The Extension of the Aho-Corasick Algorithm to Multiple Rectangular Pattern Arrays of Different Sizes and N-Dimensional Cases

RUI FENG ZHU* and TADAO TAKAOKA*

In this paper, we first show that multiple rectangular pattern arrays of various sizes can be efficiently recognized by extending the idea proposed in the AC algorithm, then demonstrate such a method permits extension to arrays of arbitrarily many dimensions. Both the running time and preprocessing time of our algorithms are proved to be linear. Possible applications are foreseen to problems such as detection of edges in digital pictures and in the field of computer graphics.

1. Introduction

As is well known, an efficient pattern matching algorithm was proposed by Aho and Corasick (AC algorithm) in [1]. A natural extension of the pattern matching problem is the one in which the pattern is a two-dimensional array $PT[1\ .\ .\ M1, 1\ .\ .\ M2]$, and the text is a second array $T[1\ .\ .\ N1, 1\ .\ .\ N2]$. In this case the problem is to determine where, if anywhere, all occurrences of the pattern array as embedded subarray in the text, that is to find all pairs (i, j) such that

$$T[i-M1+k, j-M2+l]=PT[k, l]$$

for all k and l, such that $1 \le k \le M1$ and $1 \le l \le M2$. Such a problem occurs in some methods for detecting edges in digital pictures, where a set of "edge detector" arrays is matched against sets in the picture. It also occurs in the detection of special local conditions in board games.

Karp, Miller and Rosenberg [2] have presented a method of finding all repeated occurrences of (all) square subarrays of an N by N square text array in $N*N*\log(N)$ time; In [3] Baker proposed a technique with the running time bounded by $k*N1*N2*(\log(|\Sigma|)+\log(M1))$ (Σ is the alphabet set of which the pattern and text are composed, and k is a constant independent of text, pattern and Σ); By combining the AC algorithm with KMP algorithm [5], Bird [4] (B algorithm) demonstrated an efficient algorithm for two dimensional pattern matching.

Based on the AC and B algorithms, in this paper we first present an efficient algorithm to locate all occurrences of multiple pattern arrays of various sizes in the text, then demonstrate such algorithm permits extension to arrays of arbitrarily many dimensions.

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

2. The AC and B Algorithms

Let keywords $K=\{PT_1, PT_2, \ldots PT_k\}$ be a finite set of strings and text T be an arbitrary string. The problem is to locate and identify all substrings of T which are keywords in K. Substring may overlap with one another.

The AC algorithm first construct from the set of keywords a finite state pattern matching machine, then apply the text string as input to the machine. The behavior of the pattern matching machine is dictated by three functions: a goto function g, a failure function f, and a output function output. The details of algorithms for constructions the three functions are given in [1].

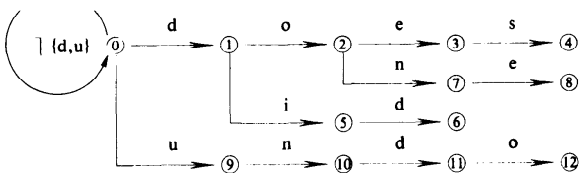
Figure 1 shows an example of the three functions for the set of keywords {do, does, did, done, undo}.

The goto function g maps a pair consisting of a state and an input symbol into a state or the message fail. The directed graph in Figure 1(a) represents the goto function. For example, the edge labeled d from 0 to 1 indicates that $g(0, d)=1$. The absence of an arrow indicates fail. Thus, $g(1, \delta)=$fail for all input symbols δ that are neither o nor i. The pattern matching machine has the property that $g(0, \delta) \neq$ fail for all input symbols δ.

The failure function f maps a state into a state. The failure function is consulted whenever the goto function reports fail. Certain states are designated as output states which indicates that a set of keywords has been found. The output function formalizes this concept by associating a set of keywords (possibly empty) with every state.

The B algorithm is designed to match a two dimensional pattern with a two dimensional text. At the row-matching stage, it takes each row of the pattern as a

Example 1.



