A Fast Matching Algorithm for Patterns with Pictures

Takeda, Masayuki
Interdisciplinary Graduate School of Engineering Sciences Kyushu University

https://hdl.handle.net/2324/3117

出版情報：RIFIS Technical Report. 11, 1989-03-03. Research Institute of Fundamental Information Science, Kyushu University
バージョン：
権利関係：

RIFIS-TR-CS-11

RIFIS Technical Report

A Fast Matching Algorithm for Patterns with Pictures

Masayuki Takeda

March 3, 1989

Research Institute of Fundamental Information Science
Kyushu University 33
Fukuoka 812, Japan
E-mail: take@rifis1.sci.kyushu-u.junet Phone: 092(641)1101 Ext.4484

A Fast Matching Algorithm for Patterns with Pictures

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

March 3, 1989

Abstract

The pattern matching problem is to find all occurrences of patterns in a text string. In this paper, we consider patterns with pictures. For example, let A be a picture for `a`, `b`, ..., `z`, and N for `0`, `1`, ..., `9`. Then we consider patterns such as `ab`NN, `a7`$NNNNA$, ..., etc. For multiple string patterns, the Aho-Corasick algorithm, which uses a finite state pattern matching machine, is widely known to be quite efficient. As a natural extension of this algorithm, we present an efficient matching algorithm for multiple patterns with pictures. A pattern matching machine can be built easily and quickly, and then it runs in linear time proportional to the text length as the Aho-Corasick achieves. Moreover, we give a validity proof of our algorithm.

1. Introduction

The pattern matching problem is to find all occurrences of character strings, called *patterns*, in another string, called a *text*. In this paper, we consider a matching problem for patterns with *pictures*. For example, let A be a picture for `a`, `b`, ..., `z`, and N for `0`, `1`, ..., `9`. Then we deal with patterns like `ab`NN, `a7`NNNNA, ..., etc. Let us call such patterns *picture-patterns.*

For string patterns, three matching algorithms are widely known: the Knuth-Morris-Pratt [4], the Boyer-Moore [3], and the Aho-Corasick [1]. While the first two are for a single pattern, the third can simultaneously deal with multiple patterns. In this method, from a collection of patterns a finite state machine is built which recognizes all occurrences of the patterns in a single pass through the text. Such a machine is called a *pattern matching machine*, pmm for short. It runs in linear time

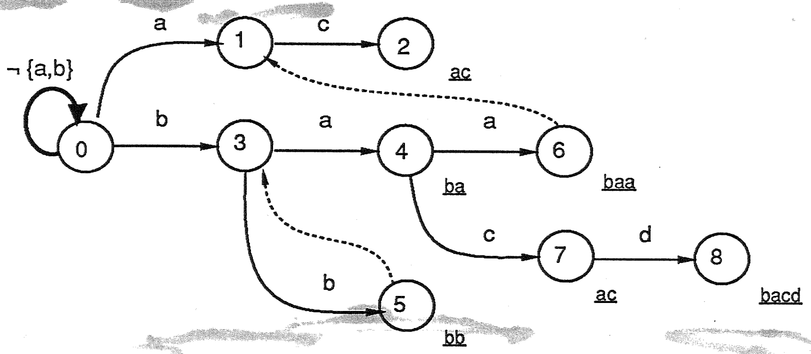


Fig. 1: The pmm for {ac, ba, bb, baa, bacd}

proportional to the text length. Moreover, the construction of the pmm takes only linear time proportional to the sum of the lengths of the patterns.

Fig. 1 shows the pmm for patterns ac, ba, bb, baa, bacd. The solid arrows represent the *goto* function, and the broken arrows represent the *failure* function, where broken arrows to the state 0 from all but the states 0, 5 and 6 are omitted. The underlined strings below the states mean the *output* function.

Consider how to construct a pmm for multiple picture-patterns. It is easy when the patterns consist only of pictures. We can construct a pmm easily if we take each picture as a character. However, it is difficult for patterns with both characters and pictures.

In this paper, we present an algorithm to construct an efficient pmm for multiple picture-patterns. A machine can be built easily and quickly, where the number of states is reasonably decreased without the state minimization technique. Moreover, we can easily transform it into a deterministic finite automaton in the same manner as the Aho-Corasick [1].

