A Proof of the Correctness of Uratani's String Searching AIgorithm

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

出版情報：RIFIS Technical Report. 7, 1988-10-28. Research Institute of Fundamental Information Science, Kyushu University

RIFIS-TR-CS-7

RIFIS Technical Report

A Proof of the Correctness of Uratani's String Searching Algorithm

Masayuki Takeda

October 28, 1988

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

A Proof of the Correctness of Uratani's String Searching Algorithm

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

Abstract

The string searching problem is to find all occurrences of the pattern(s) in a text string. The Aho-Corasick string searching algorithm finds simultaneously all occurrences of the multiple patterns during one pass through the text. On the other hand, the Boyer-Moore algorithm is understood to be the fastest algorithm for a single pattern. By combining the ideas of these two algorithms, Uratani presented an efficient string searching algorithm for multiple patterns. The algorithm runs in *sublinear* time on the average as the BM algorithm achieves, and its preprocessing time is linear proportional to the sum of the lengths of the patterns like the AC algorithm. However, the correctness of the algorithm has not been discussed. In this paper, we prove the correctness of the algorithm.

Keywords and Phrases: bibliographic search, finite state machines, information retrieval, pattern matching, string searching, text-editing, time complexity.

1 Introduction

The string searching problem is defined as follows:

> Given strings $p_1p_2...p_m$, called a *pattern*, and $a_1a_2...a_n$, called a *text*, to find all indices j with $a_ja_{j+1}...a_{j+m-1} = p_1p_2...p_m$.

Knuth, Morris and Pratt [5], [6] presented an algorithm to solve this problem, which runs in time $O(n)$. Aho and Corasick [1] have shown how it can be extended to the multiple pattern problem. In their method, a finite state machine is built which simultaneously recognizes all occurrences of the patterns during one pass through the text. Such a finite state machine is called a *pattern matching machine*, *pmm* for short. The running time is also $O(n)$.

Both of the KMP and AC algorithms scan the patterns from left to right. Boyer and Moore [2] noticed that more information about the text can be gained if scanning is done from right to left. It is understood that the BM algorithm, based on this idea, is the fastest algorithm for a single pattern. The running time on the average is *sublinear*.

To characterize the efficiency of a string searching algorithm, we use the *probe rate*, defined as the number of times a text character is checked per text character. By *sublinear*, we mean that the probe rate is less than one. For the AC algorithm the probe rate is exactly one, being the same in the average and worst cases.

Uratani [8] extended the BM algorithm to multiple patterns using the idea of pmm. The algorithm runs in sublinear time on the average as the BM algorithm achieves. For a single pattern, it runs faster than the BM algorithm. Moreover the construction of the pmm requires only linear time proportional to the sum of the lengths of the patterns like the AC algorithm. The Uratani algorithm is most suited when all patterns are sufficiently long and the alphabet size is large, but the number of patterns is not very large.

It should be mentioned that Commentz-Walter [3] reported an extension of the BM algorithm to multiple patterns. But it has received little attention, probably because her method is a straightforward modification of the BM and the construction of the algorithm is incomplete. Kowalski and Meltzer [7] sketched another extension of the BM algorithm to multiple patterns. But it is for patterns with the same length. The Uratani algorithm is more efficient and runs faster than these two algorithms.

However, [8] did not discussed the correctness of the algorithm. In this paper, we give a poof of the correctness of the algorithm. The organization of this paper is as follows: In the next two sections, we describe the Uratani algorithm using an example and show how to construct the pmm from a collection of patterns. We discuss the time complexity of the algorithm, both in the text searching and in the construction of the pmm. Then we give a proof of the correctness of the algorithm. Furthermore, we compare the Uratani method with the BM method for a single pattern.

2 The Uratani algorithm

The basic idea of the BM algorithm is to match the pattern from right to left. Uratani applied this idea to the multiple pattern problem as in the following example.

