Efficient Multiple String Replacing with Pictures

Takeda, Masayuki
Interdisciplinary Graduate School, of Engineering Science, Kyushu University

出版情報：RIFIS Technical Report. 13, 1989-03-14. Research Institute of Fundamental Information Science, Kyushu University
バージョン：
権利関係：

RIFIS-TR-CS-13

RIFIS Technical Report

Efficient Multiple String Replacing with Pictures

Masayuki Takeda

March 14, 1989

Research Institute of Fundamental Information Science
Kyushu University 33
Fukuoka 812, Japan
E-mail: arikawa@rifis.sci.kyushu-u.ac.jp Phone: 092(641)1101 Ext.4470

Efficient Multiple String Replacing with Pictures

Masayuki TAKEDA
Interdisciplinary Graduate School of Engineering Science
Kyushu University 39, Kasuga 816, Japan

March 14, 1989

Abstract

This paper gives a correct proof of the validity of Arikawa-Shiraishi's multiple key replacement algorithm (1984), and then modify the algorithm by using the idea of patterns with pictures.

1. Introduction

When editing texts, we are faced frequently with the need for replacing some words by other words. Arikawa-Shiraishi[2] presents an efficient algorithm for this, which uses a generalized sequential machine. The machine is constructed in nearly the same manner as Aho-Corasick's pattern matching machine[1], and does multiple key replacement in a single pass through the text. Even if the keywords have overlaps, it searches the text from left to right for the longest possible first occurring keyword and replaces it by the paired word. The process is repeated until the end of the text.

Suppose that we wish to change '`child`' to '`children`'. Naive replacement of '`child`' by '`children`' produces a wrong word '`childrenren`' when the text contains '`children`'. Contrary to this, the algorithm possibly produces the desired result by only giving the two replacement pairs ('`child`', '`children`'), ('`children`', '`children`').

The algorithm has been implemented as the `REPLACE` command in the text database management system SIGMA [3], and it is quite useful for editing large texts.

However, suppose that we would change '1900', '1901', ... to '(1900)', '(1901)', ... By this method, we have to give all replacement pairs such as ('1900', '(1900)'), ('1901', '(1901)'), ..., the number of states of the machine to be produced grows accordingly very large.

Such replacement is represented as follows:

$$\texttt{19}NN \;\Rightarrow\; \texttt{(19}NN\texttt{)}.$$

where N is a *picture* for the numeric letters. In this case, we have to search the pattern $\texttt{19}NN$. Then, more generally, we shall consider the matching problem for *patterns with pictures*, problem which will be formally defined in the next section. For example, let A be a picture for a, b, ..., z and N for 0, 1, ..., 9. Then we treat patterns like $\texttt{19}NN$, $\texttt{ab}AN\texttt{1}$, ..., etc.

Takeda[4] describes an algorithm for constructing an efficient pattern matching machine for such patterns. The machine is built easily and quickly, but the number of states is decreased as possibly we can without the state minimization technique. By combining the ideas of the two algorithms, we present an algorithm for replacing keywords with pictures.

This paper is organized as follows: Section 2 gives the definition of the matching problem for patterns with pictures and briefly sketches the algorithm[4] to solve it. Section 3 introduces the Arikawa-Shiraishi multiple key replacement algorithm[2], and Section 4 gives a correct proof of its validity because the 'proof' given in [2] is incomplete. Then a new algorithm is presented in Section 5.

2. Matching problem for patterns with pictures

Let Σ be a finite alphabet and let Σ^* be the free monoid generated by Σ. We call an element w of Σ^* a *string* and the length of w is denoted by $|w|$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ where ε is the empty string. We say that u is a *prefix* and v is a *suffix* of uv, where u, v are strings. $PRE(w)$ denotes the set of all prefixes of a string w, and similarly $SUF(w)$ denotes the set of all suffixes of w. For strings u, w, we say that u *occurs at position* i *in* w iff there exist strings x, y such that $w = xuy$ and $i = |x| + 1$.

Let $\Delta = \{A_1, \ldots, A_p\}$ be a collection of disjoint nonempty subsets of Σ, i.e., $\emptyset \neq A_i \subseteq \Sigma$ and $A_i \cap A_j = \emptyset$ $(i \neq j)$. Each A_i in Δ is called a *picture*. A *pattern* is chosen from $(\Sigma \cup \Delta)^+$, and a *text* from Σ^*. Suppose π is a pattern. Let w be a string. We say that π *occurs at position* i *in* w iff there exists a string u in π such that u occurs at position i in w.

Then, we consider the following problem:

Given a collection of patterns $\Gamma = \{\pi_1, \ldots, \pi_k\}$ and a text T, to find all positions of occurrences of each pattern π_i in T for $i = 1, \ldots, k$.

