PATTERN MATCHING MACHINES FOR REPLACING SEVERAL CHARACTER STRINGS

Arikawa, Setsuo
Research Institute of Fundamental Information Science, Kyushu University

Shiraishi, Shuji
Department of Information Systems, Intedisciplinary Graduate School of Engineering Sciences, Kyushu University

https://doi.org/10.5109/13361

出版情報：Bulletin of informatics and cybernetics. 21 (1/2), pp.101-111, 1984-03. Research Association of Statistical Sciences
バージョン：
権利関係：

Bulletin of Informatics and Cybernetics Vol. 21, No. 1~2, 1984

PATTERN MATCHING MACHINES FOR REPLACING SEVERAL CHARACTER STRINGS

By

Setsuo ARIKAWA* and Shuji SHIRAISHI**

Abstract

This paper presents a pattern matching machine which detects all occurrences of the longest possible keywords in a text and replaces them with the corresponding keywords. The pattern matching machine of this type is a generalized sequential machine and is constructed in nearly the same way as Aho-Corasick's pattern matching machine to locate all occurrences of keywords. We show algorithms to construct our pattern matching machine and to make the machine run on a given text string, and show the validities of them. We also consider the time complexity of the algorithms and evaluate the running time of the algorithms. Finally we discuss some applications of our pattern matching machines.

1. Introduction

In text-editing we often need to replace some strings with other strings. Usually this job is carried out by repeated uses of an edit command

$$CHANGE/x/y/ALL,$$

or something like that, which works to change all occurrences of the string x in the text by the corresponding string y. Thus if we have n pairs of (x, y) we must use the command n times ($2n-1$ times, in the worst case) and the computer also must scan the text n times ($2n-1$ times).

We present in this paper a new pattern matching machine to do such n to $2n-1$ replacements in just one scanning the text. More precisely the new pattern matching machine detects all occurrences of the longest keywords in a text and replaces them with the corresponding keywords.

As concerns the pattern matching of strings, three approaches by Knuth-Morris-Pratt [1], Boyer-Moore [2] and Aho-Corasick [3] are widely known. These are all merely to detect all occurrences of keywords in a text. The first two approaches are for a single keyword, and the third one is for a finite set of keywords, and is extensively used in a practical system [4, 5]. Hence we base our approach on the third one.

* Research Institute of Fundamental Information Science, Kyushu University 33, Fukuoka 812, Japan.

** Department of Information Systems, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University 39, Kasuga, Fukuoka 816, Japan.

In our pattern matching machine occurs a new problem to be solved. Keywords may overlap with one another and in the worst case the overlaps may propagate up to the end of text. We avoid this propagation by finding the longest possible keyword from the left.

We make the problem clearer in Section 2. Our pattern matching machine consists of three functions of goto, failure and output like Aho-Corasick's machine. The algorithms to construct these functions and to make the constructed machine run on texts are given in Section 3. The validity and complexity of these algorithms are discussed in Sections 4 and 5. An optimization of the algorithms is discussed in Section 6. Applications of our pattern matching machine are briefly referred to in the final section.

2. Problem Description

By a *keyword* or a *text* we mean a finite string of characters. Let

$$K=\{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$

be a finite set of keyword pairs, where x_i is non-empty but y_j is possibly empty. Then our problem is to find an efficient algorithm which searches a text z from left to right for the longest possible keyword x_i in

$$K_X=\{x_i|(x_i, y_i) \text{ in } K\}$$

and replaces it with the paired keyword y_i, and repeats this process from the character next to the detected keywords x_i until the text z is read through. More precisely the algorithm works at the search stage as follows. Even if it has detected a keyword x_i in the text z, it tries to find a longer keyword in K which contains x_i as a substring until any longer keyword than the longest one so far detected is no longer expected in z.