(a) Goto function

I	1	2	3	4	5	6	7	8	9	10	11	12
f(I)	0	0	0	0	0	1	0	0	0	0	1	2

(b) Failure function

I	output (I)
2	{do}
4	{does}
6	{did}
8	{done}
12	{undo, do}

(c) output function

Fig. 1 Pattern Matching Machine.

keyword, and then constructs from the set of such keywords a finite state matching machine to match with the text row by row. At column-matching stage, it checkes whether or not each row of the pattern appears in the text sequentially. For details of the B algorithm, refer to [4].

3. Our Algorithm

When the patterns are rectangular arrays of various sizes, that is

$$K=\{PT_i[1\ ..\ P_{i,1}, 1\ ..\ P_{i,2}] \mid i=1, 2, \ldots k\}$$

the text is also a rectangular array $T[1\ ..\ N1, 1\ ..\ N2]$. The problem becomes to determine all occurrences of any pattern array PT_i as embedded subarray in the text array.

The general scheme of the algorithm is composed of two distinct parts, a row-matching and a column-matching.

3.1 Row-Matching

The purpose of the row-matching step is to determine which row of the patterns matches a terminal substring of text T. This can be done using the AC algorithm in which each row of PT_i is considered as a keyword. We assume that $P_{1,2} \geq P_{2,2} \geq P_{3,2} \geq \ldots \geq P_{k,2}$, (Here $P_{i,2\,i=1,2,\ldots,k}$ is row length for each pattern PT_i), and make a partition according to the following:

Put PT_i and PT_{i+1} in the same set

$$\text{iff } P_{i,2}=P_{i+1,2}. \quad (i=1, 2, \ldots k-1)$$

the size of such a partition is assumed to be $k1 (k1 \leq k)$. We define an array PARTITION [1 . . k] of 1 . . $k1$ to partition patterns according to their row length. The computation of the PARTITION is:

Algorithm 1.

Input. each row length of the patterns $P_{1,2}, P_{2,2}, \ldots P_{k,2}$.

Output. partition of pattern row length.

Methods.

```
begin
  j:=1
  for i:=1 to k do
    begin
    PARTITION[i]:=j
    if i<k and P_{i,2}≠P_{i+1,2} then j:=j+1
    end
end.
```

The component rows of patterns within the same partition are all of the same length. It follows that no two component rows may be proper suffixes in the same partition. So, for each position in a row of the text, at most one distinct row of given length of patterns may match in that location. (If two rows of the same length match at one location, they must be identical.)

We identify the distinct rows of patterns and assign each a unique index. Let the distinct rows be $X_1, X_2, \ldots X_q$. Thus pattern PT_i can be represented by the column $(p(i, 1), p(i, 2), \ldots p(i, P_{i,1}))'$, whose elements are in $\{1, 2, \ldots q\}$ such that

$$PT_i = \begin{matrix} X_{p(i,1)} \\ X_{p(i,2)} \\ \vdots \\ X_{p(i,P_{i,1}))} \end{matrix}$$

In our algorithm, we put such a unique index in output function output instead of its corresponding row.

Example 2.

	$k=5$	Partition $k1=4$	index
PT_1:	$a\ a\ b\ b\ a=X_1$	1	$p(1, 1)=1$
	$a\ a\ a\ a\ b=X_2$	1	$p(1, 2)=2$
PT_2:	$a\ a\ a=X_3$	2	$p(2, 1)=3$
	$b\ b\ b=X_4$	2	$p(2, 2)=4$
	$a\ a\ a=X_3$	2	$p(2, 3)=3$
PT_3:	$a\ a\ a=X_3$	2	$p(3, 1)=3$
PT_4:	$a\ b=X_5$	3	$p(4, 1)=5$
	$a\ a=X_6$	3	$p(4, 2)=6$
PT_5:	$a=X_7$	4	$p(5, 1)=7$

Also we add to the output function output in our algorithm another value to retain the partition value of the row found (In fact the partition value of a row is equal to the partition value of the pattern in which the row locates). The modified algorithm for construction

goto function $g1$ is as follows:

Algorithm 2.