Example 1 Suppose that the collection of desired patterns is { `trace,artist,smart` } and the text is "`the-greatest-artist....`" . First, we align the right ends of these patterns. Since the length of the shortest pattern is 5, we start matching at the 5th character of the text. In the figures bellow, "`*`" denotes a *success* in the matching and "`x`" denotes a *mismatch.*

```
 trace
artist
 smart
 the-greatest-artist....
     x
```

Since "`g`" is known not to occur in any pattern, we can shift the patterns to right by 5, the length of the shortest pattern.

```
     trace
    artist
     smart
 the-greatest-artist....
         x*
```

Now, the current character "`e`" in the text matches the "`e`" in the pattern "`trace`", so we move the pointer on the text to left by one. Then we have a mismatch. Since the checked substring "`te`" of the text does not occur in any pattern, we can again shift the patterns to right by 5 (move the pointer to right by 6) and we have:

```
      trace
     artist
      smart
 the-greatest-artist....
               x
```

The current character "`r`" makes a mismatch. So as to align this "`r`" with the "`r`" in "`smart`", we shift the patterns to right by one.

```
      trace
     artist
      smart
 the-greatest-artist....
              x***
```

From right to left, we succeed in matching, for "`t`","`r`","`a`", successively, and fail at the next character "`-`". Consider occurrences in the patterns of the checked string "`-art`". So as to align its suffix "`art`" with the "`art`" in "`artist`", we shift the patterns to right by 3 positions (and move the pointer to right by 6).

```
                 trace
                artist
                 smart
the-greatest-artist....
                ******
```

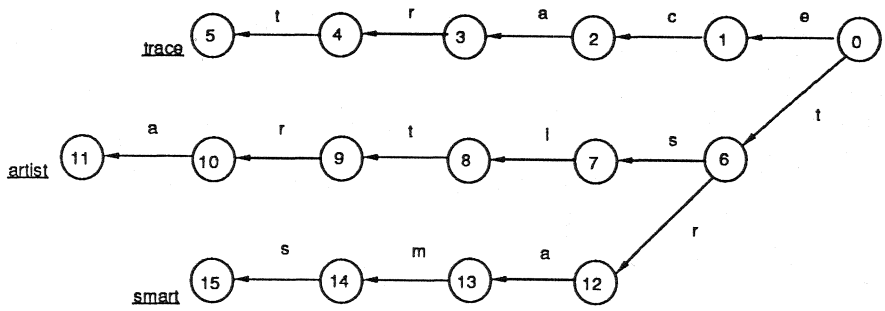
This time we discover that each character of the pattern "`artist`" matches the corresponding character in the text. To the text of 19 characters, we made only 14 references, and six of them were required to confirm the final match.

As above, starting the match at the right end of the pattern is also effective for multiple patterns. To realize this matching technique, Uratani uses a pmm similar to the AC algorithm. The pmm consists of three functions: *goto*, *output* and *failure*. **Fig. 1** shows the pmm for the patterns in **Example 1**. In contrast with the pmm of the AC algorithm, it is built for the *reversed* patterns.

By using such a pmm, we do matching as follows: Start at state 0, scanning the text backward from the *minlen*th character (*minlen* is the length of the shortest pattern), and try to make state transitions by the goto function as many as possible. If we reach the state corresponding to the left end of some pattern, it means that the pattern has been found at the position. If we fail in a state transition, we move the pointer on the text forward by the amount indicated by the failure function and start at state 0 again. In the case of **Example 1**, the pmm moves as in **Fig. 2**. The text searching algorithm is summarized in **Algorithm 1**.

3 Construction of the pmm

In this section we describe how to construct the pmm from a given collection of patterns. First, we construct the goto function g and the output function *out*. In a way similar to the AC algorithm, we construct a goto *graph* from the *reversed* patterns as in **Fig. 3**. The algorithm for this are summarized in **Algorithm 2**. Notice that there is a loop from state 0 to state 0 in **Fig. 3**. It is added for the sake of convenience to construct the failure function. After the completion of the pmm, we ignore it. In **Algorithm 2**, the functions *depth* and *emap*, required to construct the failure function, are also computed. Note that $depth(st)$ denotes the length of the shortest path from state 0 to state st and $emap(j)$ denotes the state corresponding to the left end of the jth pattern.