EXAMPLE 1. Let us consider the situations in Fig. 1, where the line denotes a text and $x_1, \cdots, x_4$ are keywords in K_X. The algorithm we are constructing detects, for example, a keyword x_1 at point Z_1'. However the x_1 is contained in x_2. Hence if the text is $[OC]$, then the algorithm replaces the occurrence of x_2 with the corresponding y_2 when it has read character next to Z_2'. If the text is $[OF]$, then it replaces x_4 with y_4. Thus we can summarize these input-output relation on the texts in Table 1, where by $[OZ_i)$ we mean a substring starting at O and ending just before Z_i, and $(Z_2'Z_3)$ and $(Z_3'E]$ are analogous to the notations for intervals of real numbers.

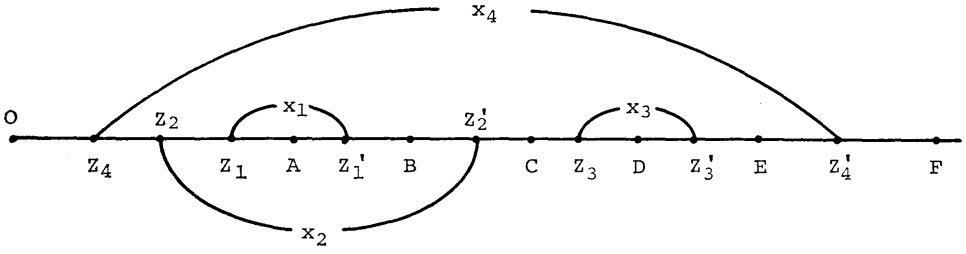


Fig. 1 Texts and keywords

Table 1. Input texts and output texts

input text	output text
$[OA]$	$[OA]$
$[OB]$	$[OZ_1)y_1(Z_1'B]$
$[OC]$	$[OZ_2)y_2(Z_2'C]$
$[OD]$	$[OZ_2)y_2(Z_2'D]$
$[OE]$	$[OZ_2)y_2(Z_2'Z_3)y_3(Z_3'E]$
$[OF]$	$[OZ_4)y_4(Z_4'F]$

3. Algorithms

We now realize the algorithm to solve our problem as a pattern matching machine of Aho-Corasick type. Our pattern matching machine for replacing character strings (*rpmm* for short) consists, like Aho-Corasick's machine, of a goto function g, failure function f and output function *output*. We denote the *rpmm* by $M=(K, g, f, output)$. Thus we need two main algorithms; one is for constructing *rpmm* and the other is for making the *rpmm* run on texts.

First we show Algorithm 1 to construct goto functions and partially computed output functions. This algorithm is the same as Aho-Corasick's one except the assignment of values to output functions.

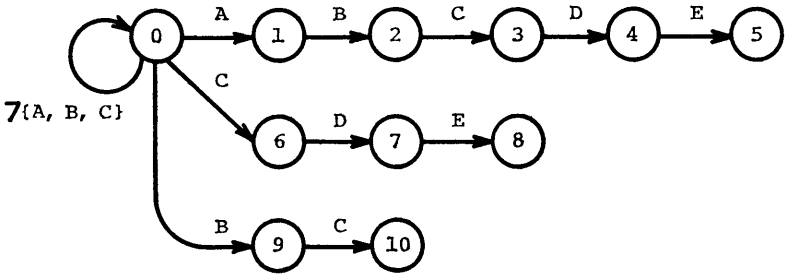
Algorithm 1. (Construction of the goto function)

Input. Set of keyword pairs $K=\{(x_1, y_1), \cdots, (x_k, y_k)\}$.

Output. Goto function g and a partially computed output function *output*.

Method.

```
begin
  newstate:=0
  for i:=1until k do enter(x_i, y_i)
  for all a such that g(0, a)=fail do g(0, a):=0
end
procedure enter(a_1a_2 ··· a_m, y)
begin
    state:=0; j:=1
  while g(state, a_j)= fail do
      begin
        state:=g(state, a_j)
        j:=j+1
      end
    for p:=j until m do
      begin
        newstate:=newstate+1
        g(state, a_p):=newstate
        state:=newstate
      end
    output(state):=y
end
```



a) goto function

s	output(s)
0	
1	
2	
3	
4	
5	α
6	
7	
8	β
9	
10	γ

b) output function

Fig. 2 Goto function and output function by Algorithm 1 for input $K=\{(ABCDE, \alpha), (CDE, \beta), (BC, \gamma)\}$

EXAMPLE 2. For an input $K=\{(ABCDE, \alpha), (CDE, \beta), (BC, \gamma)\}$, Algorithm 1 produces a goto function in Fig. 1(a) and a partially computed output function in Fig. 2(b), where the $7\{A, B, C\}$ denotes any character not in $\{A, B, C\}$.