Input. Set of keywords

$$K=\{PT_i[j, 1]\ PT_i[j, 2]\ \ldots\ PT_i[j, P_{i,2}] \mid j=1, 2, \ldots P_{i,1},\ i=1, 2, \ldots k\}$$

Output. Goto function $g1$ and a partially computed output function output1.

Method. We assume output1(l, s) (l=1, 2) is initially empty.

```
begin
newstate:=0
for i:=1 until k do
for j:=1 until P_{i,1} do
  enter (i, j, PT_i[j, 1] PT_i[j, 2] . . . PTi[j, P_{i,2}])
for all a such that g1(0, a)=fail do g1(0, a):=0
end
Procedure enter (i1, j1, A_1 A_2 . . . A_m)
begin
  state:=0; t:=1;
  while g1(state, A_t)≠fail do
    begin
      state:=g1(state, A_t);
      t:=t+1
    end
  for p:=t until m do
    begin
      newstate:=newstate+1;
      g1(state, A_p):=newstate;
      state:=newstate
    end
  output1(1, state):=p(i1, j1);
  output1(2, state):=partition(i1)
end.
```

The computation of failure function $f1$ can be made by directly using the corresponding algorithm in [1].

The Goto function $g1$, failure function $f1$, output function output1 for example 2 are shown in Fig. 2.

The modified pattern matching machine algorithm is:

Algorithm 3.

Input. A text array $T[1\ .\ .\ N1, 1\ .\ .\ N2]$ of string and a pattern matching machine $M1$ with Goto function $g1$, failure functions $f1$ and output function output1.

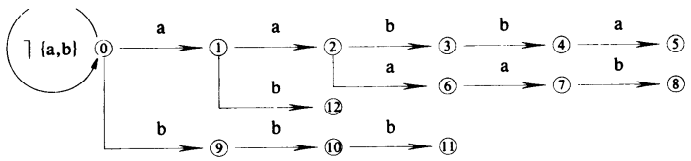
Output. locations (row, column) at which PT_i occurs in text.

Method.

```
 0 begin
 1 state1:=0;
 2 for row:=1 until N1 do
 3 for column:=1 until N2 do
 4 begin
 5 while g1(state1, T[row, column])=fail do
     state1:=f1(state1);
 6 state1:=g1(state1, T[row, column]);
 7 if output1(1, state1)≠empty then
 8 print (output1(1, state1), output1(2, state1))
 9 end
10 end.
```

3.2 Column-Matching

Discovering that the yth row of pattern PT_i occurs in a given place in the text only means that the yth row of PT_i exist in the text, so we must investigate whether or not the rows from the 1st row until $(y-1)$th row of PT_i occur immediately above the yth row in order to determine if PT_i occurs in the text. This can be done by using



(a) Goto function $g1$

i	1	2	3	4	5	6	7	8	9	10	11	12
$f1(i)$	0	1	12	10	1	2	6	3	0	9	10	9

(b) failure function $f1$

state	(output1(1, state), output1(2, state))
1	[(7,4)]
2	[(6,3),(7,4)]
3	[(5,3)]
5	[(1,1),(7,4)]
6	[(7,4),(6,3),(3,2)]
7	[(7,4),(6,3),(3,2)]
8	[(2,1),(5,3)]
11	[(4,2)]
12	[(5,3)]

(c) output function output1

Fig. 2

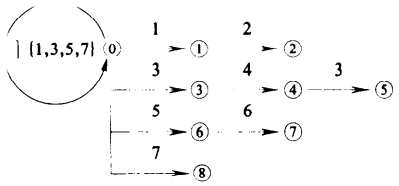
the pattern matching machine again.

We construct another pattern matching machine $M2$ with Goto function $g2$, failure function $f2$, and output function output2. The input keywords are:

$$KK=\{p(i,1)\ p(i,2)\ldots p(i,P_{i,1})\mid i=1,2,\ldots,k\}$$

The Goto function $g2$, failure function $f2$, and output function output2 for example 1 are shown in Fig. 3.

