm attempts to match from the right end of the pattern. This is a remarkable contrast.

Both the BM algorithm and the KMP algorithm have good properties such as fast running time, linear preprocessing and simplicity. The BM algorithm is superior to the KMP algorithm on the best and average case performance. In the KMP algorithm, all characters in the text are examined. However the BM algorithm often examines a fraction of the characters in the text. Therefore it runs quickly in many applications. The best case cost of the BM algorithm is $\lfloor n/m \rfloor$ and the best case cost of the KMP algorithm is n. On the other hand, the KMP algorithm is superior to the BM algorithm for the worst case performance. The worst case behaviour in the KMP algorithm is linear, that is, the cost of the KMP algorithm is bounded by $2n$. The worst case behaviour in the BM algorithm is not linear, because it forgets all 'previous information' about characters already matched when the pattern is moved to the right.

Several variations were proposed to improve the worst case performance of the BM algorithm. Variations described by Knuth [5] have gained the linear time in the worst case, but lost the linear time in the preprocessing. Galil [3] showed how to modify the BM algorithm and proved that the worst case behaviour is linear, that is, the cost of the Galil algorithm is bounded by $14n$. If the conjecture of Guibas and Odlyzko [4] is true, its bound is improved from $14n$ to $8n$. The Galil variation preserves good properties of the BM algorithm.

Recently, Apostolico and Giancarlo [1] have proposed an efficient algorithm (the AG algorithm). They have improved the BM algorithm and the Galil algorithm. It remembers all 'previous information' about characters already matched when the pattern is moved to the right. By using this information, it does its job without examining the matched characters twice. Therefore the worst case cost of the AG algorithm is bounded by $2n$. The preprocessing time is linear.

In this paper we will present an efficient string searching algorithm. The set of patterns of length m is divided into three subsets and three efficient algorithms suitable for each subset are presented. They are called A, B and C. An efficient string searching algorithm can be constructed by combining these three algorithms. It is shown that $\lfloor n/m \rfloor \leqq$ the cost $\leqq 2n$. The preprocessing time is proved to be linear.

2. Basic Idea

In this section we will show the basic idea.

First several notations are introduced.

The text is represented by an array text[1:n] and the pattern is represented by an array pattern[1:m]. The character at the position i in the text(pattern) is denoted by text[i] for $1 \leqq i \leqq n$ (pattern[i] for $1 \leqq i \leqq m$) and the characters at positions i through j in the text for $1 \leqq i \leqq j \leqq n$ (pattern for $1 \leqq i \leqq j \leqq m$) is denoted by text[$i$:$j$] (pattern[$i$:$j$]). Thus text[$i$:$i$] (pattern[$i$:$i$]) is equal to text[$i$] (pattern[$i$]). If $i > j$, then text[i:j] (pattern[i:j]) means an empty string. The notation text[i:j]

*Department of Pure and Applied Sciences, The college of Arts and Sciences, University of Tokyo, Komaba, Meguro-ku, Tokyo, 153, Japan

$=$pattern$[k:l]$ means that for $1 \leqq i \leqq j \leqq n$ and $1 \leqq k \leqq l \leqq m$, $j-i=l-k$ and text$[i+h]$=pattern$[k+h]$ $(0 \leqq h \leqq j-i)$. The notation pattern$[i:j]$=pattern$[k:l]$ has the same meaning.

Second we will define the quantity H(pattern$[1:m]$).

H(pattern$[1:m]$)$=\max\{j \mid j=1$or $(1<j \leqq m$ and pattern$[j] \neq$pattern$[i]$ $(1 \leqq i<j)\}$

Example. H(abcdef)$=6$, H(aaabbbccc)$=7$ and H(aaaaa)$=1$.

Now we will show an overview of the basic searching strategy. Implementations of the algorithms are described in the section 3, 4 and 5.

(1) The pattern$[m]$ is examined. If a match occurs, then step (2) is executed. Otherwise, the pattern is moved to the right by using the same heuristic in the BM algorithm and other information. Then step (1) is executed.

(2) The pattern$[1:h]$ (pattern$[1:h-1]$ if $h=m$) is examined from right to left, where $h=H$(pattern$[1:m]$). If no mismatch occurs, then step (3) is executed. Otherwise, the pattern is moved to the right by using the property h and the fact that a match has occurred at the position m in the pattern. Then step (1) is executed.

(3) When $h=m$ or $m-1$, the pattern has been found and moved to the right by using the same information as (2). Then step (1) is executed. When $1 \leqq h<m-1$, the pattern$[h+1:m-1]$ is examined from left to right. If no mismatch occurs, then the pattern has been found. Otherwise, the pattern is moved to the right by using the same procedure as the KMP algorithm. Then the KMP algorithm is continued or step (1) is executed.

We can easily see that the worst case cost of our algorithm is n when $h=m$. By the property of the KMP algorithm, it follows that the worst case cost of our algorithm is $2n$ when $1 \leqq h<m$. However, with some tricks, we can do our work without the KMP algorithm for the pattern$[1:m]$ such that $\lceil m/2 \rceil \leqq$H(pattern$[1:m]$)$<m$ and the worst case cost is expected to be less than $2n$. Therefore we have divided the set of patterns of length m based on the character set $\{c_1, c_2, \cdots, c_q\}$, $Q(q, m)$, into three subsets and have developed three string searching algorithms suitable for each subset.