The output function will be made total by the following algorithm, main work of which is to compute failure function.

Algorithm 2. (Construction of the failure function and completion of the output function).

Input. Goto function g and output function *output* from Algorithm 1.

Output. Failure function f and output function *output*.

Method.

begin

queue := *empty*

for *each a such that* $g(0, a)=s\neq 0$ **do**

begin

```
        queue := queue.s
        f(s) := 0
        if output(s) is undefined then output(s) := a
      end
    while queue≠empty do
      begin
        let queue=r.tail
        queue := tail
        for each a such that g(r, a)=s≠fail do
          begin
            queue := queue.s
            if output(s) is defined then f(s) := 0 else
              begin
                state := f(r)
                output(s) := output(r)
                while g(state, a)=fail do
                  begin
                    state := f(state)
                    output(s) := output(s).output(state)
                  end
                f(s) := g(state, a)
                if f(s)=0 then output(s) := output(s).a
              end
          end
      end
  end
```

EXAMPLE 3. Algorithm 2 receives the goto function and partial output function of Fig. 2, say, and then it produces a failure function and computes the output function as in Fig. 3. The broken arrows in the figure mean failure transitions and broken arrows to the states 0 from all the states but the states 0, 2 and 3 should be added. Note that the undefined value for *output*(0) will be assigned when the *rpmm* runs on texts.

Now we give the algorithm to make the *rpmm* run on texts.

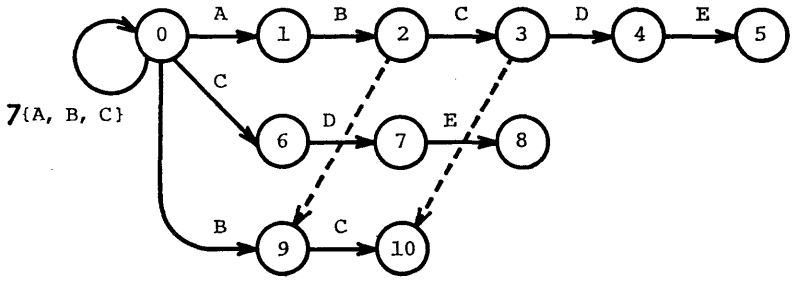
Algorithm 3. (Pattern matching machine for replacing strings)

Input. A text $z=a_1a_2\cdots a_n$ and a pattern matching machine with functions g, f and *output*.

Output. A replaced text string $w=b_1b_2\cdots b_t$.

Method.

```
  begin
    state := 0
    for i := 1 until n do
      begin
        while g(state, a_i)=fail do
          begin
            print output(state)
```



a) Goto function and failure function

s	output(s)
0	undefined
1	A
2	A (output(1))
3	A
4	AγD (output(3)·output(10)·a)
5	α
6	C
7	CD (output(6)·a)
8	β
9	B
10	γ

b) Output function

Fig. 3 Completion of *rpmm* by Algorithm 2

```
        state := f(state)
      end
    state := g(state, a_i)
    if state=0 then print a_i
  end
while state≠0 do
  begin
    print output(state)
    state := f(state)
  end
end
```

The *rpmm* $M=(K, g, f, output)$ prints outputs whenever it changes states by failure transitions. When it reads through the text, it makes a series of failure transitions from the current state back to the initial state 0 printing the corresponding outputs. Note that Algorithm 3 prints the input character a when the *rpmm* makes a transition

by $g(0, a)=0$.