We maintain a rectangular array $a[1\ldots k1, 1\ldots N2]$ of states, such that for each step (row, column) the fact that $a[k,\ \text{column}]=s$ (assume string $X_1X_2\ldots X_u$ represents state s of pattern matching machine $M2$ if the shortest path in the goto graph of $M2$ from the start state to state s spells out $X_1X_2\ldots X_u$) means just that $X_1X_2\ldots X_u$ of pattern PT_i with partition value k have been discovered to match the text in positions:



(a) goto function $g2$

i	$f2(i)$
1	0
2	0
3	0
4	0
5	3
6	0
7	0
8	0

(b) failure function $f2$

state	output2
2	$\{PT_1\}$
3	$\{PT_3\}$
5	$\{PT_2, PT_3\}$
7	$\{PT_4\}$
8	$\{PT5\}$

(c) output function output2

Fig. 3

$$\begin{array}{ll} T[\text{row}-u+1, \text{column}-P_{i,2}+1], \ldots T[\text{row}-u+1, \text{column}] & X_1 \\ T[\text{row}-u, \text{column}-P_{i,2}+1], \ldots\ldots T[\text{row}-u, \text{column}] & X_2 \\ \vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots & \\ T[\text{row}, \text{column}-P_{i,2}], \ldots\ldots\ldots T[\text{row}, \text{column}] & X_u \end{array}$$

if we have output2(s)$\neq$empty that means a complete pattern PT_i has been found as a subarray of the text at (row, column).

The algorithm is the following code:

Algorithm 4.

Input. A text array $T[1\ldots N1, 1\ldots N2]$ of string and a pattern matching machine $M1$ with Goto function $g1$, failure function $f1$ and output function output1.

Output. locations (row, column) at which PT_i occurs in text.

Method.

```
 0  begin
 1  state1:=0;
 2  for row:=1 until N1 do
 3  for column:=1 until N2 do
 4  begin
 5  while g1(state1, T[row, column])=fail do
    state1:=f1(state1);
 6  state1:=g1(state1, T[row, column]);
 7  if output1(1, state1)≠empty
 8  then for each output1(1, state1)≠empty do
 9   begin
10    state2:=a[output1(2, state1), column];
11    c:=output1(1, state1);
12    while g2(state2, c)=fail do state2:=f2(state2);
13    state2:=g2(state2, c);
14    a[output1(2, state), column]:=state2;
15    if output2(state2)≠empty then
16      begin
17        print (row, column);
18        print (output2(state2))
19      end
20    end
21  end
22  end.
```

The number of output1(1, state1) might be more than 1, but always less than $k1$. For array a to work correctly, at the begining of the algorithm, we initialize $a[t,$ column$]=0$ $1\leq t\leq k1$, and $1\leq$column$\leq N2$, that is, each element of array a bigins from start state.

3.3 The Time Complexity of Our Algorithm

According to the theorems in [1], the complexity of pattern preprocess:

- The computation of Goto function $g1$, failure function $f1$, and output function output1 for $M1$ takes time proportional to $P_{1,1}*P_{1,2}+P_{2,1}*P_{2,2}+\ldots+P_{k,1}*P_{k,2}$, that is, the sum of the pattern sizes.
- The computation of Goto function $g2$, failure function $f2$, and output function output2 for $M2$ takes time proportional to $P_{1,1}+P_{2,1}+\ldots+P_{k,1}$ that is the sum of the pattern column lengths.

Since the row lengths of patterns vary, perhaps at one position in text, we may get more than one value of output function output1. So the time complexity depends on how often rows of the patterns occur in the text, for average cases the running time is $O(N1*N2)$ (the number of state transpotation). Theoretically in (unbelievable) worst case it might be $O(k1*N1*N2)$, that is at each position in the text for each different row length one row (in fact at most one row) is found matching. But in practice the algorithm can be relied upon to take about $O(N1*N2)$ steps. The only additional space requirement of array a is $k1*N2$. So the space complexity of our Algorithm is still proportional to the size of the text, assuming $k1\leq N1$.