The set $P(q, m, h)$ $(1 \leqq h \leqq m)$ is defined as follows.

$P(q, m, h)=\{$pattern$[1:m] \mid$ pattern$[1:m] \in Q(q, m)$ and H(pattern$[1:m]$)$=h\}$

Three subsets are defined as follows.

(1) $P(q, m, m)$

(2) $P(q, m, m-1) \cup P(q, m, m-2) \cup \cdots \cup P(q, m, \lceil m/2 \rceil)$

(3) $P(q, m, \lceil m/2 \rceil - 1) \cup \cdots \cup P(q, m, 2) \cup P(q, m, 1)$

Lastly we will show an interesting result related to the set $P(q, m, h)$ $(1 \leqq h \leqq m)$. This result will make clear the difference between the sizes of the three subsets. The binomial coefficient is denoted by $C(i, j)$ and defined to be zero if $i<j$. The second kind Stirling number is denoted by $S(i, j)$.

Theorem 2.1. For $1 \leqq h \leqq m$,

$$|P(q, m, h)| = \begin{cases} q & \text{if } h=1 \\ \sum_{i=1}^{h-1} C(q-1, i)S(h-1, i)i!(i+1)^{m-h}q & \text{if } 1<h \leqq m \end{cases}$$

Proof. It is obvious that $|P(q, m, 1)|=q$. We fix the character pattern$[h]$. We assume that the strings of length $h-1$ $(1 \leqq i \leqq h-1)$ appearing on the left side of pattern$[h]$ contain i different characters $c_1, c_2, \cdots, c_i$ chosen from $q-1$ possible characters other than pattern$[h]$. There are $C(q-1, i)$ ways of choosing i different characters and the number of those strings of length $h-1$ is $S(h-1, i)i!$. Therefore there are $C(q-1, i)S(h-1, i)i!$ different strings of length $h-1$ on the left side of pattern$[h]$. The strings of length $m-h$ appearing on the right side of pattern$[h]$ have to be constructed from $i+1$ characters pattern$[h]$, $c_1, \cdots, c_i$. Therefore there are $(i+1)^{m-h}$ different strings of length $m-h$ on the right side of pattern$[h]$. Since pattern$[h]$ is one of q possible characters and $1 \leqq i \leqq h-1$, we obtain the above results.

Example. The table $|P(20, 10, h)|$ $(1 \leqq h \leqq 10)$ is computed.

h	$\lvert P(20, 10, h) \rvert$
10	6453753955580
9	2605840197520
8	884589561080
7	240221603320
6	48548239880
5	6540071320
4	491266280
3	15007720
2	97280
1	20

3. The Algorithm A

The algorithm A is designed to search a pattern of length m included in the set $P(q, m, m)$. This algorithm is based on a similar idea of Knuth's algorithm [5] proposed for patterns of length m consisting of m different characters. We note that algorithm A is identical with Knuth's algorithm for this pattern.

Two tables $d[ch]$ (ch is included in the set $\{c_1, c_2, \cdots, c_q\}$) and $g[j]$ $(1 \leqq j \leqq m)$ are precomputed and used in algorithm A. They are defined as follows.

$d[ch]=\max\{x \mid x=0$ or $(0<x \leqq m$ and pattern$[x]=ch)\}$

This means that $d[ch]$ is the rightmost position where the character ch appears in the pattern$[1:m]$. If the character ch is not found in the pattern$[1:m]$, $d[ch]$ is

defined to be zero.

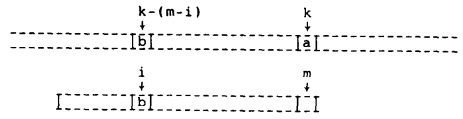
$g[j]=\max\{x|\ x=0$ or
$(0<x<j$ and pattern$[x]$=pattern$[j])\}$

This means that $g[j]$ is the rightmost position where the character pattern$[j]$ appears in the pattern$[1:j-1]$. If the character pattern$[j]$ is not found in the pattern$[1:j-1]$, then $g[j]$ is defined to be zero. We define $g[0]=0$.

The processing of algorithm A can be divided into three parts. The integer variables i, $j(k)$ are used as a pointer to the pattern(text). We suppose that the character text$[k]$ and the character pattern$[j]$ are examined. We mean by $i=0$ that we have no 'previous information' about characters already matched.

Case A1.

text$[k-(m-i)]$=pattern$[i]$ $(0\leqq i\leqq m-1)$
text$[k]\neq$pattern$[m]$

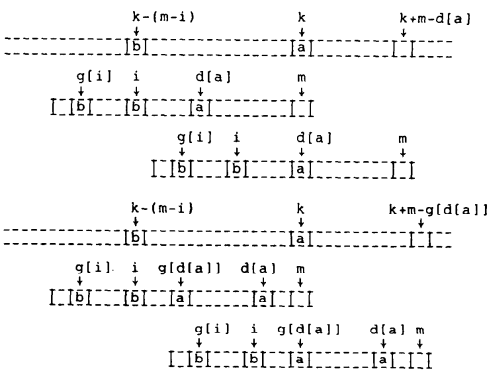


Let text$[k]=a$ and text$[k-(m-i)]=b$. When a mismatch occurs at the position $m(k)$ in the pattern(text), the pattern can be moved to the right.