Note that, when all of the patterns are strings, the above becomes the same as a well-known problem, called the *multiple string searching problem*, and the Aho-Corasick algorithm can efficiently solve it by using a pattern matching machine. We shall then consider to construct a machine that solves the above problem. When the patterns consist only of pictures, we can easily build a machine by treating each picture as a character. But it is difficult when the patterns contain both characters and pictures. The naive solution is to build a machine from all strings belonging to the patterns. That is, for the pattern $\mathtt{a}A$ a machine is built from strings $\mathtt{aa}$, $\mathtt{ab}$, ..., $\mathtt{az}$. This method, however, increases the number of states very large.

The algorithm[4] simply constructs an efficient machine for multiple patterns with pictures. Suppose the set of patterns is $\Gamma = \{A\mathtt{1}, \mathtt{a}A\mathtt{c}, \mathtt{ab}\}$. The algorithm first builds the goto graph as in Fig. 1 (a), by taking each picture as a character (denoted by the outline letter). Then, it makes the goto edges branching off to complete the machine as in Fig. 1 (b). The details will be found in [4].

3. Multiple key replacement

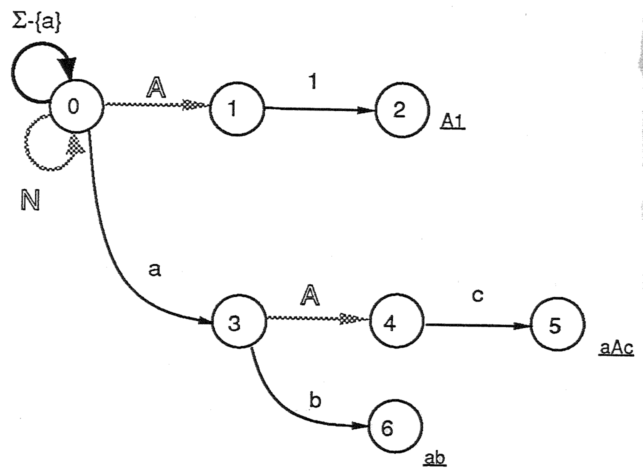
This section briefly describes the Arikawa-Shiraishi algorithm for multiple key replacement [2]. As stated before, the algorithm simultaneously replaces all occurrences of multiple keywords by the paired words in one pass through the text. When the keywords have overlaps, it works according to the following rules:

(1) Replace the leftmost occurrence of the keywords.

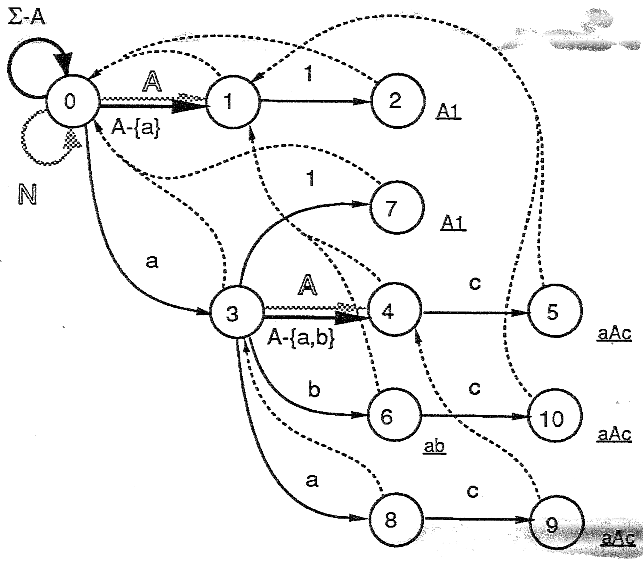
(2) If two or more keywords begin at the same position in the text, then replace the longest one.

The algorithm uses a generalized sequential machine. The machine (*rpmm*, for short) is built in nearly the same manner as the Aho-Corasick pattern matching machine, and consists of the three functions: goto, failure and output, denoted by g, f and out, respectively. The first two are nearly the same as those of the Aho-Corasick machine, but the third is used to emit the replaced text string.

The algorithm consists of the two parts. The first part (Algorithm 1, 2) constructs the rpmm from the replacement pairs. The second part (Algorithm 3) is to run it on the text and replace the keywords by the paired words.



(a)



(b)

The solid arrows represent the goto function, and the broken arrows represent the failure function. The underlined strings below the states mean the outputs from them.