4. The Extension of Our Algorithm

In section 3, if $P_{1,2}=P_{2,2}=\ldots=P_{k,2}$, that is, all patterns have a common row length ($k1=1$), and $P_{1,1}=P_{2,1}=\ldots=P_{k,1}$, we may consider the set of patterns as a three dimensional array $PT[1\ldots P1, 1\ldots P2, 1\ldots P3]$, and also text is a three dimensional array $T[1\ldots N1, 1\ldots N2, 1\ldots N3]$. The problem becomes to find all occurrences of PT as embedded subarray in text T.

The algorithm has three matching stages. We will discuss them from 4.1 to 4.3.

4.1 First Matching Stage

The first matching stage is to determine which row of pattern PT matches a substring in the text. We construct a pattern matching machine $M1$ with goto function $g1$, failure function $f1$, and output function output1 from the keywords $K1$ as described in following:

$$K_1=\{PT[i,j,1]\ PT[i,j,2]\ldots PT[i,j,P3] \mid i=1,2\ldots P1;\ j=1,2\ldots P2\}$$

and in output1 we put a unique character of Σ^1 for each distinct keyword (Σ^1 is a set, each element of Σ^1 is corresponding to a unique keyword of $K_{i,i=1,2,\ldots N}$). From such unique character of Σ^1, we construct a two-dimensional pattern PT' which is used for the second matching stage.

Example 3:

Pattern $PT[1\ldots2, 1\ldots3, 1\ldots3]$ and unique character for each distinct keyword:

$PT[1\ 1\ 1]=a$	$PT[1\ 2\ 1]=a$	$PT[1\ 3\ 1]=b$
$PT[1\ 1\ 2]=a$	$PT[1\ 2\ 2]=b$	$PT[1\ 3\ 2]=b$
$PT[1\ 1\ 3]=a$	$PT[1\ 2\ 3]=b$	$PT[1\ 3\ 3]=b$
A	B	C

$PT[2\ 1\ 1]=a$	$PT[2\ 2\ 1]=a$	$PT[2\ 3\ 1]=a$
$PT[2\ 1\ 2]=a$	$PT[2\ 2\ 2]=a$	$PT[2\ 3\ 2]=b$
$PT[2\ 1\ 3]=a$	$PT[2\ 2\ 3]=b$	$PT[2\ 3\ 3]=b$
A	D	B

So we have $PT'[1\ldots2, 1\ldots3]$:

	1	2	3
1	A	B	C
2	A	D	B

4.2 Second Matching Stage

The second stage is designed to find which row of pattern PT' matches a subarray in the text. Also using algorithm 2 we construct another pattern matching machine $M2$ with goto function $g2$, failure function $f2$, and output function output2 from keywords K_2 described below:

$$K_2=\{PT'[i,1]PT'[i,2]\ldots PT'[i,P2] \mid i=1,2,\ldots,P_1\}$$

and in output2 we put a unique character of Σ^2 for each distinct row of PT'. From Σ^2 we construct an array PT'' which is then used for the third matching stage. In the case of example 3, such PT'' is:

$PT'[1\ldots2, 1\ldots3]$:	$PT''[1\ldots2]$:
ABC	X
ADB	Y

4.3 Third Matching Stage

We use algorithm 2 to construct $M3$ with Goto function $g3$, failure function $f3$, and output function output3. The keywords K_3 is defined as following:

$$K_3=\{PT''[1]\ PT''[2]\ldots PT''[P1]\}$$

The following is modified algorithm 2:

Algorithm 5

```
1  begin
2    state1:=0; state3:=0;
3    for i1:=1 to N1 do
4    for i2:=1 to N2 do
5    for i3:=1 to N3 do
6      begin
7        while   g1(state1, T[i1, i2, i3])=fail   do
           state1:=f1(state1);
8        state1:=g1(state1, T[i1, i2, i3]);
9        if output1(state1)≠empty
10       then begin
11         state2:=a[i1];
12         while g2(state2, output1(state1))=fail do
             state2:=f2(state2);
13         state2:=g2(state2, output1(state1));
14         a[i1]:=state2;
15         if output2(state2)≠empty
16           then begin
17             while g3(state3, output2(state2))=
                 fail do state3:=f3(state3);
18           state3:=g3(state3, output2(state2));
19           if output3(state3)≠empty
20           then print('found at', i1, i2, i3)
21             end
22       end
23 end.
```

In the above algorithm an array a of size [1 . . $N1$] is used to store states for goto function $g2$ at the second matching stage. The function of array a is much like that defined in section 3.2.