If $m-d[a]\geqq i-g[i]$, then the pattern will be moved to the right by $m-d[a]$ and the pointer i is set to $d[a]$.

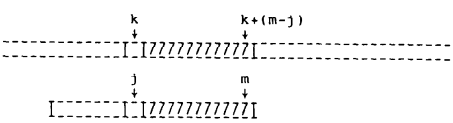
If $m-d[a]<i-g[i]$, then the pattern will be moved to the right by $m-g[d[a]]$ and the pointer i is set to $g[d[a]]$.

Then a new matching attempt is undertaken at the position $m(k+m-i)$ in the pattern(text).

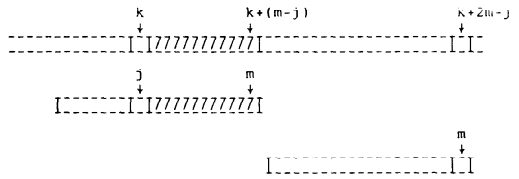


Case A2.

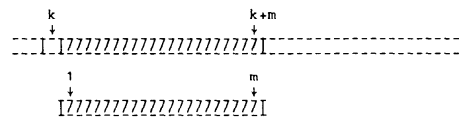
text$[k+1:k+(m-j)]$=pattern$[j+1:m]$
text$[k]\neq$pattern$[j]$$(1\leqq j\leqq m-1)$



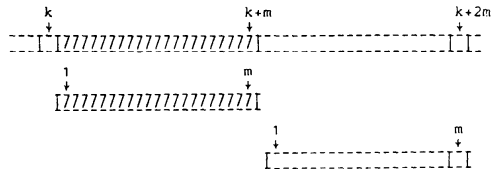
When a mismatch occurs at the position $j(k)$ in the pattern(text), the pattern can be moved to the right by m and the pointer i is set to zero. Then a new matching attempt is undertaken at the position $m(k+2m-j)$ in the patter(text).



Case A3. text$[k+1:k+m]$=pattern$[1:m]$



The pattern has been found. The pattern can be moved to the right by m and the pointer i is set to zero. Then a new matching attempt is undertaken at the position $m(k+2m)$ in the pattern(text).



The algorithm A is written in Pascal-like language and shown in Fig. 1.

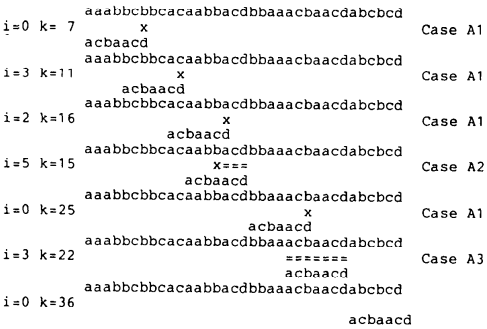
Example. We consider 6 possible characters a, b, c, d, e, f and pattern[1:7]=acbaacd included in the set P(6, 7, 7). The mismatched character is marked $\times$ and the matched character is marked $=$. Two tables $d[ch]$ and $g[j]$ are precomputed.

We consider text=aaabbcbbcacaabbacdbbaaacbaacdabcbcd.

```
begin
  k:=m; i:=0;
1:
  if k>n then stop;
  if text[k]≠pattern[m] then begin
  {Case 1}
    l:=d[text[k]]; if m-l≧i-g[i] then i:=l else i:=g[l];
    k:=k+m-i; goto 1;
  end;
  j:=m;
2:{Case A2}
  k:=k-1; j:=j-1;
  if j=0 then goto 3;
  if text[k]≠pattern[j] then begin
    i:=0; k:=k+2*m-j; goto 1;
  end;
  goto 2;
3:{Case A3}
  i:=0; k:=k+2*m; goto 1;
end.
```

Fig. 1. The algorithm A.

ch	a	b	c	d	e	f		
d[*ch*]	5	3	6	7	0	0		
j	0	1	2	3	4	5	6	7
g[*j*]	0	0	0	0	1	4	2	0



4. The Algorithm B

The algorithm B is designed to search a pattern of length m included in the set P(q, m, h) ($\lceil m/2 \rceil \leqq h \leqq m-1$).

In addition to tables $d[ch]$ and $g[j]$, three tables $gg[j]$ ($0 \leqq j \leqq m$), $f[j]$ ($h+1 \leqq j \leqq m-1$), $f[m+1]$ and $ff[j]$ ($h+1 \leqq j \leqq m-1$) are precomputed and used in algorithm B. The table $f[j]$ is the same as the failure function introduced in the KMP algorithm. The processing of algorithm B is divided into five parts.

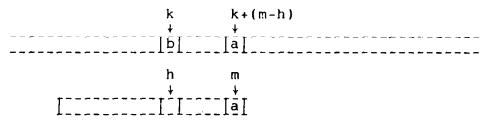
Case B1.

text[$k-(m-i)$]=pattern[i] ($0 \leqq i \leqq m-1$)
text[k] $\neq$ pattern[m]

The processing of this case is the same as that of case A1.

Case B2.