2. Observations

We begin by defining the *picture-pattern matching problem.*

Definition 1. Let Σ be an alphabet and let $\Delta = \{A_1, A_2, \ldots, A_p\}$, where $\emptyset \neq A_i \subseteq \Sigma$ and $A_i \cap A_j = \emptyset$ $(i \neq j)$. Each A_i is called a *picture.* A *pattern* is an element in the set $(\Sigma \cup \Delta)^+$. We then extend the ordinary pattern matching problem as follows:

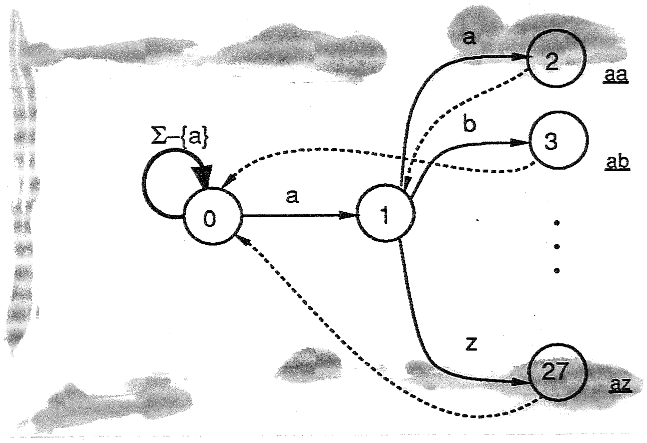
> Given a pattern $X_1 X_2 \ldots X_m$ $(X_i \in \Sigma \cup \Delta)$ and a text $a_1 a_2 \ldots a_n$ $(a_i \in \Sigma)$, to find all indices j with $a_j a_{j+1} \ldots a_{j+m-1} \in X_1 X_2 \ldots X_m$.

We consider this problem for multiple patterns.

Now, let us consider to solve this problem by using the Aho-Corasick type pmm. From here on, we assume that

$$A = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}, \quad N = \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}\}, \quad \Sigma = A \cup N, \quad \Delta = \{A, N\}.$$

Method 1. The naive solution is to construct the pmm for all strings in patterns. That is, if the given pattern is aA, then we construct the pmm for strings aa,ab,. . .,az as below.

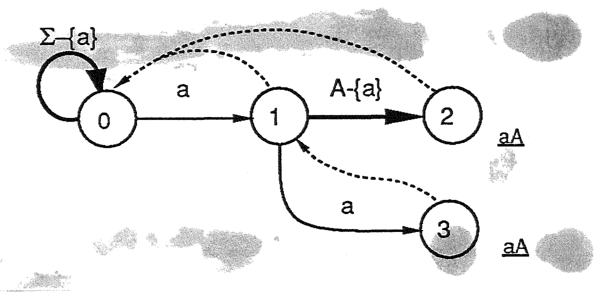


However, for a pattern A^m the number of states of the pmm is

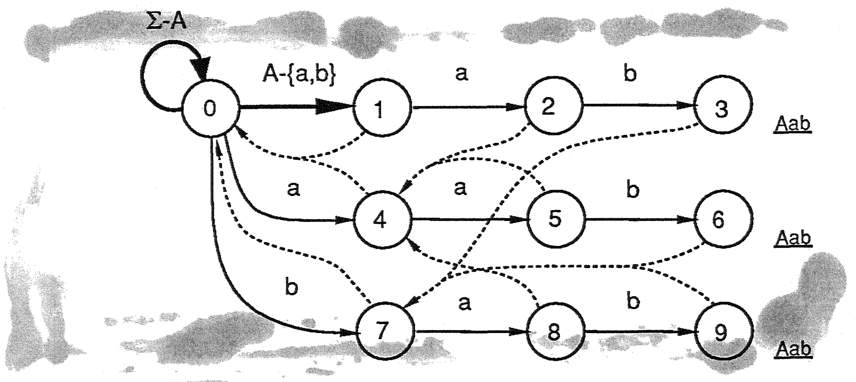
$$1 + 26 + 26^2 + \ldots + 26^m = \frac{26^m - 1}{25}.$$

This method thus increases the number of states exponentially.

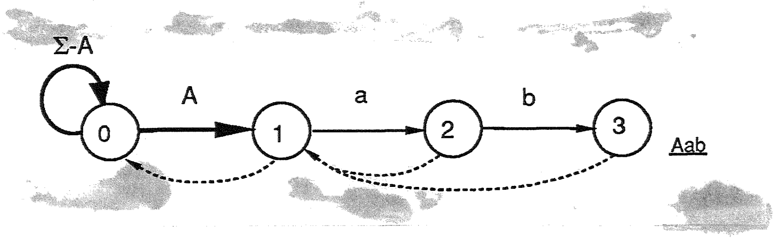
Method 2. Another solution is as follows: We determine the family of disjoint *sub-pictures* from Δ and the characters which occur in the patterns, then we construct a pmm by taking each of these sub-pictures as a characters. For the pattern aA, we divide A into $\{\mathtt{a}\}$ and $A - \{\mathtt{a}\}$ to obtain:



However, for the pattern Aab, since A is divided into $\{\mathtt{a}\}$, $\{\mathtt{b}\}$ and $A - \{\mathtt{a}, \mathtt{b}\}$, we obtain:



This pmm is not efficient, because the following pmm suffices for Aab:



In this case, on the goto edge labeled A from 0, it is not necessary to distinguish each character in A, hence there is no need to make the edge branch off.