Fig. 1: The machine for $\{A1, \mathrm{a}A\mathrm{c}, \mathrm{ab}\}$

Algorithm 1 Construction of the goto function

input: A collection of pattern pairs $\Pi = \{(x_1, y_1), \ldots, (x_k, y_k)\}$.
output: Goto function g and a partially computed function *out*.
method:

```
{
    $nst := 0$;
    for $i := 1$ to $k$ do $enter(x_i, y_i)$;
    for $\forall c \in \Sigma$ such that $g(0, c) =$ fail do $g(0, c) := 0$
}


procedure $enter(b_1 b_2 \ldots b_m, y)$:
{
    $state := 0$;
    for $j := 1$ to $m$ do {
        if $g(state, b_j) \neq$ fail then {
            $state := g(state, b_j)$
        }else {
            $g(state, b_j) := nst$;
            $state := nst$;
            $nst := nst + 1$
        }
    }
    $out(state) := y$
}
```

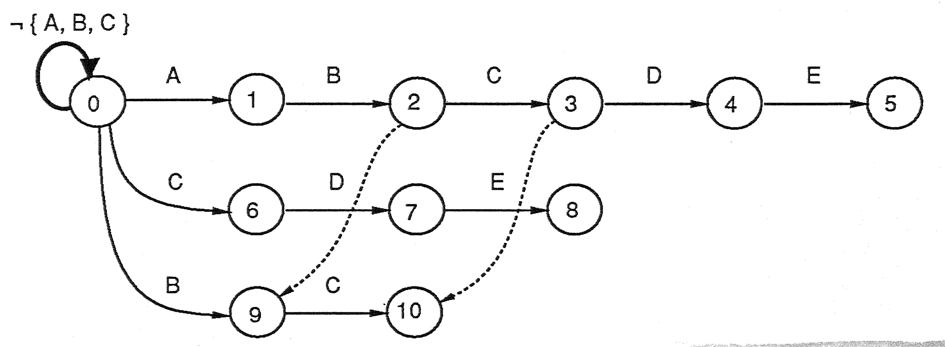
Algorithm 2 Construction of the failure function

input: Goto function g and a partially computed function *out* from Algorithm 1.

output: Failure function f and output function *out*.

method:

```
{
    $queue$ := empty;
    for $\forall c \in \Sigma$ such that $g(0,c) = s \neq 0$ do {
        $queue := queue \cdot s$;
        $f(s) := 0$;
        if $out(s)$ is undefined then $out(s) := c$
    }
    while $queue \neq$ empty do {
        let $queue = r \cdot tail$
        $queue := tail$;
        for $\forall c \in \Sigma$ such that $g(r,c) = s \neq$ fail do {
            $queue := queue \cdot s$;
            if $out(s)$ is defined then {
                $f(s) := 0$
            }else {
                $fst := f(r)$;
                $out(s) := out(r)$;
                while $g(fst,c) =$ fail do {
                    $out(s) := out(s) \cdot out(fst)$;
                    $fst := f(fst)$
                }
                $f(s) := g(fst,c)$;
                if $f(s) = 0$ then $out(s) := out(s) \cdot c$
            }
        }
    }
}
```



(a) The goto and failure functions

The solid arrows represent the goto function, and the broken arrows represent the failure function, where the broken arrows to the state 0 from the states all but 0, 2, 3.

st	*out*(*st*)	
0	undefined	
1	A	
2	A	(*out*(1))
3	A	
4	AαD	(*out*(3) · *out*(10)·D)
5	α	
6	C	
7	CD	(*out*(6)·D)
8	β	
9	B	
10	γ	

(b) The output function

Fig. 2: The rpmm for $\Pi = \{(\texttt{ABCDE}, \alpha), (\texttt{CDE}, \beta), (\texttt{BC}, \gamma)\}$

Algorithm 3 Pattern matching machine for replacing strings

input: A text string $T = a_1 a_2 \ldots a_n$ and a pattern matching machine M with functions g, f and out.

output: A replaced text string T'.

method:

```
{
    $state := 0$;
    for $q := 1$ to $n$ do {
        while $g(state, a_q) =$ fail do {
            print $out(state)$;
            $state := f(state)$
        }
        $state := g(state, a_q)$;
        if $state = 0$ then print $a_q$
    }
    while $state \neq 0$ do {
        print $out(state)$;
        $state := f(state)$
    }
}
```

