A FAST MATCHING ALGORITHM FOR PATTERNS WITH PICTURES

Takeda, Masayuki
Department of Information Systems, Interdisciplinary Graduate School of Engineering Science, Kyushu University

https://doi.org/10.5109/13427

出版情報：Bulletin of informatics and cybernetics. 25 (3/4), pp.137-153, 1993-03. Research Association of Statistical Sciences
バージョン：
権利関係：

Bulletin of Informatics and Cybernetics, Vol. 25, No. 3~4, 1993

A FAST MATCHING ALGORITHM FOR PATTERNS WITH PICTURES

By

Masayuki TAKEDA[†]

Abstract

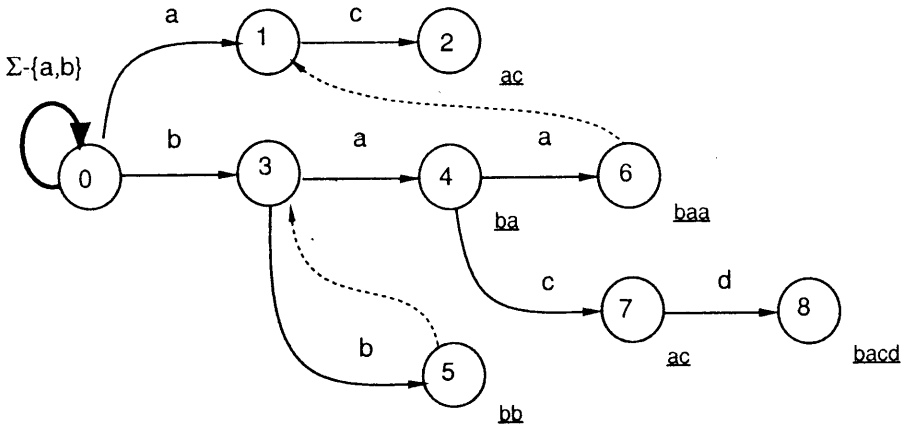
The pattern matching problem is to find all occurrences of patterns in a text string. An extended problem is considered for patterns with pictures, where a picture is a don't-care that matches only the symbols in the set it represents. For example, let A be a picture **a**, **b**, ..., **z**, and N for **0**, **1**, ..., **9**. Then, **ab***NN*, **a7***NNNNA*, ... are patterns. For multiple string patterns, the Aho-Corasick algorithm, which uses a finite state pattern matching machine, is widely known to be quite efficient. A fast matching algorithm for multiple patterns with pictures is presented as a natural extension of the AC algorithm. A pattern matching machine can be built easily, and then it runs in linear time proportional to the text length as the AC algorithm achieves. Moreover, a correctness proof of the algorithm is given.

1. Introduction

The pattern matching problem is to find all occurrences of *patterns* in a *text*, where the patterns and the text are strings of symbols. The most important techniques are pattern matching algorithms due to Knuth, Morris and Pratt [19], Boyer and Moore [11], and Aho and Corasick [3], respectively. The first two are for a single pattern, but the third can simultaneously deal with multiple patterns. In this method, from a collection of patterns a finite state pattern matching machine (PMM, in short) is built which simultaneously recognizes all occurrences of the patterns in a single pass through a text. Figure 1 shows the PMM for patterns {**ac, ba, baa, bacd**}, and Fig. 2 shows its behavior in processing the text "**cbaac**". The construction of PMM takes linear time proportional to the sum of the lengths of the patterns. The text processing requires only linear time proportional to the text length, independently of the number of patterns. This property is of supreme importance, especially when we should cope with very large texts, e.g., the text files of total size 26MB that are the whole volumes written by a novelist Thomas Mann [17, 8], or DNA files of length $10^6 \sim 10^9$.

In this paper we consider an extended pattern matching problem where patterns contain *pictures*. A *picture* is a kind of don't-care, but matches only the symbols in the set it represents. For example, let A be a picture for **a, b, ..., z**, and N for **0, 1, ..., 9**.

[†] Department of Information Systems, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University 39, Kasuga 816, Japan



A PMM consists of the goto, failure, and output functions. The solid arrows represent the goto function. The broken arrows represent the failure function, where broken arrows to state 0 from all states but 0, 5, and 6 are omitted. The underlined strings adjacent to the states mean the outputs from them.