The time complexity of first matching stage and second matching stage are bounded by $O(P1*P2*P3)$ and $O(P1*P2)$ respectively, according to the theorems in [1].

THEOREM. The running time of algorithm 5 is $O(N1*N2*N3)$.

PROVE: Obviously the running time of algorithm 5 is

$O(N1*N2*N3)$ in average, The worst case occurs when at line 9 output1(state1)≠empty is always true, so the loop 10–22 is carried out. At line 13 we make a state transition according to goto function $g2$, this makes the depth of state2 increase one step in goto graph of $g2$, we assume the total number of such increase of depth is K', and $K' \leq N1*N2*N3$. At line 12 failure function $f2$ is used when the goto function $g2$ reports fail, and this makes the depth of state2 decrease, we assume the number of such cases is $K1'$. However the steps of depth increase is equal or more than that of decrease, that is:

$K' \geq K1'*L1$ ($L1$ is the average steps made at line 12) and the total steps made at line 12 and 13 is: $K'+K1'*L$, so the average steps L made at line 12 and 13 for each operating cycle is:

$$L=(K'+K1'*L1)/(N1*N2*N3)$$
$$\leq(2*K')/(N1*N2*N3)\leq 2$$

Line 16–21 in algorithm 5 can also be analysed in the same way.

#

Including all the relevant tables and the array a, the space requirement is still bounded by $O(N1*N2*N3)$.

4.4 The N-Dimensional Case

Obviously we can extend our algorithm to N-dimensional array problem that is the pattern is an N-dimensional array $PT[1\ldots P_1, 1\ldots P_2, \ldots, 1\ldots P_N]$, and the text is also an N-dimensional array $T[1\ldots N_1, 1\ldots N_2, \ldots, 1\ldots N_N]$.

We have $N-1$ stages for the pattern preprocess, and implement each stage in the following way:

BEGIN
  $PT_1:=PT$;
  for $j:=1$ to $N-1$ do
  begin
1. Construct goto function g_j, failure function f_j, and output function output$_j$ from keywords set K_j. The definition of K_j is:

$$K_j=\{PT_j[i_1, i_2, \ldots, i_{N-j}, 1], PT_j[i_1, i_2, \ldots, i_{N-j}, 2], \ldots, PT_j[i_1, i_2, \ldots, i_{N-j}, P_{N-j+1}] \mid i_1=1, 2, \ldots, P_1;\ i_2=1, 2, \ldots, P_2;\ \ldots;\ i_{N-j}=1, 2, \ldots P_{N-j}\};$$

2. Identify distinct keywords in K_j and assign each a unique chracter Z of Σ^{j+1}, and put Z into output j;
3. Construct a $N-j$ dimensional array PT_{j+1} from Z of Σ^{j+1} for $j+1$ stage
  END.

The pattern preprocess takes time proportional to:

$$T=P_1*P_2*\ldots*P_N+P_1*P_2*\ldots*P_{N-1}+P_1*P_2+P_1$$
$$=P_1(1+P_2(1+\ldots(P_{N-1}(1+P_N))\ldots))$$

as $P_i>1$ $(i=1, 2, \ldots, N)$, we may consider $T=t*(P_1*P_2*\ldots*P_N)$, t is a constant.