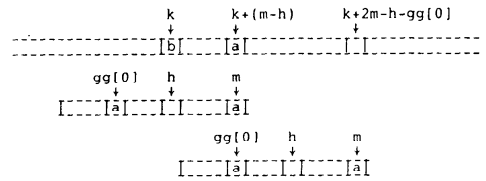
text[$k+(m-h)$]=pattern[m]
text[k] $\neq$ pattern[h]



When a mismatch occurs at the position $h(k)$ in the pattern(text), the pattern can be moved to the right. Let text[$k+(m-h)$]=a and text[k]=b. We define $gg[0]$ as follows.

max{x| $x=0$ or
($0<x \leqq m-h$ and pattern[x]=pattern[m])}.

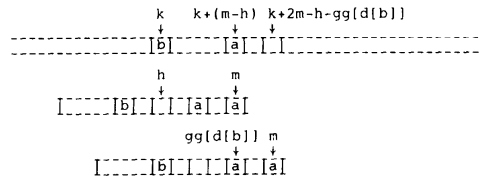
This means that $gg[0]$ is the rightmost position in the pattern[1:$m-h$] where the character is equal to pattern[m]. When $d[b]=0$, the pattern will be moved to the right by $m-gg[0]$ and the pointer i is set to $gg[0]$.



When $d[b]>0$, we have to examine whether the pointer x ($m-h<x \leqq m-1$), such that pattern[$x-(m-h)$]=b and pattern[x]=pattern[m], exists or not. We define $gg[d[b]]$ ($d[b]>0$) as follows.

max{x| $gg[0]$ or
($m-h<x \leqq m-1$ and pattern[$x-(m-h)$]=b
and pattern[x]=pattern[m])}.

If it is found, we can move the pattern to the right by $m-gg[d[b]]$ and the pointer i is set to $gg[d[b]]$.

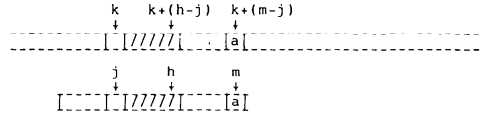


If it is not found, we can move the pattern to the right by $m-gg[0]$. The pointer i is set to $gg[0]$.

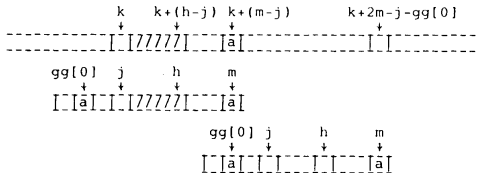
A new matching attempt is undertaken at the position $m(k+2m-h-gg[d[b]])$ in the pattern(text).

Case B3.

text[$k+(m-j)$]=pattern[m]
text[$k+1$:$k+(h-j)$]=pattern[$j+1$:h]
text[k] $\neq$ pattern[j] ($1 \leqq j < h$)

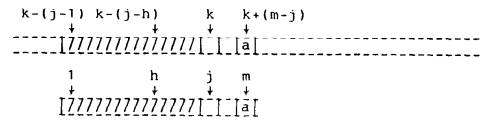


When a mismatch occurs at the position $j(k)$ in the pattern(text), the pattern can be moved to the right by $m-gg[0]$ and the pointer i would be set to $gg[0]$. A new matching attempt is undertaken at the position $m(k+2m-j-gg[0])$ in the pattern(text).



Case B4.

text[$k+(m-h)$]=pattern[m]
text[$k-(j-1)$:$k-1$]=pattern[1:$j-1$]
text[k] $\neq$ pattern[j] ($h<j \leqq m-1$)

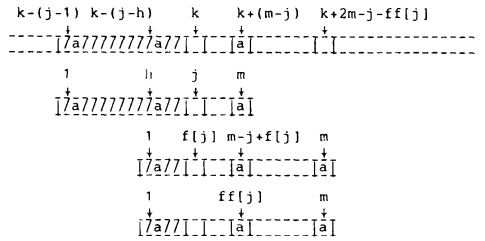


In this case we use the table $f[j]$ $(h+1 \leqq j \leqq m-1)$ introduced in the KMP algorithm.

$f[j]=\max\{x|\ x=0$ or
$(1 \leqq x<j$ and pattern$[x] \neq$ patter$[j]$
and pattern$[1:x-1]=$pattern$[j-x+1:j-1])\}$

When a mismatch occurs at the position $j(k)$ in the pattern(text), the pattern can be moved to the right.

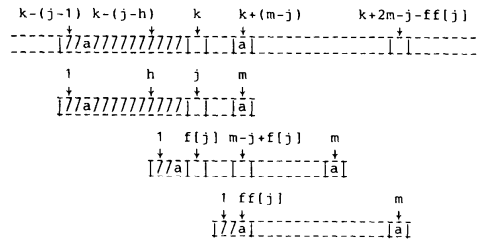
If pattern$[m-j+f[j]]=$pattern$[m]$, then $ff[j]$ is defined to be $f[j]+m-j$. The pattern will be moved to the right by $m-ff[j]=j-f[j]$ and the pointer i is set to $ff[j]$.



If pattern$[m-j+f[j]] \neq$pattern$[m]$, then $ff[j]$ is defined to be

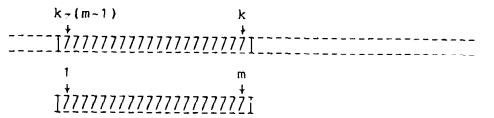
$\max\{x|\ x=0$ or
$(0<x<m-j+f[j]$ and pattern$[x]=$pattern$[m])\}$.