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

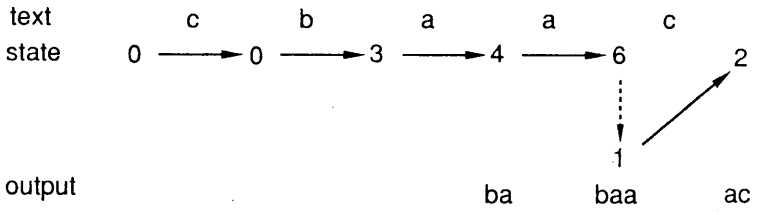


Fig. 2. The behavior of the PMM

We then deal with patterns like **ab***NN*, **a7***NNNNA*, ..., etc. Obviously the AC algorithm can be applied to this problem since the pattern is a finite set of strings [2]. This method, however, makes the number of states of PMM very large. Here we present an algorithm for constructing a more efficient (though not minimum) PMM for multiple patterns with pictures. The construction is easy, and it is a natural extension of the AC algorithm in the sense that, when all the pattern are strings, it produces the same PMM as the AC algorithm does.

In the worst case both the construction time and the number of states may be exponential in the length of pattern. But this is not a crucial problem in the case of very large text, because the pattern length is much smaller than the text length.

The organization of this paper is as follows. Section 2 gives preliminaries. Section 3 defines the extended pattern matching problem and Section 4 presents an algorithm for constructing a PMM for the problem. Section 5 gives a correctness proof of the algorithm. Section 6 gives a theoretical analysis on the time complexity. Section 7 mentions an extension of the algorithm to the two-dimensional pattern matching problem.

This paper is based in large part on [24].

2. Preliminaries

Let Σ be a finite set of *symbols (or characters)*, called the *alphabet*. A *string* is a finite sequence of symbols juxtaposed. The *length* of a string w, denoted by $|w|$, is the number of symbols composing the string. The *empty string*, denoted by ε, is the string consisting of zero symbols. Thus $|\varepsilon| = 0$. The set of all strings over the alphabet Σ is denoted by Σ^*. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. When string w can be written as $w = uv$, strings u and v are called a *prefix* and a *suffix* of w, respectively. Let

$$PRE(w) = \{u \in \Sigma^* | u \text{ is a prefix of } w\},$$
$$SUF(w) = \{v \in \Sigma^* | v \text{ is a suffix of } w\}.$$

These are extended to sets of strings by

$$PRE(X) = \bigcup_{w \in X} PRE(w), \quad SUF(X) = \bigcup_{w \in X} SUF(w),$$

where X is a set of strings.

3. The Problem

Let $\Delta = \{A_1, \ldots, A_p\}$ be a collection of disjoint nonempty subsets of Σ, i.e., $\emptyset \neq Ai \subseteq \Sigma$ and $A_i \cap A_j = \emptyset$ $(i \neq j)$. An element of Δ is called a *picture*. A *pattern* is taken from $(\Sigma \cup \Delta)^+$, and represents a set of strings, which are all the same length. Thus, the *length* of a pattern π can be defined to be m such that $\pi \in (\Sigma \cup \Delta)^m$, and denoted by $|\pi|$.

For example, consider the situation where

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

Then, **ab***NN*, **a7***NNNNA* are patterns, which are of length 4 and 7, respectively.

We say that *a pattern π occurs at position i of a string w* iff $w \in \Sigma^{i-1} \cdot \pi \cdot \Sigma^*$. We now consider the following problem:

> Given a collection of patterns $\Gamma = \{\pi_1, \ldots, \pi_k\}$ and a text T, find all positions of T at which π_i occurs, for $i = 1, \ldots, k$.

Note that if $\Delta = \{\Sigma\}$, a pattern in $(\Sigma \cup \Delta)^+$ is called a pattern with *don't-cares*. It should be stated that Fischer and Paterson [13] presented an algorithm that finds all occurrences of such a pattern of length m in a text of length n in time $O(n \log m \log \log m)$ using the Schonhage-Strassen algorithm [4] for integer multiplication. However, this is not practical because the constant of the proportionality is extremely large. It should also be stated that Abrahamson [1] discussed a more general problem where:

(1) Σ is infinite,
(2) an element of Δ is a nonempty subset of Σ which is either finite or cofinite, and
(3) Δ is allowed not to be disjoint.