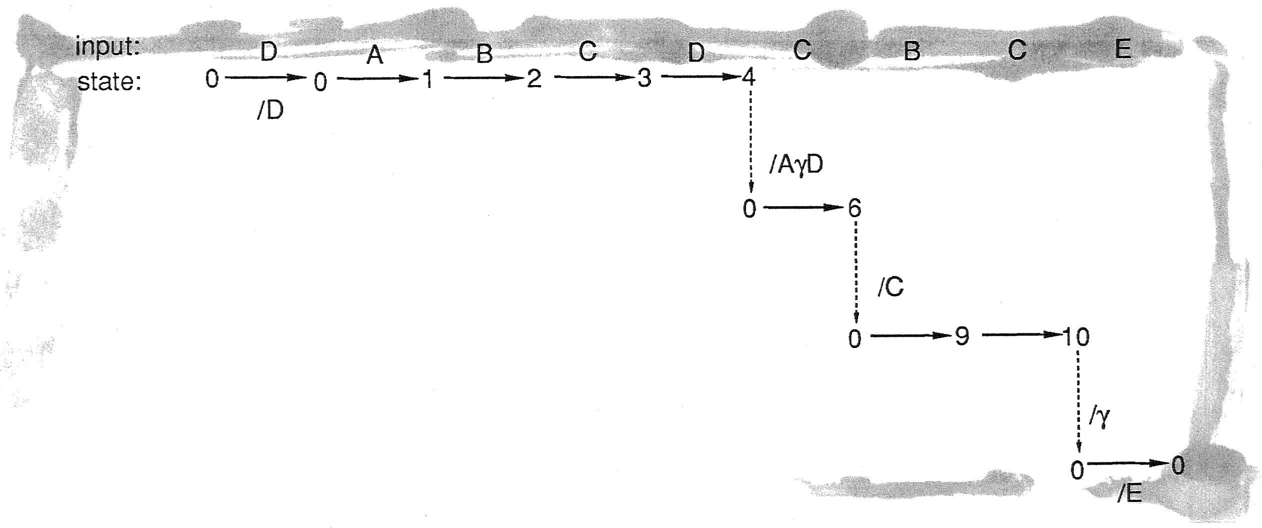


Fig. 3: The move of the rpmm

For example, from the input $\Pi = \{(\texttt{ABCDE}, \alpha), (\texttt{CDE}, \beta), (\texttt{BC}, \gamma)\}$ we construct the rpmm as in Fig. 2. It runs on the text "DABCDCBCE" as in Fig. 3. It should be mentioned that, after reading the fourth character of the text, M is in state 3 and the keyword "BC" is found, but M tries to find another keyword "ABCDE" which contains "BC" as a substring. However, since the sixth character is not "E", M fails from 4 to 0 and emits "$\texttt{A} \cdot \gamma \cdot \texttt{D}$" where γ is the paired word with "BC".

4. The validity of the Arikawa-Shiraishi algorithm

Arikawa-Shiraishi[2] discusses the validity of the algorithm, but the 'proof' is incomplete especially with respect to the characterization of the failure and output functions. These functions must be correctly characterized for our purpose, i.e., to modify the algorithm by using the idea of patterns with pictures. In this section, we shall give a correct proof.

The following lemma, given by Aho and Corasick in [1], holds for the failure function of their pattern matching machine.

Lemma 1 (Aho-Corasick). Suppose that in the goto graph state s is represented by the string u and state t is represented by the string v. Then, $f(s) = t$ iff v is the longest proper suffix of u that is also a prefix of some keywords.

Arikawa-Shiraishi [2] uses the lemma again to characterize the failure function of the rpmm, but obviously it does not hold for this function. (Consider the failure value of the state 4 in Fig. 2 (a)).

Let us see the differences of the constructions of these two failure functions. The failure function f of the Aho-Corasick machine is recursively computed as follows:

Basis. Set $f(s) := 0$ for each state s of depth 1.

Recursive Step. Let s be a state of depth d ($d > 1$). We assume that $f(t)$ are already defined for all states t of depth less than d. Let r be the father of s, and let c be the label on the goto edge from r to s. Let fst be the nearest state from r which is reachable by consecutive failure transitions and which has a goto edge labeled c to any state. Then, set $f(s) := g(fst, c)$.

On the other hand, in the basis of the rpmm, we set $f(s) := 0$ for the states s corresponding to the right ends of the keywords, as well as for the states of depth 1. The recursive step is the same as above except the states whose failure value is already defined in the basis.

How these two functions differ owing to such differences of the bases? To see that, and to give a correct proof of the validity of the rpmm, we shall first introduce some notions.

Let K be a finite subset of Σ^+, elements of which are called *keywords*. Let φ be a mapping from K to Σ^*. Here, we assume that φ is defined by giving the collection of all pairs $\Pi = \{\, (\alpha, \varphi(\alpha)) \mid \alpha \in K \,\}$.

Definition 1. Consider a keyword-occurrence in string u

$$u = x{\cdot}\alpha{\cdot}y \quad (\alpha \in K,\ x, y \in \Sigma^*\,),$$

where the dot "$\cdot$" is used to separate the keyword α from u. We say that this keyword-occurrence is the *LL-occurrence* (*Longest-Leftmost occurrence*) iff:

(1) no keyword begins more left position than the α, and

(2) the α is the longest keyword which begins at the position.

We also call such factorization of u the *LL-factorization of u (on K)*.