These observations tell us, for each goto edge labeled by a picture, to make it branch off to other states only when necessary. To do this, during the construction of the failure function, we shall make such edges branch off according to the values of the failure function. Next section describes our algorithm, based on this idea, to construct an efficient pmm.

3. The algorithm

The algorithm to construct the pmm consists of two parts, which are summarized in Algorithm 1 and 2. We illustrate their behaviors by the following examples.

Example 1. Suppose aAbA is the pattern. In the first part, we treat each picture (denoted by an outline letter) as a character, and obtain the graph:

Algorithm 1
input: A collection of patterns $\Gamma = \{\alpha_1, \dots, \alpha_k\}$.
output: Partially computed functions g and out.
method:

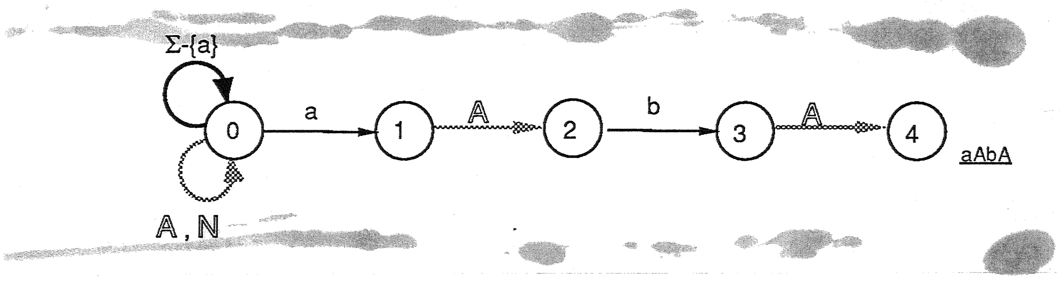
```
begin
    $nst := 0$;
    for $i := 1$ to $k$ do $enter(\alpha_i)$;
    for $\forall X \in \Delta \cup \Sigma$ such that $g(0, X) =$ fail do
        $g(0, X) := 0$
end


procedure $enter(B_1 B_2 \dots B_m)$:
begin
    $state := 0$;
    for $j := 1$ to $m$ do
        if $g(state, B_j) \neq$ fail then
            $state := g(state, B_j)$
        else
            begin
                $g(state, B_j) := nst$;
                $state := nst$;
                $nst := nst + 1$
            end;
    $out(state) := \{B_1 B_2 \dots B_m\}$
end
```

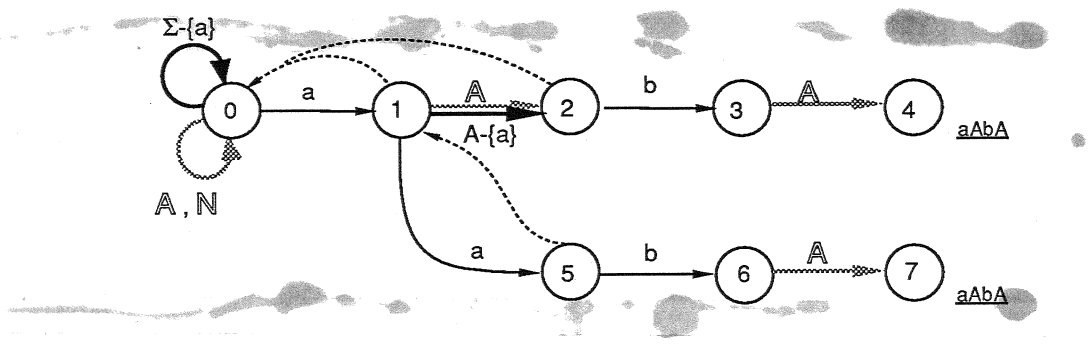
Algorithm 2
input: Partially computed functions g and out.
output: Functions g,f and out.
method:

```
begin
    $queue$ := empty;
    for $\forall c \in \Sigma$ such that $g(0,c) = s \neq 0$ do
        begin
            $queue := queue \cdot s$;
            $f(s) := 0$
        end;
    for $\forall X \in \Delta$ such that $g(0,X) = s \neq 0$ do
        begin
            $queue := queue \cdot s$;
            $f(s) := 0$;
            for $\forall c \in X$ do
                if $g(0,c) = t \neq 0$ then
                    $copy\_subtree(s,t)$
                else
                    $g(0,c) := s$
        end;
    while $queue \neq$ empty do
        begin
            let $queue = r \cdot tail$
            $queue := tail$;
            for $\forall c \in \Sigma$ such that $g(r,c) = s \neq$ fail do
                begin
                    $queue := queue \cdot s$;
                    $fst := f(r)$;
                    while $g(fst,c) =$ fail do $fst := f(fst)$;
                    $f(s) := g(fst,c)$;
                    $out(s) := out(s) \cup out(f(s))$
                end;
            for $\forall X \in \Delta$ such that $g(r, X) = s \neq$ fail do
                begin
                    $queue := queue \cdot s$ ;
                    $fst := f(r)$;
                    while $g(fst, X) =$ fail do $fst := f(fst)$;
                    $f(s) := g(fst, X)$;
                    for $\forall c \in X$ do
                        if $g(r, c) = t \neq$ fail then
                            $copy\_subtree(s, t)$
                        else
                            begin
                                $fst := f(r)$;
                                while $g(fst, c) =$ fail do $fst := f(fst)$;
                                $st := g(fst, c)$;
                                if $st \neq f(s)$ then
                                    begin
                                        $new := nst$;
                                        $nst := nst + 1$;
                                        $copy\_subtree(s, new)$;
                                        $g(r, c) := new$;
                                        $queue := queue \cdot new$;
                                        $f(new) := st$;
                                        $out(new) := out(new) \cup out(f(new))$
                                    end
                                else
                                    $g(r, c) := s$
                            end;
                    $out(s) := out(s) \cup out(f(s))$
                end
        end
end
```