The pattern will be moved to the right by $m-ff[j]$ and the pointer i is set to $ff[j]$.



A new matching attempt is undertaken at the position $m(k+2m-j-ff[j])$ in the pattern(text).

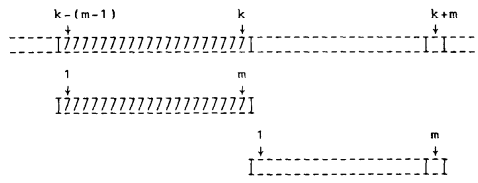
Case B5. text$[k-(m-1):k]=$pattern$[1:m]$



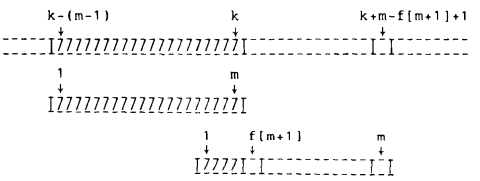
The pattern has been found. In this case we use the table $f[m+1]$ introduced in the KMP algorithm.

$f[m+1]=\max\{x|\ x=0$ or $(2 \leqq x \leqq m$ and
pattern$[1:x-1]=$pattern$[m-x+2:m])\}$

If $f[m+1]=0$, then the pattern will be moved to the right by m and the pointer i is set to zero. A new matching attempt is undertaken at the position $m(k+m)$ in the pattern(text).



If $f[m+1]>0$, then the pattern will be moved to the right by $m-f[m+1]+1$ and the pointer i is set to $f[m+1]-1$. A new matching attempt is undertaken at the position $m(k+m-f[m+1]+1)$ in the pattern(text).



The algorithm B is written in a Pascal-like language and shown in Fig. 2.

Example. We consider 6 possible characters a, b, c, d, e, f and pattern[1:10]=abcabdacab included in the set P(6, 10, 6). We consider text=abcabdabcbaabdbabbabcbccabcaabcabdacabab.

ch	a	b	c	d	e	f						
d[ch]	9	10	8	6	0	0						
j	0	1	2	3	4	5	6	7	8	9	10	11
g[j]	0	0	0	0	1	2	0	4	3	7	5	
gg[j]	2	2	2	2	2	2	2	2	2	5	2	
f[j]							0	2	0			3
ff[j]							2	2	0			

```
                abcabdabcbaabdbabbabcbccabcaabcabdacabab
i=0 k= 8 =======x =                                      Case B4
                abcabdacab
                abcabdabcbaabdbabbabcbccabcaabcabdacabab
i=2 k=11           x===   =                              Case B3
                 abcabdacab
                abcabdabcbaabdbabbabcbccabcaabcabdacabab
i=2 k=22                   x   =                         Case B2
                         abcabdacab
                abcabdabcbaabdbabbabcbccabcaabcabdacabab
i=2 k=34                           x                     Case B1
                                 abcabdacab
                abcabdabcbaabdbabbabcbccabcaabcabdacabab
i=6 k=38                           ==========            Case B5
                                   abcabdacab
                abcabdabcbaabdbabbabcbccabcaabcabdacabab
i=2 k=46
                                             abcabdacab
```

```
begin
  k:=m; i=0;
1:
  if k>n then stop;
  if text[k]≠pattern[m] then begin
  {Case B1}
    l:=d[text[k]]; if m−l≧i−g[i] then i:=l else i:=g[l];
    k:=k+m−i; goto 1;
  end;
  {Case B2}
  k:=k−(m−h); j:=h;
  if text[k]≠pattern[j] then begin
    i:=gg[d[text[k]]]; k:=k+2*m−h−i; goto 1;
  end;
2:{Case B3}
  k:=k−1; j:=j−1; if j=0 then goto 3;
  if text[k]≠pattern[j] then begin
  i:=gg[0]; k:=k+2*m−j−i; goto 1;
  end;
  goto 2;
3:
  k:=k+h+1; j:=h+1;
4:{Case B4}
  if k>n then stop;
  if j≧m then goto 5;
  if text[k]=pattern[j] then begin
    k:=k+1; j:=j+1; goto 4;
  end;
  i:=ff[j]; k:=k+2*m−j−i; goto 1;
5:{Case B5}
  if f[m+1]=0
    then begin i:=0; k:=k+m; end
    else begin i:=f[m+1]−1; k:=k+m−i; end;
  goto 1;
end.
```

Fig. 2. The algorithm B.

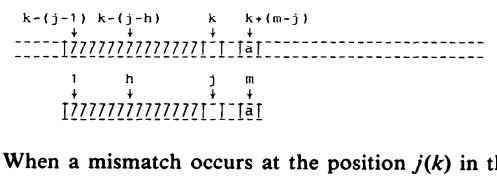
5. The Algorithm C

The algorithm C is designed to search a pattern of length m included in the set $P(q, m, h)$ $(1 \leqq h < \lceil m/2 \rceil)$.