Let $M(u)$ denote the string to be emitted from the rpmm M on an input u. Then, the validity of M can be defined as follows:

Definition 2. We say that rpmm M is *valid* for $\langle K, \varphi \rangle$ iff the following holds for any string u:

If u contains some keywords, suppose that the LL-factorization of u is

$$u = x\cdot\alpha\cdot y \quad (\alpha \in K,\ x, y \in \Sigma^*),$$

then,

$$M(u) = x\cdot\varphi(\alpha)\cdot M(y)\,;$$

Otherwise,

$$M(u) = u\,.$$

We shall define sets and a mapping as follows: For the set of the keywords K, put

$$W = \bigcup_{x\in K} PRE(x).$$

Define a mapping **state** recursively by

$$\begin{cases} \mathbf{state}(\varepsilon) = 0 \\ \mathbf{state}(ua) = g(\mathbf{state}(u), a) \quad (u \in W\,,\ a \in \Sigma\,,\ ua \in W). \end{cases}$$

Put $S = \{\mathbf{state}(u) \mid u \in W\}$. Note that W is the set of all prefixes of the keywords and S is the set of states. Note also that the mapping **state** from W to S is obviously a bijection.

We need two more definitions.

Definition 3. We say that the LL-occurrence of u

$$u = x\cdot\alpha\cdot y \quad (\alpha \in K,\ x, y \in \Sigma^*)$$

is *independent* iff for any string $w \in \Sigma^*$,

$$uw = x\cdot\alpha\cdot(yw)$$

is the LL-factorization of uw. Also, we call such LL-factorization the *independent-LL-factorization of u (on K)*.

Definition 4. We say that a factorization of u

$$u = x_1\cdot\alpha_1\cdot x_2\cdot\alpha_2\cdot\ldots\cdot x_m\cdot\alpha_m\cdot y\cdot v \quad (m \geq 0)$$

is the *r-factorization of u (on K)* iff the following conditions (1)~(4) hold.

(1) $x_i \in \Sigma^*$, $\alpha_i \in K$ $(1 \leq i \leq m)$, and $y, v \in \Sigma^*$.

(2) $m = 0$ (i.e., $u = y{\cdot}v$) iff $PRE(u) \cap K = \emptyset$ and u has no independent-LL-factorization.

(3) When $m \geq 1$, we put

$$\begin{aligned} &u_i = x_i{\cdot}\alpha_i{\cdot} \ldots {\cdot}x_m{\cdot}\alpha_m{\cdot}y{\cdot}v \quad (i = 1, 2, \ldots, m), \\ &u_{m+1} = y{\cdot}v\,. \end{aligned}$$

Then,

$$u_1 = x_1{\cdot}\alpha_1{\cdot}u_2$$

is the LL-factorization of u_1, and if $x_1 \neq \varepsilon$ then it is independent. For each i with $2 \leq i \leq m$, if exists,

$$u_i = x_i{\cdot}\alpha_i{\cdot}u_{i+1}$$

is the independent-LL-factorization of u_i. Also, u_{m+1} has no independent-LL-factorization.

(4) The v is the longest string in $(SUF(yv) - \{u\}) \cap W$.

We are now ready to characterize the failure and output functions produced by Algorithm 1, 2.

Lemma 2. Let $st \in S$, $st \neq 0$, and let $st = \mathbf{state}(u)$. Suppose that

$$u = x_1{\cdot}\alpha_1{\cdot}x_2{\cdot}\alpha_2{\cdot} \ldots {\cdot}x_m{\cdot}\alpha_m{\cdot}y{\cdot}v \quad (m \geq 0)$$

is the r-factorization of u on K. Then,

$$f(st) = \mathbf{state}(v)$$

and

$$out(st) = x_1{\cdot}\varphi(\alpha_1){\cdot}x_2{\cdot}\varphi(\alpha_2){\cdot} \ldots {\cdot}x_m{\cdot}\varphi(\alpha_m){\cdot}y\,.$$

Proof. Induction on the length of u. It is clear when $|u| = 1$. Suppose that $|u| > 1$. It is clear for $u \in K$, so we shall assume that $u \notin K$. Let $u = u'c$ ($u' \in \Sigma^+$, $c \in \Sigma$), and put $r = \mathbf{state}(u')$. Then, there exists a finite sequence of states $r_0, r_1, \ldots, r_n$ such that

$$\begin{aligned} &r_0 = r, \\ &r_i = f(r_{i-1}) \ \ (1 \leq i \leq n), \\ &g(r_i, c) = \mathbf{fail} \ \ (1 \leq i \leq n-1), \\ &g(r_n, c) \neq \mathbf{fail}. \end{aligned}$$

Now, put $v^{(0)} = u'$, and suppose that for each i with $1 \le i \le n$, the r-factorization of $v^{(i-1)}$ is

$$v^{(i-1)} = x_1^{(i)} \cdot \alpha_1^{(i)} \cdot \ldots \cdot x_{m_i}^{(i)} \cdot \alpha_{m_i}^{(i)} \cdot y^{(i)} \cdot v^{(i)}.$$