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

(a) **The goto and output functions**

character\state	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	2	6	*	8	9	10	5	6	7	8	*	10	*	6	7	8
c	1	*	7	8	9	10	5	6	7	8	9	10	6	6	7	8
e	*	6	7	8	9	10	5	6	7	8	9	10	6	6	7	8
i	2	6	7	8	9	10	5	*	7	8	9	10	6	6	7	8
m	3	6	7	8	9	10	5	6	7	8	9	10	6	*	7	8
r	1	6	7	*	9	10	*	6	7	*	9	10	6	6	7	8
s	1	6	7	8	9	10	*	6	7	8	9	10	6	6	*	8
t	*	6	7	8	*	10	5	6	*	8	9	10	6	6	7	8
else	5	6	7	8	9	10	5	6	7	8	9	10	6	6	7	8

(b) **The failure function**

Fig. 1. The pmm for { trace,artist,smart }

```
t h e - g r e a t e s t - a r t i s t
            0
              1← 0
                                        0
                                13←12← 6← 0
                                11←10← 9← 8← 7 ← 6 ←0
```

Fig. 2. The moves of the pmm

input: A text string $X = a_1 a_2 \ldots a_n$ and a pattern matching machine with functions g, f and *out*.
output: Locations at which patterns occur in X.
method:
/* \$ is the symbol which is known not to occur in any pattern. */
/* *minlen* is the length of the shortest pattern. */

```
begin
    a_0 :=$;
    state := 0;
    q := minlen;
    while q ≤ n do
        begin
            while g(state, a_q) > 0 do
                begin
                    state := g(state, a_q);
                    if out(state) ≠empty then print q, out(state) ;
                    q := q − 1
                end;
            q := q + f(state, a_q);
            state := 0
        end
end
```

Algorithm 1

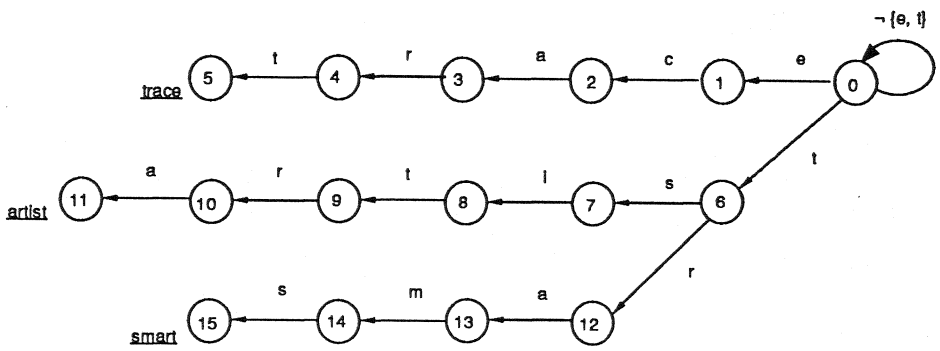


Fig. 3. An example of the goto graph constructed by Algorithm 2

Now, using the goto graph just constructed, we compute the failure function f. Let $a_1a_2\ldots a_n$ be a text string. Suppose that we have just scanned the matched characters $a_{i+1},\ldots,a_j$ backward and reached state st and a mismatch occurs at a_i. To compute the amount by which we can move the pointer on the text, the following two cases are considered in [8] :

(1) The scanned string $a_ia_{i+1}\ldots a_j$ equals to a substring of some pattern.

(2) A suffix of the matched string $a_{i+1}\ldots a_j$ equals to a prefix of some pattern.

For each of the conditions (1), (2), let us denote the possible move of the pointer by $shift1$, $shift2$, respectively. $shift1$ is a function of st and a_i, and $shift2$ is a function of only st. Then, the failure function f is defined by

$$f(st,a) = \min\{shift1(st,a), shift2(st)\},$$

for each pair (st,a) with $g(st,a) =$ **fail** (or 0). Here, if $shift1(st,a)$ is undefined, $f(st,a)$ should be $shift(st)$. Note that $shift2(st)$ is defined for all states st.

The construction of the functions $shift1$ and $shift2$ requires another function ϕ. The function ϕ is similar to the failure function of the AC algorithm. **Fig. 4** shows an example of the function ϕ.