In addition to tables $d[ch]$ and $g[j]$, two tables $gg[j]$ $(0 \leqq j \leqq m)$ and $f[j]$ $(1 \leqq j \leqq m+1)$ are precomputed and used by algorithm C. The table $f[j]$ is the same as the failure function introduced in the KMP algorithm. The processing of algorithm C is divided into five parts. The processing of case C$i$$(1 \leqq i \leqq 3)$ is similar to that of case B$i$$(1 \leqq i \leqq 3)$. The processing of case C$i$$(4 \leqq i \leqq 5)$ is different from that of case B$i$$(4 \leqq i \leqq 5)$.

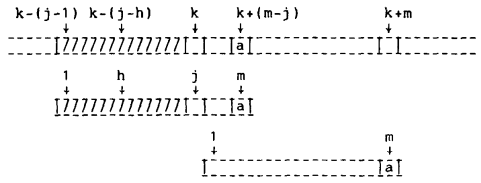
Case C4.

text$[k+(m-j)]$=pattern$[m]$
text$[k-(j-1):k-1]$=pattern$[1:j-1]$
text$[k] \neq$ pattern$[j]$ $(h < j \leqq m-1)$

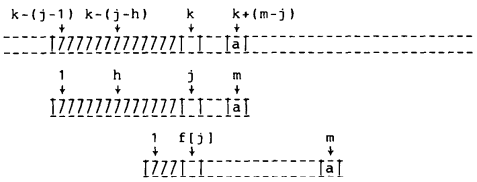


When a mismatch occurs at the position $j(k)$ in the pattern(text), the pattern can be moved to the right by $j-f[j]$. We note the fact that text$[k+(m-j)]$=pattern$[m]$ is not used in this case.

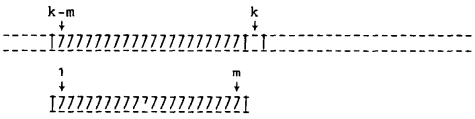
If $f[j]=0$, then a new matching attempt is undertaken at the position $m(k+m)$ in the pattern(text) and the processing of case C1 starts. The pointer i is set to zero.



If $f[j]>0$, then a new matching attempt is undertaken at the position $f[j](k)$ in the pattern(text) and the KMP algorithm is applied. While a mismatch occurs at the position j and $f[j]>0$, the KMP algorithm is continued. When $f[j]=0$, the processing of case C1 starts. The pointer i is set to zero.

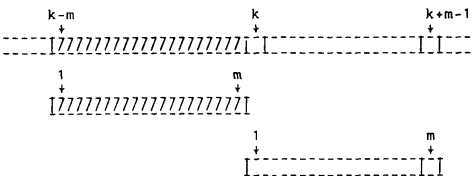


Case C5. text$[k-m:k-1]$=pattern$[1:m]$

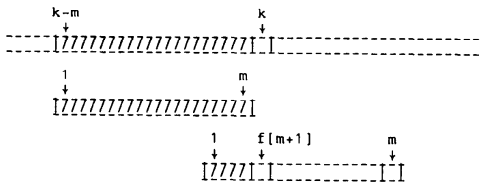


The pattern has been found.

If $f[m+1]=0$, then the pattern can be moved to the right by m and the pointer i would be set to zero. A new matching attempt is undertaken at the position $m(k+m-1)$ in the pattern(text) and the processing of case C1 starts.



If $f[m+1]>0$, then the pattern can be moved to the right by $m-f[m+1]+1$ and the KMP algorithm starts at the position $f[m+1](k)$ in the pattern(text). While a mismatch occurs at position j and $f[j]>0$, the KMP algorithm is continued. When $f[j]=0$, the processing of case C1 starts. The pointer i is set to zero.




```
begin
  k:=m; i:=0;
1:
  if k>n then stop;
  if text[k]≠pattern[m] then begin
  {Case C1}
    l:=d[text[k]]; if m-l≧i-g[i] then i:=l else i:=g[l];
    k:=k+m-i; goto 1;
  end;
  {Case C2}
  k:=k-(m-h); j:=h;
  if text[k]≠pattern[j] then begin
    i:=gg[d[text[k]]]; k:=k+2*m-h-i; goto 1;
  end;
2:{Case C3}
  k:=k-1; j:=j-1; if j=0 then goto 3;
  if text[k]≠pattern[j] then begin
      i:=gg[0]; k:=k+2*m-j-i; goto 1;
  end;
  goto 2;
3:
  k:=k+h+1; j:=h+1;
4:{Case C4}
  if k>n then stop;
  if j>m then goto 6;
5:
  if text[k]=pattern[j] then begin
    k:=k+1; j:=j+1; goto 4;
  end;
  j:=f[j];
  if j=0 then begin i:=0; k:=k+m; goto 1; end
         else goto 5; {The KMP algorithm is executed}
6:{Case C5}
  if f[m+1]=0 then begin i:=0; k:=k+m-1; end
              else begin j:=f[m+1]; goto 5; end;
  goto 1;
end.
```

Fig. 3. The algorithm C.

The algorithm C is written in a Pascal-like language and shown in Fig. 3.