By the induction hypothesis, we obtain for each i with $1 \le i \le n$,

$$r_i = \mathtt{state}(v^{(i)}),$$
$$out(r_{i-1}) = x_1^{(i)} \cdot \beta_1^{(i)} \cdot \ldots \cdot x_{m_i}^{(i)} \cdot \beta_{m_i}^{(i)} \cdot y^{(i)},$$

where $\beta_j^{(i)} = \varphi(\alpha_j^{(i)})$. It is straightforward that the r-factorization of u, if $v^{(n)} = \varepsilon$ and $c \notin W$, is

$$u = x_1^{(1)} \cdot \alpha_1^{(1)} \cdot \ldots \cdot x_{m_1}^{(1)} \cdot (y^{(1)} x_1^{(2)}) \cdot \alpha_1^{(2)} \cdot \ldots \cdot x_{m_n}^{(n)} \cdot \alpha_{m_n}^{(n)} \cdot (y^{(n)} c) \cdot \varepsilon,$$

otherwise, is

$$u = x_1^{(1)} \cdot \alpha_1^{(1)} \cdot \ldots \cdot x_{m_1}^{(1)} \cdot (y^{(1)} x_1^{(2)}) \cdot \alpha_1^{(2)} \cdot \ldots \cdot x_{m_n}^{(n)} \cdot \alpha_{m_n}^{(n)} \cdot y^{(n)} \cdot (v^{(n)} c) .$$

So it is clear. □

From the above lemma the next theorem follows.

Theorem 1. The rpmm M produced by Algorithm 1, 2 is valid for $\langle K, \varphi \rangle$ when it runs according as Algorithm 3.

Proof. Straightforward. □

5. Replacement using patterns with pictures

Consider the replacement

$$\mathtt{19}NN \;\Rightarrow\; \mathtt{(19}NN\mathtt{)}.$$

Using $\langle K, \varphi \rangle$ notation, it can be written as $\langle \mathtt{19}NN, \varphi \rangle$, where $\varphi(x) =$ '(' $\cdot\, x \,\cdot$ ')' for all x in $\mathtt{19}NN$. More generally, let $\Gamma = \{\pi_1, \pi_2, \ldots, \pi_k\}$ be a set of patterns with pictures, i.e., $\pi_i \in (\Sigma \cup \Delta)^+$, $1 \le i \le k$. Suppose that for each i, a mapping φ_i from π_i to Σ^* is defined in a simple way. Put $K = \pi_1 \cup \pi_2 \cup \ldots \cup \pi_k$. Let us define a mapping φ from K to Σ^* by

$$\varphi(x) = \varphi_j(x), \quad \text{where } j \text{ is the largest index with } x \in \pi_j. \qquad (1)$$

Then, our problem is to construct a machine M for $\langle K, \varphi \rangle$.

The Arikawa-Shiraishi method may increase the number of states of M very large. Then we shall modify it by using the idea of [4], briefly sketched in Section 2. Our new algorithm for constructing M is shown in Algorithm 4 and 5.

Suppose $\Gamma = \{\mathtt{19}NN\}$. The machine to be produced is as follows:

Algorithm 4 Construction of the goto function

input: A collection of patterns $\Gamma = \{\pi_1, \ldots, \pi_k\}$.
output: Partially computed functions g and *out*.
method:

```
{
    $nst := 0$;
    for $i := 1$ to $k$ do $enter(\pi_i, i)$;
    for $\forall c \in \Delta \cup \Sigma$ such that $g(0, c) = \textbf{fail}$ do $g(0, c) := 0$
}
```

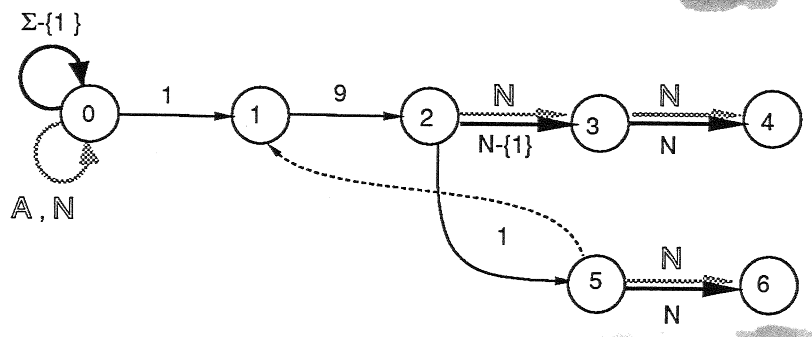
```
procedure $enter(B_1 B_2 \ldots B_m, i)$:
{
    $state := 0$;
    for $j := 1$ to $m$ do {
        if $g(state, B_j) \neq \textbf{fail}$ then {
            $state := g(state, B_j)$
        }else {
            $g(state, B_j) := nst$;
            $state := nst$;
            $nst := nst + 1$
        }
    }
    $out(state) := [\mathrm{K}, i]$
}
```