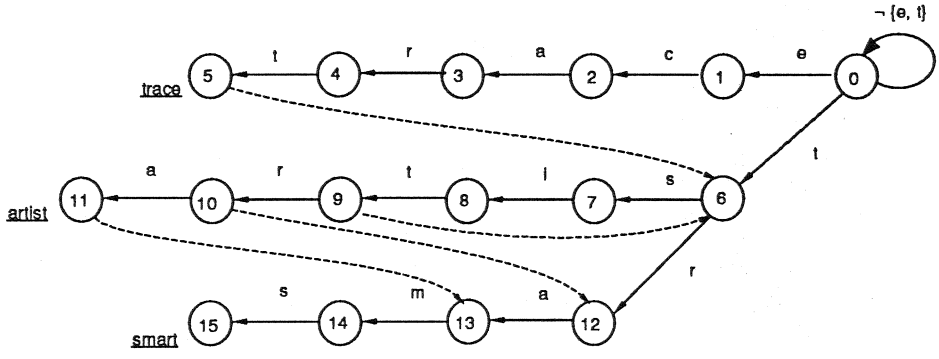
The algorithms to construct the functions ϕ, $shift1$ and $shift2$ are summarized in **Algorithm 3.1**, **3.2** and **3.3**, respectively. In the next section, we prove the correctness

input: A collection of patterns K=$\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$.
output: Functions g, *depth*, *emap* and *out*.
method:
/* **large** is a positive integer which is much larger than lengths of patterns. */
/* We initialize g to **fail** (< 0) and *out* to **empty**. */

```
begin
    minlen :=large
    nst := 1;        depth(0) := 0;
    for i = 1 to k do enter(α_i, i);
    for each c such that g(0,c) =fail do g(0,c) := 0
end

procedure enter(a_1 a_2 ... a_m, i) :
begin
    state := 0;      j := m;
    while (g(state, a_j) ≠ fail) and (j > 0) do
        begin
            state := g(state, a_j);
            j := j − 1
        end;
    while j > 0 do
        begin
            g(state, a_j) := nst;
            depth(nst) := m − j + 1;
            state := nst;
            nst := nst + 1;
            j := j − 1
        end;
    emap(i) := state;
    out(state) := {a_1 a_2 ... a_m} ;
    if minlen > m then minlen := m
end
```

Algorithm 2 Construction of the goto and output functions



The broken arrows represent the function ϕ, where broken arrows to the state 0 from all but the states 0, 5, 9, 10 and 11 are omitted.

Fig. 4. An example of the function ϕ

of these algorithms.

4 The correctness

In this section, we characterize each of the functions, and then we prove the correctness of the Uratani algorithm. Initially, we define the sets, mapping and relation required for our proof.

Definition 1 Let Σ be an alphabet set, and for each u in Σ^*, put

$$PRE(u) = \{x \in \Sigma^* \mid \exists y \in \Sigma^* : xy = u\},$$
$$SUF(u) = \{y \in \Sigma^* \mid \exists x \in \Sigma^* : xy = u\}.$$

Let $K = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ be a collection of patterns, and put

$$W = \bigcup_{\alpha \in K} SUF(\alpha).$$

```
input:   Goto function g.
output:  Function φ.
method:

begin
    queue :=empty;
    for each c such that g(0,c) = s > 0 do
        begin
            queue := queue · s;
            φ(s) := 0
        end;
    while queue ≠empty do
        begin
            let queue = r · tail
            queue := tail;
            for each c such that g(r,c) = s ≠fail do
                begin
                    queue := queue · s;
                    fst := φ(r);
                    while g(fst,c) =fail do
                        fst := φ(fst);
                    φ(s) := g(fst,c)
                end
        end
end
```

Algorithm 3.1 Construciton of the function ϕ

```
input:   Functions g, φ.
output:  Function shift1.
method:
/* We initialize shift1 to large. */

begin
    queue :=empty;
    for each c such that g(0,c) = s > 0 do
        queue := queue · s;
    while queue ≠ empty do
        begin
            let queue = r · tail
            queue := tail;
            for each c such that g(r,c) = s ≠ fail do
                begin
                    queue := queue · s;
                    fst := φ(r);
                    while g(fst,c) = fail do
                        begin
                            if shift1(fst,c) > depth(r) then
                                shift1(fst,c) := depth(r);
                            fst := φ(fst)
                        end;
                    if (g(fst,c) = 0) and (shift1(fst,c) > depth(r)) then
                            shift1(fst,c) := depth(r)
                end
        end
end
```