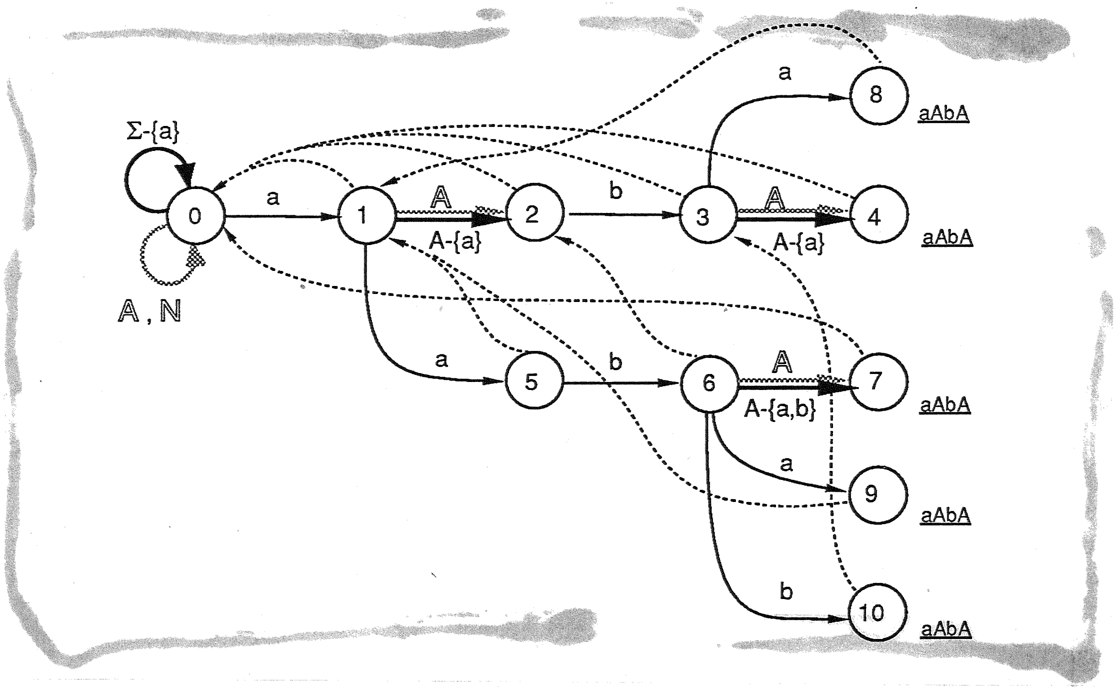
```
procedure $copy\_subtree(st1, st2)$ :
begin
    $queue2 := \{(st1, st2)\}$;
    while $queue2 \neq$ empty do
        begin
            let $queue2 = (r1, r2) \cdot tail$
            $queue2 := tail$;
            $out(r2) := out(r2) \cup out(r1)$;
            for $\forall X \in \Delta \cup \Sigma$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



In the second part, we construct the failure function and make goto paths branch off according to need. We first set $f(1) = 0$ since it is the state of depth 1. Then, we would compute the failure function for all states recursively, in nearly the same manner as the Aho-Corasick method. We would set $f(2) = 0$. However, if the input symbol on which we have made a goto transition from 1 to 2 is a, the failure value should be 1; Otherwise, 0. Therefore we shall make the edge branch off only for a to obtain:

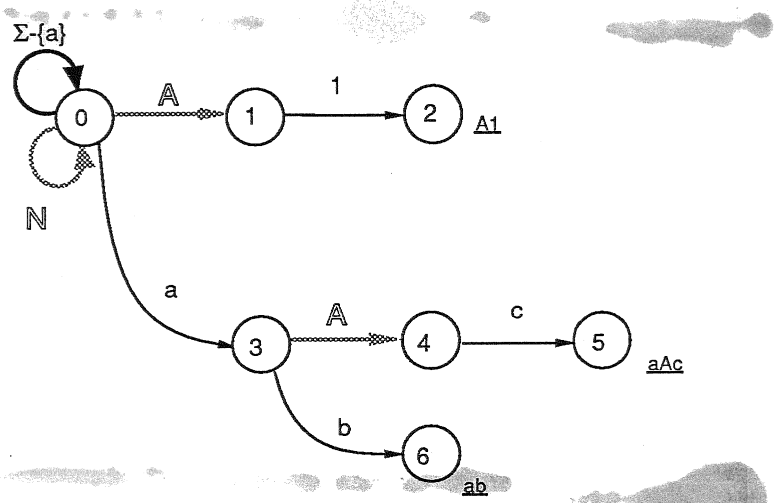


Note that we have copied the subtree whose root is 2 to the new state 5. Continuing in this fashion, we obtain:

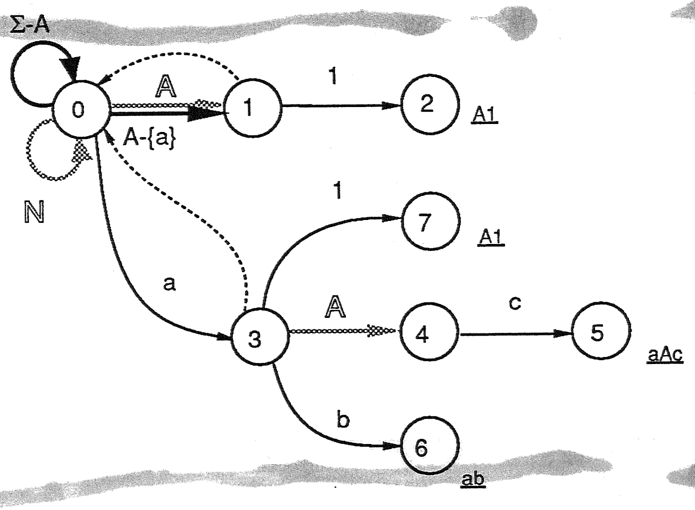


We next give an example for multiple patterns.

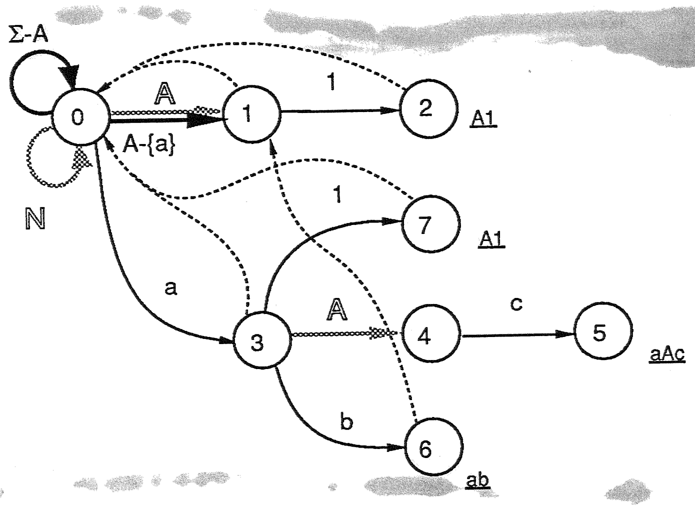
Example 2. Suppose that $A1$, $\mathtt{a}A\mathtt{c}$, $\mathtt{ab}$ are the patterns. In the first part, we obtain the graph:



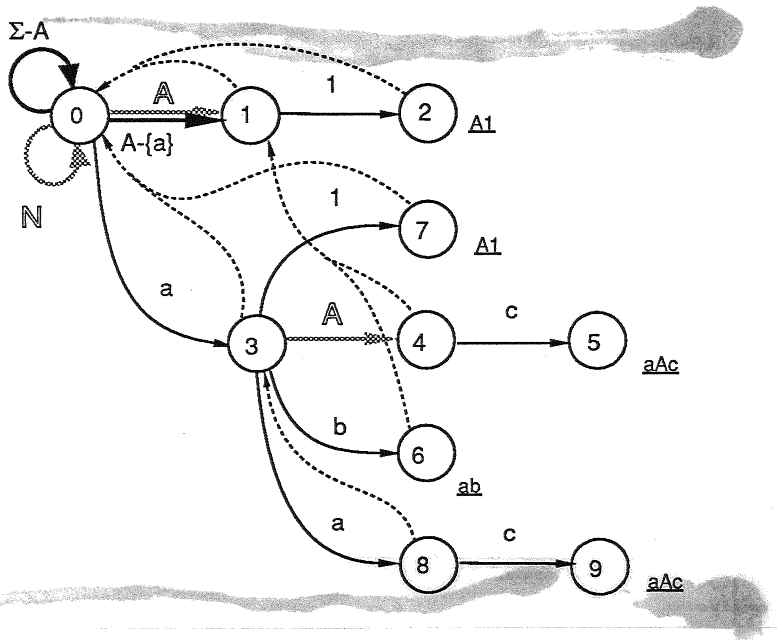
In the second part, we initially set the graph as below:



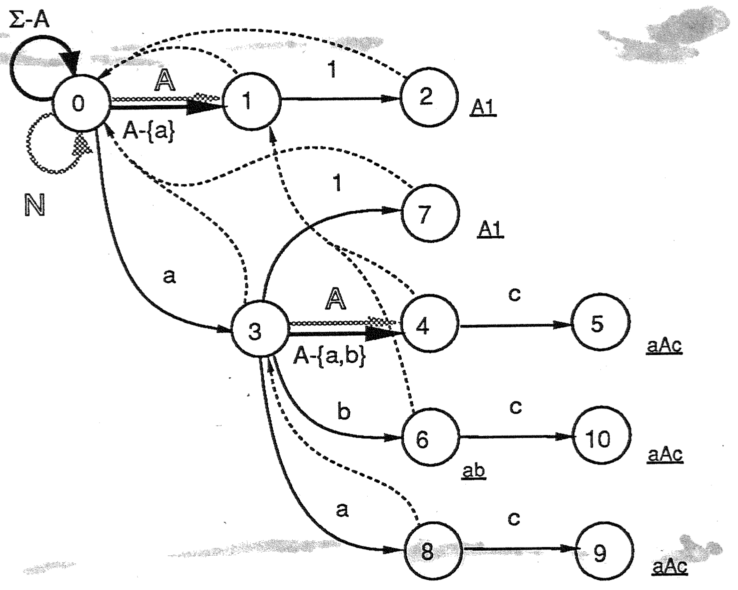
Note that the subtree whose root is 1 has been copied to the state 3. We then inspect the goto edge from 1 and get $f(2) = 0$. Now, we inspect all edges from 3. We first inspect edges labeled by characters, and we get:



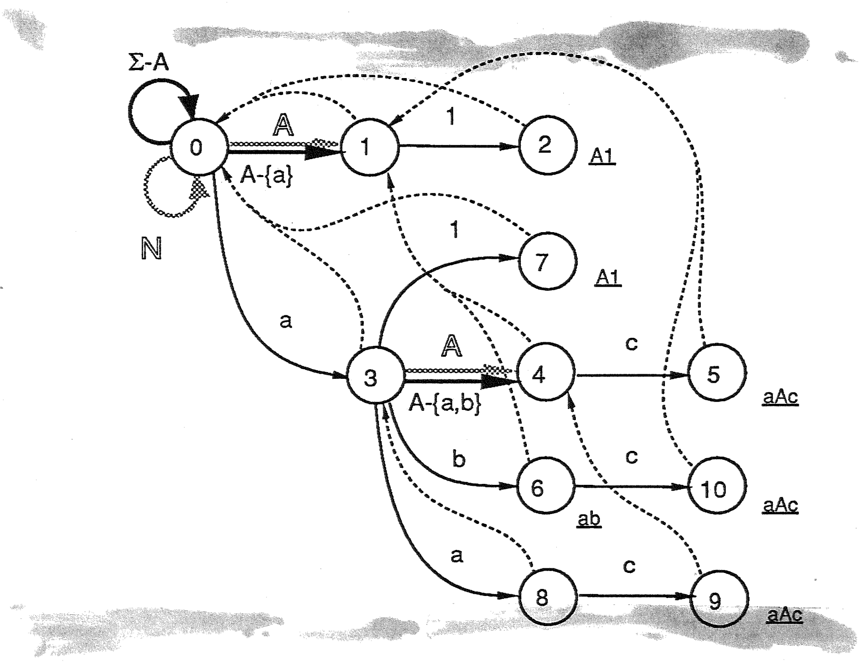
We next inspect edges labeled by pictures, and we set $f(4) = 1$. Now we determine the next state from 3 for each character in the picture A. For the character a, the failure value should be 3, hence we make the edge branch off as below:



For b, since there already exists an edge labeled by b to the state 6, we shall copy there the subtree whose root is 4. For the other characters in A, the next states should be all 4. Thus we obtain:



Continuing in this fashion, we can complete the pmm as is shown below:



Thus, our algorithm simply produces an efficient pmm from multiple picture-patterns. The text searching algorithm, which is summarized in Algorithm 3, is exactly the same as Aho-Corasick's. Moreover, it is easy to transform a pmm into a deterministic finite automaton, in the same manner as the Aho-Corasick [1].

Algorithm 3

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: Locations at which patterns occur in T.

method:

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

4. Validity of the Algorithm

Definition 2. We say the pmm M is *valid* for a set of patterns Γ when with M Algorithm 3 indicates that pattern α ends at position i of text T if and only if there exists $y \in \alpha$ with $T = uyv$ and the length of uy is i.

This section shows the pmm produced by Algorithm 1, 2 is valid. We first have the following lemma.

Lemma 1. Let $b_1, b_2, \ldots, b_m \in \Sigma$. Then, the following are equivalent.

(1) There exists a pattern α such that $b_1 b_2 \ldots b_m y \in \alpha$, for some $y \in \Sigma^*$.

(2) There exists a sequence of non-zero states $r_1, r_2, \ldots, r_m$ such that

$$g(0, b_1) = r_1, \quad g(r_i, b_{i+1}) = r_{i+1}\ (1 \leq i < m).$$

Proof. It is clear from the construction of the goto function.□

We shall define sets and a mapping as follows: For a given set of patterns $\Gamma = \{\alpha_1, \ldots, \alpha_k\}$, we put $K = \alpha_1 \cup \ldots \alpha_k$. K is the set of all strings to be searched. We say that u is a *prefix* and v is a *suffix* of the string uv. Then we let W be the set of all prefixes of elements in K. We define a mapping *state* from W to the set of non-negative integers by

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

where ε denotes the *empty word*. Note that *state* is *well-defined* from the result of Lemma 1. We also put $S = \{state(u) | u \in W\}$, which is the set of all states reachable from the initial state 0. For all $st \in S$, we put $rep(st) = \{u \in W | state(u) = st\}$. It should be noted that for any $s, t \in S$, we have $rep(s) \cap rep(t) = \emptyset$ if $s \neq t$.