Example. We consider 6 possible characters a, b, c, d, e, f and pattern[1:9]=abcdabcab included in the set P(6, 9, 4). We consider text=abcdababbbaabdbabbababcdabcabbcabbbacbca.

ch	a	b	c	d	e	f
d[*ch*]	8	9	7	4	0	0

j	0	1	2	3	4	5	6	7	8	9	10
g[*j*]	0	0	0	0	0	1	2	3	5	6	
gg[*j*]	2	2	2	2	2	2	2	2	6	2	
f[*j*]		0	1	1	1	0	1	1	4	1	3

```
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=0 k= 7  ======x =                                  Case C4
          abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=0 k= 9        ==x                                  Case C4
             abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=0 k= 9          x                                  Case C4
                  abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=0 k=13        x   =                                Case C2
              abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=2 k=25                      x                      Case C1
                        abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=5 k=30                      =========              Case C5
                              abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=5 k=30                            x                Case C5
                                 abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=5 k=30                            x                Case C5
                                    abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=0 k=39                                      x      Case C1
                                     abcdabcab
          abcdababbbaabdbabbababcdabcabbcabbbacbca
i=7 k=41
                                        abcdabcab
```

6. Cost of algorithms

We measure the cost of the string searching algorithm by the number of comparisons performed between characters of the pattern and characters of the text. In order to estimate the worst case cost, we consider the ratio of the number of comparisons performed (from the start of new matching attempt to the occurrence of a mismatch) to the movement of the pattern caused by a mismatch. It follows that the worst case cost is less than or equal to the maximum ratio times n. The following theorem is obvious.

Theorem 6.1 For a pattern included in the set P(q, m, m), $\lfloor n/m \rfloor \leqq$ the cost of the algorithm A $\leqq n$.

We denote by $r[j]$ $(1 \leqq j \leqq 5)$ the maximum ratio for each case Bj.

Theorem 6.2 For a pattern included in the set P(q, m, h) ($\lceil m/2 \rceil \leqq h \leqq m-1$),

$\lfloor n/m \rfloor \leqq$ the cost of the algorithm B

$$\leqq \max_{1 \leqq j \leqq 5} \{r[j]\} n \leqq 2n.$$

Proof. The best case cost is easily derived.

Case B1.

It is obvious that $r[1] \leqq 1$.

Case B2.

The number of comparisons=2.

The movement=$m-gg[d[ch]]$.

It follows that $r[2]=\max\{2/(m-gg[d[ch]])\}$.

By the fact that $gg[d[ch]] \leqq m-1$, it is derived that $r[2] \leqq 2$.

Case B3.

The number of comparisons=$h-j+2$ $(1 \leqq j \leqq h-1)$.

The movement=$m-gg[0]$.

It follows that

$$r[3]=\max_{1 \leqq j \leqq h-1} \{(h-j+2)/(m-gg[0])\}$$

$$=(h+1)/(m-gg[0]).$$

By the fact that $gg[0] \geqq m-h$ and $h \geqq \lceil m/2 \rceil$, we can conclude that $r[3] \leqq (h+1)/h \leqq 2$.

Case B4.

The number of comparisons $=j+1$ $(h+1 \leqq j \leqq m-1)$.
The movement $=m-ff[j]$.
It follows that

$$r[4]=\max_{h+1 \leqq j \leqq m-1}\{(j+1)/(m-ff[j])\}.$$

By the fact that $ff[j] \leqq m-h$, we can conclude that $r[4] \leqq m/h \leqq 2$.

Case B5.

The number of comparisons $=m$.
The movement

$$=\begin{cases} m & \text{if } f[m+1]=0. \\ m-f[m+1]+1 & \text{if } f[m+1]>0. \end{cases}$$

It follows that

$$r[5]=\begin{cases} 1 & \text{if } f[m+1]=0. \\ m/(m-f[m+1]+1) & \text{if } f[m+1]>0. \end{cases}$$

By the fact that $f[m+1] \leqq m-h+1$, we can conclude that $r[5] \leqq m/h \leqq 2$.

Theorem 6.3 For a pattern included in the set P(q, m, h) $(1 \leqq h < \lceil m/2 \rceil)$,
$\lfloor n/m \rfloor \leqq$ the cost of the algorithm C $\leqq 2n$.

Proof. The best case cost is easily derived.

Let L be the total length of the substrings in the text to which the KMP algorithm is applied. We can conclude that the cost $\leqq$ 2L for those substrings. When the KMP algorithm is not working, it is shown that the maximum ratio $\leqq 2$ in a similar way.

Therefore we obtain the above result.

Example. We show the ratio $r[j]$ $(1 \leqq j \leqq 5)$ of a pattern included in the set P(q, m, h) $(\lceil m/2 \rceil) \leqq h \leqq m-1)$.

pattern	m	h	$r[1]$	$r[2]$	$r[3]$	$r[4]$	$r[5]$	max
abcdec	6	5	1/1	2/3	6/6		6/6	1/1
aaaabbbbcb	10	9	1/1	2/2	10/10		10/10	1/1
ababab cb	8	7	1/1	2/2	8/8		8/8	1/1
ababab ca	8	7	1/1	2/3	8/7		8/7	8/7
aabbccb	7	5	1/1	2/3	6/7	7/7	7/7	1/1
aabbccc	7	5	1/1	2/1	6/7	7/7	7/7	2/1
aabbcca	7	5	1/1	2/5	6/5	7/5	7/6	7/5
ababcab	7	5	1/1	2/3	6/5	7/7	7/5	7/5
abcabc	6	3	1/1	2/3	4/3	6/6	6/3	6/3
abcdabcd	8	4	1/1	2/4	5/4	8/8	8/4	8/4
abbabbcabc	10	7	1/1	2/3	8/10	10/10	10/10	1/1

We note that several patterns exist with maximum ratios of less than 2.