Algorithm 3.2 Construciton of the function $shift1$

```
input:   Functions g, depth and emap.
output:  Function shift2.
method:
/* We initialize shift2 to large. */

begin
    for j = 1 to k do
        begin
            jst := emap(j);
            fst := φ(jst);
            while fst > 0 do
                begin
                    if shift2(fst) > depth(jst) then
                        shift2(fst) := depth(jst);
                    fst := φ(fst)
                end
        end;
    shift2(0) := minlen;
    queue := {0};
    while queue ≠empty do
        begin
            let queue = r · tail
            queue := tail;
            for each c such that g(r,c) = s > 0 do
                begin
                    queue := queue · s;
                    if shift2(s) > shift2(r) + 1 then
                        shift2(s) := shift2(r) + 1
                end
        end
end
```

Algorithm 3.3 **Construciton of the function** $shift2$

On the other hand, let S be the set of states appearing in the goto graph constructed from K. Let us define a mapping $state : W \to S$ by

$$\begin{cases} state(\varepsilon) = 0, \\ state(aw) = g(state(w), a) \quad (aw \in W\,, a \in \Sigma\,, w \in \Sigma^*), \end{cases}$$

where ε denotes the *empty word.* Note that the mapping *state* is a *bijection.* Moreover, we define a relation $\vdash_\phi$ on S by

$$s \vdash_\phi t \quad \Leftrightarrow \quad \phi(s) = t,$$

where $s, t \in S$.

Concerning the failure function f, the following theorem holds.

Theorem 1 Suppose $u \in W$, $a \in \Sigma$ and $au \notin W$. Let v be the shortest string in Σ^+ which satisfies at least one of the following conditions:

(1) $auv \in W$,
(2) $\exists y \in SUF(u) : yv \in K$.

Put $st = state(u)$. Then

$$f(st, a) = |u| + |v|.$$

To prove this theorem, we should characterize the functions ϕ, $shift1$ and $shift2$. The following lemma, which is essentially the same as Aho and Corasick [1], is to characterize the function ϕ constructed by **Algorithm 3.1**.

Lemma 1 Suppose $u \in W - \{\varepsilon\}$, and put $st = state(u)$. Let v be the longest string in the set

$$[PRE(u) - \{u\}] \cap W,$$

then

$$\phi(st) = state(v).$$

Proof By induction on the length of u. □

We immediately get the following corollary.

Corollary If $u, v \in W$ and $u \neq \varepsilon$, then

$$v \in PRE(u) - \{u\} \;\Leftrightarrow\; state(u) \vdash_{\phi}^{+} state(v)\,.$$

Concerning the function $shift1$ constructed by **Algorithm 3.2**, the following lemma holds (see **Fig. 5**).

Lemma 2 For each pair (u, a) in $W \times \Sigma$ with $au \notin W$, put

$$H(u, a) = \{v \in \Sigma^{+} \mid auv \in W\}$$

and $st = state(u)$. Then

$$shift1(st, a) = \begin{cases} |u| + \min\{|v| \mid v \in H(u, a)\}, & \text{if } H(u, a) \neq \emptyset; \\ \textbf{large (or undefined)}, & \text{otherwise.} \end{cases}$$

Proof For each pair (st, a) with $g(st, a) = \textbf{fail}$ (or 0), let

$$A(st, a) = \left\{ r \in S \;\middle|\; \begin{array}{l} r \neq 0, \quad g(r, a) \neq \textbf{fail} \\ \exists r_0, \ldots, \exists r_n \in S : \\ r_0 = r, \quad r_n = st, \\ r_{i+1} = \phi(r_i)\ (0 \leq \forall i < n), \\ g(r_i, a) = \textbf{fail}\ (1 \leq \forall i < n) \end{array} \right\}$$

From **Algorithm 3.2**, it is clear that

$$shift1(st, a) = \begin{cases} \min\{depth(r) \mid r \in A(st, a)\}, & \text{if } A(st, a) \neq \emptyset; \\ \textbf{large}, & \text{otherwise.} \end{cases}$$

Hence, it is sufficient to show the following (1) and (2):

(1) $H(u, a) = \emptyset \;\Leftrightarrow\; A(st, a) = \emptyset$,

(2) $\min\{|uv| \mid v \in H(u, a)\} = \min\{depth(r) \mid r \in A(st, a)\}$, if $H(u, a) \neq \emptyset$.