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

Lemma 2. Let $\alpha = X_1 \ldots X_m$ be a pattern. Then, for any j with $1 \leq j \leq m$, there uniquely exists a nonempty subset I of S such that

$$X_1 \ldots X_j = \bigcup_{st \in I} rep(st), \quad depth(st) = j\ (\forall st \in I)\,,$$

where $depth(st)$ denotes the depth of st, i.e., the length of the shortest path from 0 to st.

Proof. Using the result of Lemma 1, we prove this by induction on j.□

This lemma claims that the goto function g is valid in a simple sense. But we have to show that Algorithm 2 produces sufficient branchings of the goto paths to compute the valid failure function.

Lemma 3. Suppose that $s \in S$, $s \neq 0$. Let u be a string in $rep(s)$. Let v be the longest string in $(SUF(u) - \{u\}) \cap W$, where $SUF(u)$ denotes the set of suffixes of u. Then, $v \in rep(f(s))$.
Proof. By induction on the depth of s.□.

Concerning with the output function *out*, the following lemma holds.

Lemma 4. For all $s \in S$ with $s \neq 0$,

$$out(s) = \{\alpha \in \Gamma | \exists x, \exists y \in \Sigma^* : xy \in rep(s), y \in \alpha\} .$$

Proof. It follows from the construction of *out* and from the results of Lemmas 2, 3. □

The following lemma characterizes the behavior of Algorithm 3 on a text $T = a_1 a_2 \dots a_n$.

Lemma 5. After jth pass through the **for**-loop, Algorithm 3 will be in state s if and only if $rep(s)$ contains the longest string in $SUF(a_1 a_2 \dots a_j) \cap W$.
Proof. By induction on the depth of s. □

We now have the following theorem.

Theorem 1. The pmm M produced by Algorithm 1, 2 is valid.
Proof. By Lemmas 4, 5. □

5. Time complexity

It is obvious that Algorithm 3 runs in linear time proportional to the text length. We then discuss the time complexity of Algorithm 1, 2, which are for the construction of pmm. Clearly, Algorithm 1 takes only linear time proportional to the sum of the lengths of the patterns. Algorithm 2 does so if the patterns consist only of characters, or only of pictures. However, it is not so simple when the patterns contain both characters and