Table 1 Computer tests. The ratio means the average cost of our algorithm/the average cost of the BM algorithm.

q	m	BM algorithm average cost	Our algorithm average cost	ratio
2	4	10202	8480	.831
2	6	8162	8902	1.091
2	8	7225	8864	1.227
2	10	6147	8485	1.380
2	12	5874	8720	1.485
2	14	5263	8541	1.623
2	16	5041	8060	1.599
3	3	7245	6491	.896
3	6	5232	4514	.863
3	9	4394	3802	.865
3	12	3919	3502	.894
3	15	3699	3332	.901
3	18	3559	3418	.960
3	21	3282	3376	1.029
3	24	3082	3221	1.045
3	27	3057	3308	1.082
4	4	4919	4562	.927
4	8	3412	2866	.840
4	12	3100	2451	.791
4	16	2793	2234	.800
4	20	2611	2097	.803
4	24	2534	2123	.838
4	28	2521	2092	.830
4	32	2408	2087	.867
8	4	2445	3361	.976
8	8	2142	1936	.904
8	16	1579	1220	.773
8	24	1422	1019	.717
8	32	1322	930	.703
8	40	1326	893	.673
8	48	1255	860	.685
8	56	1274	889	.698
8	64	1230	866	.704
16	4	2930	2906	.992
16	8	1646	1589	.965
16	16	1024	913	.892
16	32	746	577	.773
16	48	671	481	.717
16	64	659	442	.671
32	4	2706	2700	.998
32	8	1437	1421	.989
32	16	803	771	.960
32	32	507	445	.878
32	64	364	281	.772

7. Computation of tables

In this section we will show the method of computing tables used in algorithm A, B and C. We can easily see that the time required to compute each table is linear.

Program. The tables $d[ch]$ $(ch \in \{c_1, c_2, \cdots, c_q\})$ and $g[j]$ $(0 \leqq j \leqq m)$.

```
for i:=1 to q do d[c_i]:=0;
g[0]:=0;
for j:=1 to m do begin
  g[j]:=d[pattern[j]]; d[pattern[j]]:=j;
end;
```

Program. The quantity H(pattern[1:m]).

```
h:=m;
while g[h]≠0 do h:=h−1;
```

Program. The table $gg[j]$ $(0 \leqq j \leqq m)$.

```
j:=m;
while g[j]>m−h do j:=g[j];
gg[0]:=g[j];
for j:=1 to m do gg[j]:=gg[0];
j:=m;
while g[j]>m−h do begin
  i:=d[pattern[g[j]−(m−h)]];
  if gg[i]=gg[0] then gg[i]:=g[j];
  j:=g[j];
end;
```

We assume that the table $f[j]$ $(h+1 \leqq j \leqq m+1)$ is prepared. First $w[j]$ $(1 \leqq j \leqq m-h)$ is determined. We mean by $w[j]$

$\max\{i \mid i=0$ or $(0<i \leqq j$ and pattern$[i]=$pattern$[m])\}$.

Program. The table $ff[j]$ $(h+1 \leqq j \leqq m-1)$.

```
jj:=0;
for j:=1 to m−h do begin
  if pattern[j]=pattern[m] then begin
    w[j]:=j; jj:=j;
  end else w[j]:=jj;
end;
for j:=h+1 to m−1 do begin
  if pattern[f[j]+m−j]≠pattern[m] then
    ff[j]:=w[f[j]+m−j] else ff[j]:=f[j]+m−j;
    +m−j;
end;
```

8. Computer tests

We have examined the costs of the BM algorithm and our algorithm for texts and patterns which are generated under the assumption that q possible characters appear uniformly. Computer tests have been done for q ($q=2, 3, 4, 8, 16, 32$). We have searched 100 patterns of length m in a text of length 10000. The results are shown in Table 1 and indicate that for a large q the average cost of our algorithm is less than that of the BM algorithm. We note that most parts of the text were examined at most once by the BM algorithm for a large q.

Acknowlegement

The author would like to thank Prof. Shimizu, Prof. Nozaki, Prof. Kobayashi and Prof. Noshita for their hearty encouragements. The author wish to thank the referees for carefully reading this paper and valuable advices.

References

1. Apostolico, A. and Giancarlo, R., The Boyer-Moore-Galil String Searching Strategies Revisited (to appear).
2. Boyer, R. S. and Moore, J. S., A Fast String Matching Algorithm, *Comm. ACM, 20, 10 (1977), 762-772.*
3. Galil, Z., On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm, Comm. ACM, 22, 9 (1979), 505-508.
4. Guibas, L. J. and Odlyzko, A. M., A New Proof of the Linearity of the Boyer-Moore String Searching Algorithm, *SIAM J. Computing*, 9, 4 (1980), 672-682.
5. Knuth, D. E., Morris, J. H. and Pratt, V. R., Fast Pattern Matching in Strings, *SIAM J. Computing*, 6, 2(1977), 323-350.
6. Liu C. L., Introduction to Combinatorial Mathematics.

(Received August 22, 1984; revised April 12, 1985)