Algorithm 5 Construction of the failure function

input: Partially computed functions g and *out*.
output: Functions g,f and *out*.
method:

```
    /* The symbol $\circ$ means "concatination" of lists. */
{
    $queue$ := **empty**;
    **for** $\forall P \in \Delta$ such that $g(0, P) = s \neq 0$ **do**
        **for** $\forall c \in P$ such that $g(0, c) = t \neq 0$ **do**
            $copy\_subtree(s, t)$;
    **for** $\forall c \in \Sigma$ such that $g(0, c) = s \neq 0$ **do** {
        $queue := queue \cdot s$;
        $f(s) := 0$;
        **if** $out(s)$ is **undefined then** $out(s) := [\mathbf{B}, 1]$
    }
    **for** $\forall P \in \Delta$ such that $g(0, P) = s \neq 0$ **do** {
        $queue := queue \cdot s$;
        $f(s) := 0$;
        **if** $out(s)$ is **undefined then** $out(s) := [\mathbf{B}, 1]$;
        **for** $\forall c \in P$ such that $g(0, c) = 0$ **do** $g(0, c) := s$
    }
    **while** $queue \neq$ **empty do** {
        **let** $queue = r \cdot tail$
        $queue := tail$;
        **for** $\forall P \in \Delta$ such that $g(r, P) = s \neq$ **fail do**
            **for** $\forall c \in P$ such that $g(r, c) = t \neq$ **fail do**
                $copy\_subtree(s, t)$;
        **for** $\forall c \in \Sigma$ such that $g(r, c) = s \neq$ **fail do** {
            $queue := queue \cdot s$;
            **if** $out(s)$ is **defined then** {
                $f(s) := 0$
            }**else** {
                $fst := f(r)$;
                $out(s) := out(r)$;
                **while** $g(fst, c) =$ **fail do** {
                    $out(s) := out(s) \circ out(fst)$;
                    $fst := f(fst)$
                }
                $f(s) := g(fst, c)$;
                **if** $f(s) = 0$ **then** $out(s) := out(s) \circ [\mathbf{B}, 1]$;
            }
        }
        for $\forall P \in \Delta$ such that $g(r,P) = s \neq$ fail do {
            $queue := queue \cdot s$ ;
            if $out(s)$ is defined then {
                $f(s) := 0$;
                for $\forall c \in P$ such that $g(r,c) =$ fail do $g(r,c) := s$
            }else {
                $fst := f(r)$;
                $out(s) := out(r)$;
                while $g(fst,P) =$ fail do {
                    $out(s) := out(s) \circ out(fst)$;
                    $fst := f(fst)$
                }
                $f(s) := g(fst,P)$;
                if $f(s) = 0$ then $out(s) := out(s) \circ [\mathrm{B},1]$;
                for $\forall c \in P$ such that $g(r,c) =$ fail do {
                    $fst := f(r)$;
                    $temp := out(r)$;
                    while $g(fst,c) =$ fail do {
                        $temp := temp \circ out(fst)$;
                        $fst := f(fst)$
                    }
                    $st := g(fst,c)$;
                    if $st = 0$ then $temp := temp \circ [\mathrm{B},1]$;
                    if $st \neq f(s)$ then {
                        $new := nst$;
                        $nst := nst + 1$;
                        $copy\_subtree(s, new)$;
                        $g(r,c) := new$;
                        $queue := queue \cdot new$;
                        $f(new) := st$;
                        $out(new) := temp$
                    }else {
                        $g(r,c) := s$
                    }
                }
            }
        }
    }
}
```

```
procedure $copy\_subtree(st1, st2)$ :
{
    $queue2 := \{(st1, st2)\}$;
    while $queue2 \neq$ empty do {
        let $queue2 = (r1, r2) \cdot tail$
        $queue2 := tail$;
        let $out(r1) = [\mathrm{K}, n_1]$ and $out(r2) = [\mathrm{K}, n_2]$
        if $n_2 > n_1$ then $out(r2) := out(r1)$;
        for $\forall X \in \Delta \cup \Sigma$ such that $g(r1, X) = s \neq$ fail do {
            if $g(r2, X) =$ fail then {
                $st := nst$;
                $nst := nst + 1$;
                $g(r2, X) := st$
            }else {
                $st := g(r2, X)$;
                $queue2 := queue2 \cdot (s, st)$
            }
        }
}
```



The goto and failure functions are the same as those of the Arikawa-Shiraishi. However, the output function should be different because each edge may be labeled by two or more characters. Suppose that M is in state 3 and the next character is not numeric. Then, M fails to 0 by the failure function f, but the string to be emitted cannot be determined without knowing the character causing the transition from 3 to 4. Hence, we shall use a cyclic buffer, from which we read the characters required

Algorithm 6 Pattern matching machine for replacing strings

input: A text string $T = a[1:n]$ and a pattern matching machine M with functions g, f and *out*.

output: A replaced text string T'.