pictures. Our algorithm is designed to do branchings of the goto paths according to need during the construction of the failure function so as to decrease the number of states. The cost varies depending on how the paths will branch off. However, even in the worst case, it is bounded by those of Method 1 and 2 described in Section 2.

Consider the cost of Method 1. We denote by $\#(X)$ the number of elements in a set X. Then, for a pattern $\alpha = X_1X_2\dots X_m$, the number of strings belonging to α is given by

$$\#(\alpha) = \#(X_1)\cdot\#(X_2)\cdot\dots\cdot\#(X_m).$$

Note that $\#(\alpha) = 1$ if α is a string pattern. We also denote by $|\alpha|$ the length of a pattern α. Suppose that $\Gamma = \{\alpha_1, \alpha_2, \dots, \alpha_k\}$ is the set of patterns. Then, Method 1 takes to construct the pmm linear time proportional to

$$\sum_{i=1}^{k}\#(\alpha_i)\cdot|\alpha_i|.$$

We then consider Method 2. Let c_i be the number of different characters in A_i appearing in the patterns. Let $d(X) = 1$, if $X \in \Sigma$; $c_i + 1$, if $X = A_i$. Put $d(\alpha) = d(X_1)\cdot d(X_2)\cdot\dots\cdot d(X_m)$, for a pattern $\alpha = X_1X_2\dots X_m$. Then, the cost of Method 2 is linearly proportional to

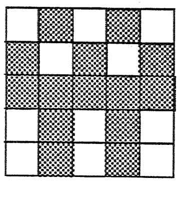
$$\sum_{i=1}^{k} d(\alpha_i)\cdot|\alpha_i|.$$

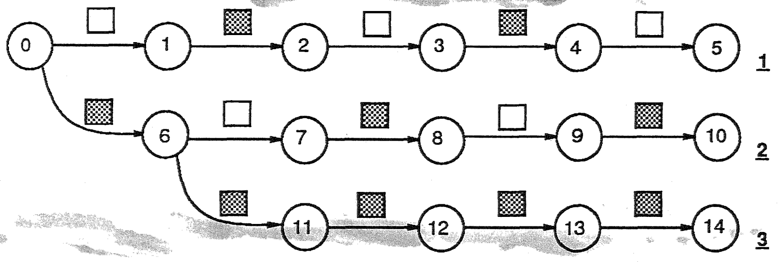
Unless a large number of different characters appear in the patterns, $d(\alpha_i)$ is much smaller than $\#(\alpha_i)$. Moreover, if many characters appear in the patterns, accordingly the number of occurrences of pictures in the patterns is small, hence $d(\alpha_i)$ will be 1 frequently.

Our method is better than these two methods.

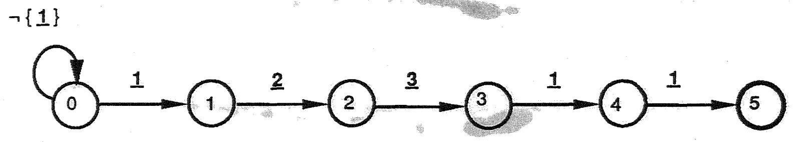
6. An application to the two-dimensional pattern matching

In the two-dimensional pattern matching problem, both pattern and text are two-dimensional arrays of characters. Bird [2] described an algorithm to solve this problem by using the Aho-Corasick method. Suppose that Σ = {▩, □}, and let the pattern be





(a) row-matching

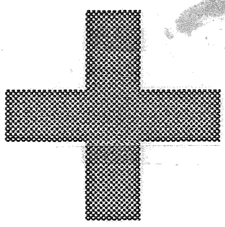


(b) column-matching

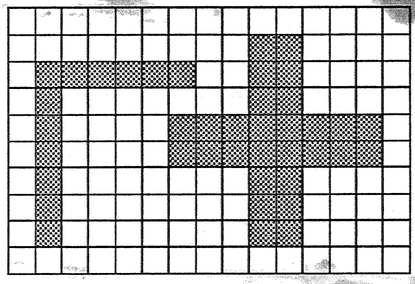