EXAMPLE 4. The *rpmm* in Fig. 3 behaves on a text "*DEABCCBCE*" as in Fig. 4. When the text is read through we get an output text $DEA\gamma C\gamma E$ by concatenating the outputs produced at failure transitions.

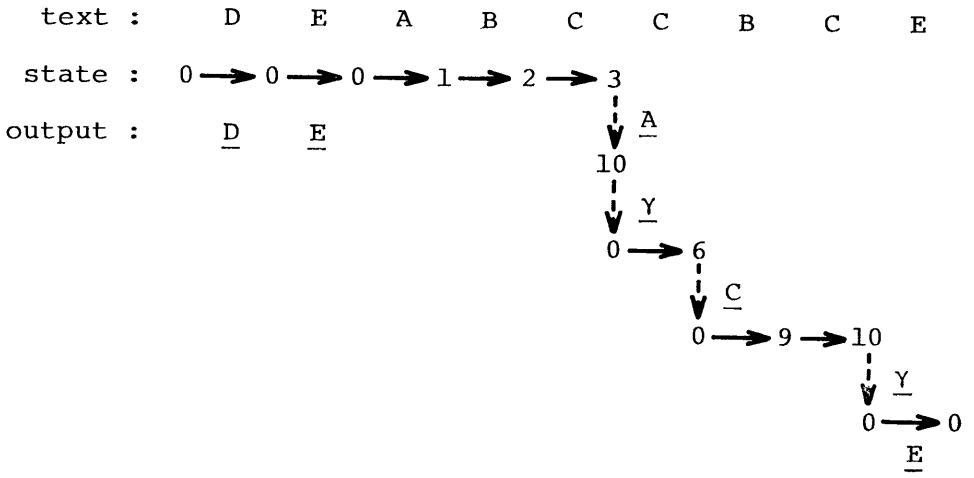


Fig. 4 Behavior of *rpmm*

4. Validity of Algorithms

We can say that *rpmm* $M=(K, g, f, output)$ constructed by Algorithm 1 and 2 is valid if M replaces text with a desired output text by using Algorithm 3. Formal definition follows. For our *rpmm* $M=(K, g, f, output)$ and an input text w, let $M(w)$ be the output string by Algorithm 3.

DEFINITION. For any text w, let u be the shortest string such that

$$w=uxv \text{ and } x \text{ is in } K_X$$

if there is such an x, and then let x be the longest string satisfying the condition above for the u. If there is no such x, then let u be the initial character of w and x be the empty string ε. Then an *rpmm* M is said to be *valid* if

$$M(w)=uK(x)M(v),$$

where $K(x)=y$ for (x, y) in K and $K(\varepsilon)=\varepsilon$.

In order to show the validity of our *rpmm*'s we first characterize the output function. Let $rep(s)$ be the string spelled out by the shortest path from the initial state 0 to the state s in the goto graph. Then by Algorithm 1,

$$output(s)=y,$$

for any keyword pair (x, y) and $x=rep(s)$.

For the other states, the output function is characterized by the failure function. So here we state the property of the failure function.

LEMMA 1 (Aho-Corasick). *Let* $f(s)=t$, $rep(s)=a_1a_2\cdots a_m$ *and* $rep(t)=b_1b_2\cdots b_n$.

Then $b_1b_2\cdots b_n$ is the longest proper suffix of $a_1a_2\cdots a_m$ and also a prefix of some keyword.

Algorithm 2 characterizes the output function as follows:

LEMMA 2. *Let $f(s)=t$, $rep(s)=a_1a_2\cdots a_m$, $rep(t)=a_ia_{i+1}\cdots a_m$ and $rep(u)=a_1a_2\cdots a_{i-1}$ (When $t=0$, $rep(t)=\varepsilon$ and $rep(s)=rep(u)$). Then,*

$$output(s)=output(u).$$

PROOF. The proof is done by induction on the depth of states s

Basis: Suppose $depth(s)=1$. Then by Algorithm 2, $f(s)=t=0$. Since $rep(s)=a$, $rep(t)=\varepsilon$ and $rep(u)=a$, we have

$$output(s)=a=output(u).$$

Induction step: Assume the lemma holds for $depth(s)<m$. Let $depth(s)=m$ and $f(s)=t$. Then by Lemma 1,

$$rep(s)=a_1a_2\cdots a_m,$$

$$rep(t)=a_ia_{i+1}\cdots a_m, \qquad (1\leqq i\leqq m+1).$$

Algorithm 2 defines the failure function for state s of $depth(s)\geqq 2$ and the output function as follows:

First the failure function $f(s)=t$ is defined by:

$$g(r_0,\ a_m)=s,$$

$$r_j=f(r_{j-1});$$

$$g(r_j,\ a_m)=fail \qquad (1\leqq j\leqq k),$$

$$g(r_k,\ a_m)=t.$$

Then the output function $output(s)$ is defined by:

$$f(s)=t$$

iff

$$output(s)=\begin{cases} output(r_0)\cdots output(r_{k-1}).a_m & if\ t=0,\\ output(r_0)\cdots output(r_{k-1}) & otherwise.\end{cases}$$

Let $rep(r_{j-1})=a_{i_{r_{j-1}}}\cdots a_{m-1}$, $rep(r_j)=a_{i_{r_j}}\cdots a_{m-1}$, where $i_{r_{j-1}}<i_{r_j}<m-1$, $a_{i_{r_k}}=a_i$, $a_{i_{r_0}}=a_1$. Since $r_j=f(r_{j-1})$, by the induction hypothesis,

$$output(r_{j-1})=output(u_{j-1}),$$

$$rep(u_{j-1})=a_{i_{r_{j-1}}}\cdots a_{i_{r_j}-1} \qquad (1\leqq j\leqq k).$$

Thus we get,

$$output(r_0)\,output(r_1)\cdots output(r_{k-1})(.\,a_m) \qquad (if\ t=0)$$

$$=output(u_0)\,output(u_1)\cdots output(u_{k-1})(.\,a_m) \qquad (if\ t=0).$$

But since

$$rep(u_0)=a_{i_{r_0}}\cdots a_{i_{r_1}-1},$$

$$rep(u_1)=a_{i_{r_1}}\cdots a_{i_{r_2}-1},$$

$$\vdots$$

$$rep(u_{k-1})=a_{i_{r_{k-1}}}\cdots a_{i_{r_k}-1},$$

where $a_{i_{r_0}}=a_1$, $a_{i_{r_k}-1}=a_{i-1}$, we have

$$rep(u_0)\,rep(u_1)\cdots rep(u_{k-1})=a_1\cdots a_{i-1},$$

and since $rep(u)=a_1a_2\cdots a_{i-1}$, we have

$$output(u)=output(u_0)\,output(u_1)\cdots output(u_{k-1})(.\,a_m).$$

Therefore

$$output(s)=output(u).$$

Note that by Algorithm 1, if $rep(s)=x$ then

$$f(s)=0 \quad iff \quad output(s)=y,$$

where (x, y) is in K.

THEOREM 3. *Rpmm $M=(K, g, f, output)$ is valid when it runs according as Algorithm* 3.

PROOF. It suffices to show $M(w)=uK(x)$, where $w=ux$. Let $u=u_1u_2\cdots u_m$, $rep(s)=u$ and $x=x_1x_2\cdots x_n$. Consider the transition of M at $ux_1x_2\cdots x_i=rep(r_i)$. Since x is a keyword, there exists t_i such as $f(r_i)=t_i$ and $rep(t_i)=x_1x_2\cdots x_i$. Using Lemma 2, we have $output(r_i)=output(s)=u_1u_2\cdots u_m$, since there is no keyword in u and M is not fail in x_{i+1}. Again since x is a keyword, $output(r_n)=uK(x)$.

5. Time Complexity

By nearly the same discussion as in Aho-Corasick [1] we have the following results on time complexity of our *rpmm*'s,

THEOREM 4. *Algorithm* 3 *makes fewer than* $2n$ *state transitions, including failure transitions, in processing a text of length* n.