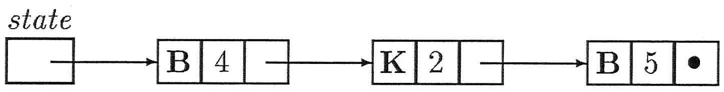
method:

```
{
    $state := 0$;
    for $q := 1$ to $n$ do {
        while $g(state, a[q]) =$ fail do {
            $print\_output(q-1, state)$;
            $state := f(state)$
        }
        $state := g(state, a[q])$;
        if $state = 0$ then print $a[q]$
    }
    while $state \neq 0$ do {
        $print\_output(n, state)$;
        $state := f(state)$
    }
}
```

```
procedure $print\_output(p, state)$:
{
    let $out(state) = [\mathbf{B}, i_1] \circ [\mathbf{K}, n_1] \circ \ldots \circ [\mathbf{B}, i_m] \circ [\mathbf{K}, n_m] \circ [\mathbf{B}, i_{m+1}]$
    $l := p - depth(state)$;
    for $j := 1$ to $m$ do {
        print $a[l+1 : l+i_j]$;
        $l := l + i_j$;
        $len := length(\pi_{n_j})$;
        $x := a[l+1 : l+len]$;
        print $\varphi_{n_j}(x)$;
        $l := l + len$
    }
    print $a[l+1 : l+i_{m+1}]$;
}
```

to emit the output strings.

The output function may have a linked-list structure such as



Here, **B** means 'buffer' and **K** means 'keyword'. The entire list represents as follows: Emit 4 characters from the buffer, replace the occurrence of π_2 by the corresponding string and emit 5 characters again. The algorithm to make the obtained machine run on a text is summarized in Algorithm 6.

We shall mention the properties of our machine. Define sets W, S and a mapping **state** as in Section 4. This time the mapping **state** from W into S is not a bijection, then let

$$rep(s) = \{\, u \in W \mid \mathbf{state}(u) = s \,\}.$$

Then, the following lemma holds for the failure function f and the output function *out* produced by Algorithm 4, 5.

Lemma 3. Let $s, t \in S$, $s \neq 0$, $f(s) = t$. Then, for any u in $rep(s)$, the following holds:

> Suppose that the r-factorization of u on K is
>
> $$u = x_1{\cdot}\alpha_1{\cdot}x_2{\cdot}\alpha_2{\cdot}\ldots{\cdot}x_m{\cdot}\alpha_m{\cdot}y{\cdot}v \quad (m \geq 0).$$
>
> Then,
>
> $$v \in rep(t)$$
>
> and $out(s)$ equals to
>
> $$[\mathbf{B}, |x_1|\,] \circ [\mathbf{K}, num(\alpha_1)\,] \circ \ldots \circ [\mathbf{B}, |x_m|\,] \circ [\mathbf{K}, num(\alpha_m)\,] \circ [\mathbf{B}, |y|\,],$$
>
> where $num(\alpha) = \max\{\, i \mid 1 \leq i \leq k,\ \alpha \in \pi_i\}$

Proof. Similar to the proof of Lemma 2. □

6. Conclusions

We have proved the validity of the multiple key replacement algorithm by Arikawa-shiraishi[2], and have modified the algorithm by using the idea of patterns with pictures. The obtained algorithm replaces the longest first found pattern by the corresponding word. When the patterns have intersections, like aA and Aa, replacement will be done according to their priorities given by the user.

Our algorithm requires as input the list of pairs $(\pi_1, \varphi_1), (\pi_2, \varphi_2), \ldots, (\pi_k, \varphi_k)$, where each π_i is a pattern and φ_i is a mapping from π_i into Σ^*. It is most suited when each π_i contains pictures and $\varphi_i(x)$ can be simply expressed with x, such as

$$\begin{aligned} \varphi_i(x) &= \text{`('} \cdot x \cdot \text{`)'}\ , \\ \varphi_j(x) &= x \cdot \text{`}\texttt{\textbackslash index\{}\text{'} \cdot x \cdot \text{`\}'}\ , \end{aligned}$$

and so on. If there is no apparent way to describe φ_i other than to store for each $x \in \pi_i$ the value $\varphi_i(x)$, then, the Arikawa-Shiraishi method would be applied for the input $\Pi = \{(x, \varphi(x)) | x \in K\}$, where $K = \pi_1 \cup \ldots \cup \pi_k$ and φ is defined by Equation (1) in Section 5. However, our method is possibly substituted for it with little loss of the running speed.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, June 1975.

[2] S. Arikawa and S. Shiraishi. Pattern matching machines for replacing several character strings. *Bull. Inform. Cybern.*, 21(1–2):101–111, 1984.

[3] S. Arikawa et al. SIGMA: A text database management system. Technical Report RIFIS-TR-CS-4, Res. Inst. of Fundamental Information Science, KYUSHU University, June 1988.

[4] M. Takeda. A fast matching algorithm for patterns with pictures. Technical Report RIFIS-TR-CS-11, Res. Inst. of Fundamental Information Science, KYUSHU University, March 1989.