Fig. 2: The Bird method

The method by the Bird [2], regarding each row as a string pattern, builds a pmm as shown in Fig. 2 (a). Then, the pmm runs on the text row by row searching for the rows of the pattern array (*row-matching*). On the other hand, the machine shown in Fig. 2 (b) is used to determine whether or not the entire pattern array occurs in the text (*column-matching*). The algorithm takes $O(n_1 \cdot n_2)$ time to find all occurrences of the pattern in a text of size $n_1 \times n_2$.

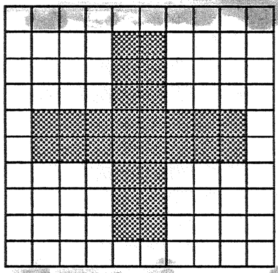
However, this method has the following problem: Suppose that we would detect



in the text

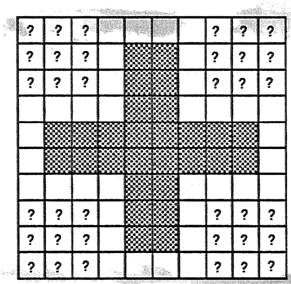


If we search for the rectangular pattern



by using the above method, we can not find it in the text.

Contrary to this, our method is able to deal with the pattern containing wild cards, i.e., pictures '?', such as



hence we can find the cross in the text.

7. Conclusion

We have presented an efficient matching algorithm for patterns with pictures. Since it is a natural extension of the Aho-Corasick algorithm, it is applicable to various problems.

Japanese texts consist of both 1-byte and 2-byte characters with shift codes. Shinohara and Arikawa [5] have developed an algorithm to construct the pmm for such texts, based on the Aho-Corasick algorithm. Taking each byte as a input symbol, the pmm runs on a Japanese text without loosing the efficiency. If we combine our algorithm with this, then we can deal with not only 1-byte pictures but also some 2-byte pictures, e.g., *kanji*, *hiragana*, etc., by regarding them as concatenations of two 1-byte pictures.

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] Bird, R. S. : Two Dimensional Pattern Matching, *Inf. Process. Lett.*, Vol. **6**, No. **5** (1977), pp. 168-170.

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

[4] 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.

[5] Shinohara, T. and Arikawa, S. : Pattern Matching Machines for Japanese Texts, Research Report No. **110** (1986), Research Institute of Fundamental Information Science, Kyushu University.