THEOREM 5. *Each of Algorithm* 1 *and* 2 *takes time proportional to the sum of the length of keywords in* K.

6. Optimization

The *rpmm* constructed by Algorithm 1 and 2 has, in general, some unnecessary failure transitions. Consider the *rpmm* M in Fig. 2. The failure transitions from states 2 and 3 to states 9 and 10, respectively, are unnecessary. Both of them can be changed to the failure transitions to the state 0, and the values of output function at these states should be changed accordingly. We can systematically eliminate such unnecessary failure transitions just like the case of Aho-Corasick's pattern matching machines. Let $M=(K, g, f, output)$ be an *rpmm* constructed by Algorithm 1 and 2. Then we define new functions f' and $output'$ as follows:

$$f'(1)=0$$

$$f'(i)=\begin{cases} f'(f(i)) & \text{if } g(f(i, a)) \text{ implies } g(i, a) \\ f(i) & \text{otherwise}, \end{cases}$$

and

$$output'(1)=output(1)$$

$$output'(i)=\begin{cases} output(i) & \text{if } f'(i)=f(i) \\ output(i)\, output'(f(i)) & \text{otherwise}, \end{cases}$$

where the state i means a state of depth i counting from the root of the goto tree (graph).

Applying the procedure above to the *rpmm* in Fig. 2 we have an optimized failure function f' with

$$f'(s)=0 \quad \text{for all } s$$

and an output function output' with

$$output'(2)=AB$$

$$output'(3)=A\gamma$$

$$output'(s)=output(s) \quad \text{for } s\neq 2, 3.$$

The next stage of optimizing our *rpmm*'s is to transform them into deterministic *gsm's* (generalized sequential machines), as Aho-Corasick did for their pattern matching machines. As we have seen in Algorithm 3, our *rpmm* positively uses the essence of the failure transitions. (Recall that outputs are emitted only when goto transitions fail.) Thus the deterministic *gsm* necessarily becomes of Mealy type instead of Moore type. So the machine size becomes nearly twice as big as the original one.

7. Concluding Remarks

The authors developed an information system named *SIGMA* and implemented it at University Computer Center [5]. As the data treated in the system is string data, we have developed a fast one-way sequential processor based upon Aho-Corasick's pattern matching machine. The system has *REPLACE* command to simultaneously replace several character string. The command is realized by our *rpmm*, and is quite useful in editing long string data. The users of *SIGMA* may simply list the pairs (x, y).

The other fields of applications include

1) Rōmaji-Kana and Kana-Rōmaji transcriptions,
2) syntactic parsers,
3) indexing systems to extract keywords from documents by using a set of stop words.

In another study [6] we have succeeded to make the Aho-Corasick's pattern matching machine much faster holding the table for the machine in a reasonable size. To apply the technique to our *rpmm* should be another practical problem.

References

[1] KUNUTH, D.E., MORRIS, J.H. JR. and PRATT, V.R.: *Fast Pattern Matching in Strings*, TRCS-74-440, Stanford Univ., (1974).

[2] BOYER, R.S. and MOORE, J.S.: *A Fast String Search Algorithm*, C. ACM **20**, (1977), 762-772.

[3] AHO, A.V. and CORASICK, M.T.: *Efficient String Matching: An Aid to Bibliographic Search*, C. ACM **18**, (1975), 333-340.

[4] ARIKAWA, S.: *One-Way Sequential Search Systems and Their Powers*, Bull. Math. Stat., **19**, (1981), 69-85.

[5] ARIKAWA, S. et al.: *SIGMA—An Information Systems for Researchers Use*, Bull. Inform. Cybernetics, **20**, (1982), 97-114.

[6] ARIKAWA, S. and SHINOHARA, T.: *A Time-Space Trade Off in Realizing Pattern Matching Machines*, Proc. 27th National Conference of Inform. Processing Society, (1983), 11-12 (Japanese).

Received October 15, 1983