Since

$$A(st, a) \subseteq \{state(uv) \mid v \in H(u, a)\},$$

clearly

$$H(u,a) = \emptyset \;\Rightarrow\; A(st,a) = \emptyset.$$

Suppose $H(u,a) \neq \emptyset$. Let v be the shortest string in $H(u,a)$, and put $r = state(uv)$. Then we have

$$\exists r_0, \ldots, \exists r_n \in S \;:\; [r_0 = r\,,\; r_n = st,\; r_{i+1} = \phi(r_i)\;(0 \leq \forall i < n)]\,.$$

We wish to show $r \in A(st,a)$. To do this, we must show that

$$1 \leq \forall i < n\,,\; g(r_i, a) = \textbf{fail}.$$

Suppose

$$1 \leq \exists j < n \;:\; g(r_j, a) \neq \textbf{fail}.$$

Then, for the $v' \in W - \{\varepsilon\}$ with $state(uv') = r_j$, we get

$$v' \in H(u,a)\,,\; |v'| < |v|,$$

which is a contradiction. So we have

$$\begin{aligned} &A(st,a) \neq \emptyset, \\ &\min\{|uv| \mid v \in H(u,a)\} \geq \min\{depth(r) \mid r \in A(st,a)\}\,. \end{aligned}$$

The converse inequality is clear. □

The following lemma characterizes the function $shift2$ constructed by **Algorithm 3.3**.

Lemma 3 For each u in W, put

$$V(u) = \{v \in W - \{\varepsilon\} \mid \exists y \in SUF(u) \;:\; yv \in K\},$$

and $st = state(u)$. Then,

$$shift2(st) = |u| + \min\{\,|v| \mid v \in V(u)\,\}.$$

Proof Put

$$B(st) = \{r \in state(K) \mid r \vdash_{\phi}^{+} st\,\},$$

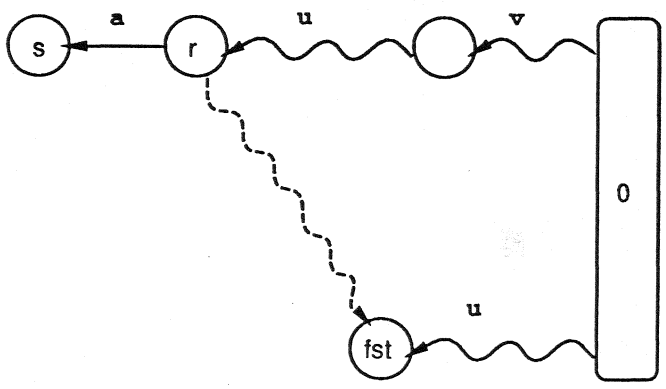


Fig. 5. Construction of the Function $shift1$

then, from **Algorithm 3.3**, we get

$$shift2(st) = \begin{cases} minlen\,, & \text{if } st = 0; \\ \min[\{depth(r) \mid r \in B(st)\} \cup \{shift2(fath(st)) + 1\}\,], & \text{if } st > 0\,, \end{cases}$$

where $fath(st)$ denotes the *father* of st. It is clear for $u = \varepsilon$. When $u \neq \varepsilon$, let $u = cu'$ with $c \in \Sigma, u' \in \Sigma^*$. Then

$$V(u) = \{v \in W - \{\varepsilon\} \mid uv \in K\} \cup V(u').$$

So it is clear. □

Theorem 1 follows from **Lemmas 2** and **3**. Now, let us prove the correctness of **Algorithm 1**, the text searching algorithm. It is sufficient to show the following lemma.