The behavior of pattern matching machine is:

Algorithm N;
begin
  state$_1$:=0; state$_N$:=0;
  for i_1:=1 to N_1 do
  for i_2:=1 to N_2 do
    . . .
  for i_N:=1 to N_N do
begin
while g_1(state$_1$, $T[i_1, i_2, \ldots, i_N]$)=fail do
state$_1$:=f_1(state$_1$);
state$_1$:=g_1(state$_1$, $T[i_1, i_2, \ldots, i_N]$);
if output$_1$(state$_1$)≠empty
  then begin state$_2$:=$a_2[i_1, i_2, \ldots, i_{N-2}]$;
  while g_2(state$_2$, output$_1$[state$_1$])=fail do
state$_2$:=f_2(state$_2$);
  state$_2$:=g_2(state$_2$, output$_1$(state$_1$));
  $a_2[i_1, i_2, \ldots, i_{N-2}]$:=state$_2$;
  if output$_2$(state$_2$)≠empty then
   begin
      . . .
    state$_{N-1}$:=$a_{N-1}[i_1]$
    while g_{N-1}(state$_{N-1}$, output$_{N-2}$(state$_{N-2}$))=fail do
    state$_{N-1}$:=f_{N-1}(state$_{N-1}$);
    state$_{N-1}$:=g_{N-1}(state$_{N-1}$, output$_{N-2}$(state$_{N-2}$));
    if output(state$_{N-1}$)≠empty then
      begin
        while g_N[state$_N$, output$_{N-1}$[state$_{N-1}$]]=fail do
        state$_N$:=f_N(state$_N$);
        state$_N$:=g_N(state$_N$, output$_{N-1}$(state$_{N-1}$));
        if output$_N$(state$_N$)≠empty
         then print('found at', $i_1, i_2, \ldots, i_N$)
      end
    end
  . . .
  end
end.

The additional arrays $a_2[1\ldots N_1, \ldots, 1\ldots N_{N-2}]$, $a_3[1\ldots N_1, \ldots, 1\ldots N_{N-3}]$, $\ldots$, $a_{N-1}[1\ldots N_1]$ are initialized in the same way as described in section 3.2.

5. Conclusion

In this paper, we have demonstrated that multiple rectangular pattern arrays of various sizes and N-dimensional arrays can be efficiently recognized by using the method proposed in AC algorithm [1]. The algorithms described here possess the following noteworthy properties:

(1) Both its running time and space requirement are linearly proportional to the size of text, which is nearly optimal within constant factor (see [7]).
(2) The on-line nature, that is the input is scanned only once, and after scanning the character at any position of the input, before scanning further, it is possible to answer yes or no to whether any of the patterns match at that position.

(3) its extension to any dimensional cases makes it very useful in fields such as computer graphics, picture processing, etc.

Acknowledgement

The authors are grateful to Mr. Tamaki, Department of Information Science, Ibaraki University for his helpful discussions and anonymous referees for their constructive comments.

References

1. AHO, A. V. and CORASICK, M. J. Efficient string matching, An aid to bibliographic search, *Comm. ACM* 18(6) (1975), 333–340.
2. KARP, R. M., MILLER, R. E. and ROSENBERG, A. L. Rapid identification of repeated patterns in string, tree and array, Proc. of *the 4th Annual ACM Symposium on Theory of Comput.* (1972), Assoc. for Comput. Mach, New York, 125–136.
3. BAKER, T. P. A technique for extending rapid exact-match string matching to arrays of more than one dimension, *SIAM Journal on computing* 7(4) (1978), 533–541.
4. BIRD, R. S. Two dimensional pattern matching, *Information Processing Lett.* 6(5) (1977), 168–170.
5. KNUTH, D. E., MORRIS, J. H. and PRATT, V. R. Fast pattern matching in strings *SIAM journal on computing*, 6(2) (1977), 323–350.
6. AHO, A. V., HOPCROFT, J. E. and ULLMAN, J. D. The Design and Analysis of Computer Algorithm, Addison-Wesley, Reading, MA, (1974).
7. RIVEST, R. L. On the worst case of string-searching algorithm, *SIAM Journal on Computing* 6 (1977), 669–674.

(Received September 24, 1987; reviced May 12, 1988)