Lemma 4 Let $a_1 a_2 \ldots a_n$ be a text string, and set $a_0 :=$ \$ (the symbol which is known not to occur in any pattern). Let

$$u = a_{p+1} \ldots a_q \in W \quad (0 \leq p \leq q \leq n)\,,$$
$$st = state(u)$$

(if $p = q$ then let $u = \varepsilon$). If $g(st, a_p) =$**fail** (or 0), then

$$[a_i \ldots a_j \in K\,,\ q < j] \Rightarrow p + f(st, a_p) \leq j\,.$$

Proof By **Theorem 1**. □

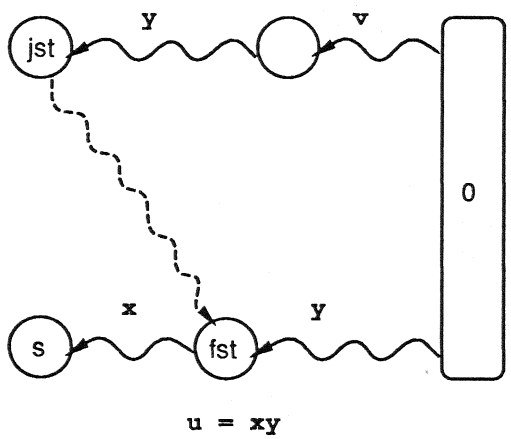


Fig. 6. Construction of the function $shift2$

5 Time complexity

For simplicity, we assume that there are k patterns which are all of the same length m. Suppose that the alphabet size q is much larger than k, m. Uratani [8] showed that the efficiency on the average of our text searching algorithm is nearly given by

$$probe\ rate = \frac{1}{m} + \frac{(m+1)k}{2mq}\ (<1).$$

In the worst case, the *probe rate* is *maxlen*, the length of the longest pattern.

Let us consider the time required to construct the pmm from the patterns. We can construct the pmm, namely the goto, output and failure functions, in time linear proportional to the sum of the lengths of the patterns like the AC method.

6 Comparison with the BM method

We shall compare the Uratani method with the BM method for a single pattern.

Example 2 Suppose that for the pattern "ABBCABCA" we have :

```
pattern: ABBCABCA
   text: ....CBCA.......
results:     x***
```

By the BM method, using *delta*2, we try to find the *right-most plausible reoccurrence* of the matched string "BCA" (see [2]). Since we can shift the pattern by 3, we get

```
pattern:    ABBCABCA
   text: ....CBCA.......
```

and we go on matching. However, it is obvious that a mismatch will occur.

Against this, by the Uratani method, we try to find the right-most occurrence of the scanned string "CBCA" . So we can shift the pattern by 7 and get

```
pattern:        ABBCABCA
   text: ....CBCA.......
```

As above, the Uratani method can often shift the pattern farther than the BM method. A more strict argument follows: Let *pat* be a pattern and *patlen* be its length. Put

$$delta2'(st) = delta2(patlen - st)\,.$$

We would replace $f(state, a_q)$ of **Algorithm 1** by

$$\max\{delta1(a_q)\,,\ delta2'(state)\}\,,$$

we could obtain the BM algorithm. The functions *delta*1, *delta*2′ are characterized in the following lemma.

Lemma 5 Suppose $bu \in SUF(pat)$, $b \in \Sigma$. Let v be the shortest string in Σ^+ which satisfies either (1) or (2):

(1) $\exists c \in \Sigma - \{b\} \,:\, cuv \in SUF(pat)$,

(2) $\exists y \in SUF(u) \,:\, yv = pat$.

Put $st = state(u)$. Then

$$delta2'(st) = |u| + |v|\,.$$

On the other hand, let $a \in \Sigma$ and let w be the shortest string in Σ^* which satisfies either (3) or (4):

(3) $aw \in SUF(pat)$,

(4) $w = pat$.

Then

$$delta1(a) = |w|\,.$$

Commentz-Walter [3] presented an extension of the BM algorithm to multiple patterns. She also used a pmm. Her method, essentially, differs from the Uratani method only in the amount by which the patterns are shifted if a mismatch occurs. Suppose that we have just matched the text substring $a_{i+1} \ldots a_j$ backward and we have a mismatch at a_i . To compute the amount, the following three cases are considered in [3]:

(1) The matched string $a_{i+1} \ldots a_j$ occurs in some pattern.

(2) A suffix of the matched string $a_{i+1} \ldots a_j$ equals to a prefix of some pattern.

(3) The mismatched character a_i occurs in some patterns.

Obviously, the Uratani method often shifts the patterns farther than the C-W method does.

Roughly speaking, (1) and (2) correspond to $delta2$ of the BM, and (3) to $delta1$. That is, if a mismatch occurs, the C-W (and the BM) tries to find occurrences of the matched string and of the mismatched character, separately. Therefore the Uratani method is better than the C-W.

We should point out that the construction of the C-W algorithm is incomplete. The amount concerning the case (2) is denoted by $shift2$. As described before, $shift2$ can be computed by

$$shift2(st) = \begin{cases} minlen\,, & \text{if } st = 0; \\ \min[\{depth(r) \mid r \in B(st)\} \cup \{shift2(fath(st)) + 1\}\,], & \text{if } st > 0\,, \end{cases}$$

where

$$B(st) = \{r \in state(K) \mid r \vdash_{\phi}^{+} st\}.$$

C-W described that $B(st)$ can be replaced by

$$B'(st) = \{r \in state(K) \mid \phi(r) = st\}.$$

However, it is not correct. (Consider the value of $shift2(12)$ in **Fig. 7.**)

7 Conclusion

We introduced Uratani's string searching algorithm for multiple patterns. If the lengths of the patterns and the alphabet size are sufficiently large, his algorithm runs quite fast. We have proved the correctness of the algorithm. Moreover, we have shown that his algorithm is not only an extension of the BM algorithm but also better than it for a single pattern.

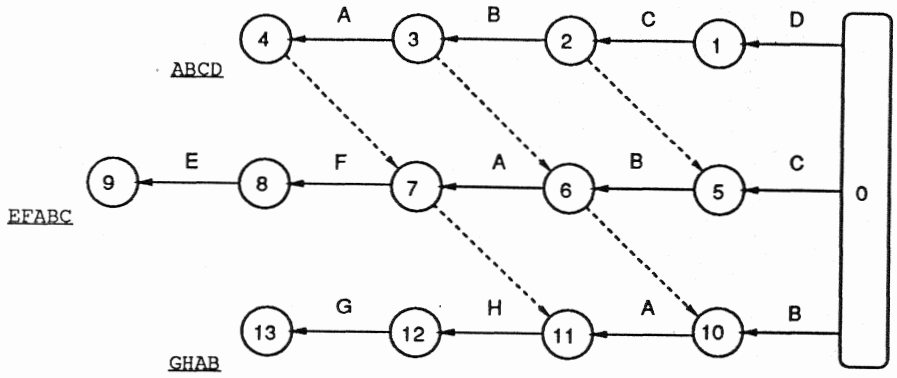


Fig. 7.

In the worst case, the Uratani algorithm does not run in linear time. It can be improved to do so by using a similar idea to [4], but the overhead is very high.

References

[1] Aho, A. V. , Corasick, M. J. : Efficient String Matching : An Aid to Bibliographic Search, *Comm. ACM,* Vol. **18**, No. **6** (1975), pp. 333-340.

[2] Boyer, R. S. , Moore, J. S. : A Fast String Searching Algorithm, *Comm. ACM,* Vol. **20**, No. **10** (1977), pp. 762-772.

[3] Commentz-Walter, B : A String Matching Algorithm Fast on the Average, Lecture Notes in Computer Science, Vol. **71**, (1979), pp. 118-132, Springer.

[4] Galil, Z. : On Improving the Worst Case Running Time of the Boyer-Moore String Searching Algorithm, *Comm. ACM,* Vol. **22**, No. **9** (1979), pp. 505-508.

[5] Knuth, D. E. , Morris, J. H. , Pratt, V. R. : Fast Pattern Matching in Strings, *Tech. Rep. CS* 440, Comput. Sci. Dep. , Stanford Univ. (1974).

[6] Knuth, D. E. , Morris, J. H. , Pratt, V. R. : Fast Pattern Matching in Strings, *SIAM J. Comput.,* Vol. **6**, No. **2** (1977), pp. 323-350.

[7] Kowalski, G. , Meltzer, A. : New Multi-Term High Speed Text Search Algorithms, *First International Conference on Computers and Applications* (CAT. No. 84ch2039-6), pp. 514-522, XIV+905 (1984).

[8] Uratani, N. : A Fast Algorithm for Matching Patterns, IECEJ Technical Report, SS87-27 (1988) (In Japanese).