He presented a practical fast algorithm for the problem, using arithmetic and bitwise

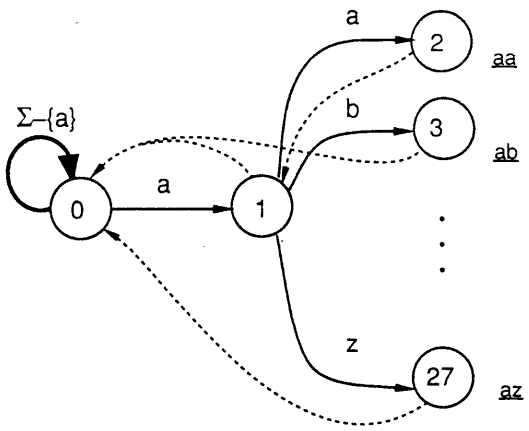
Boolean operations on m-bit integer where m is the length of pattern. If we apply his algorithm to our problem in the case of single pattern, it runs, in typical application, in time $O(n)$ excluding preprocessing time. However, we want to search for multiple patterns very fast at a time. Hence we shall consider constructing a PMM for multiple patterns with pictures.

When the patterns consist only of pictures, we can easily build a PMM by treating each picture as a symbol. However, it is not easy when the patterns contain both symbols and pictures. From here on, we often assume the situation Eq. 1.

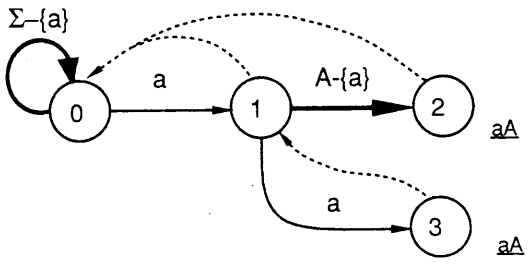
METHOD 1. The naive solution is to construct a PMM for all strings in patterns. For example, if the given pattern is $\mathbf{a}A$, we construct a PMM for strings **aa, ab,** ..., **az** as in Fig. 3(a). However, the number of states of PMM for a pattern A^m is

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

This method thus makes the number of states very large.

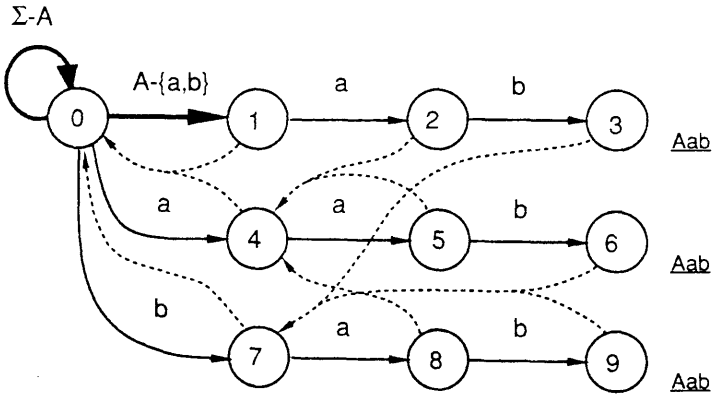


(a) The PMM constructed by Method 1

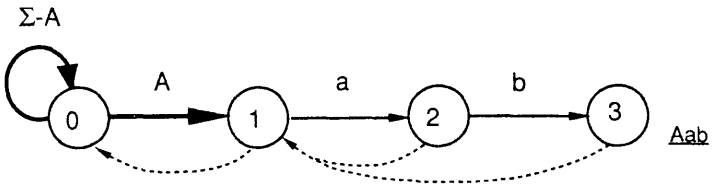


(b) The PMM constructed by Method 2

Fig. 3. Two PMMs for the pattern aA



(a) The PMM constructed by Method 2



(b) The reduced PMM

Fig. 4. Two PMMs for the pattern Aab

METHOD 2. Another solution is as follows: Subdivide the family Δ into a new family Δ' so that there will be a picture $\{\sigma\}$ in Δ', for every symbol σ appearing in the patterns. For example, if the pattern is again $\mathbf{a}A$, we divide A into $\{\mathbf{a}\}$ and $A - \{\mathbf{a}\}$, and obtain the PMM of Fig. 3(b). The method, however, constructs the PMM of Fig. 4(a) for the pattern $A\mathbf{ab}$, dividing A into $\{\mathbf{a}\}$, $\{\mathbf{b}\}$, and $A - \{\mathbf{a}, \mathbf{b}\}$. It is redundant because of the reduced PMM of Fig. 4(b). In this case, on the goto edge labeled A from 0, it is not necessary to distinguish each symbol in A, and therefore to make the edge branch off.

These observations tell us, for each goto edge labeled by a picture, to make it branch off only when necessary. Then, during the construction of the failure function, we shall make such edges branch off according to the values of the failure function. The next section presents a new algorithm, based on this idea, for constructing an efficient PMM.

4. The Algorithm

We illustrate the behavior of the algorithm with the following two examples, where we assume the situation Eq. 1 again. Here we suppose, for simplicity of description, the two dimensional array realization of the goto function.

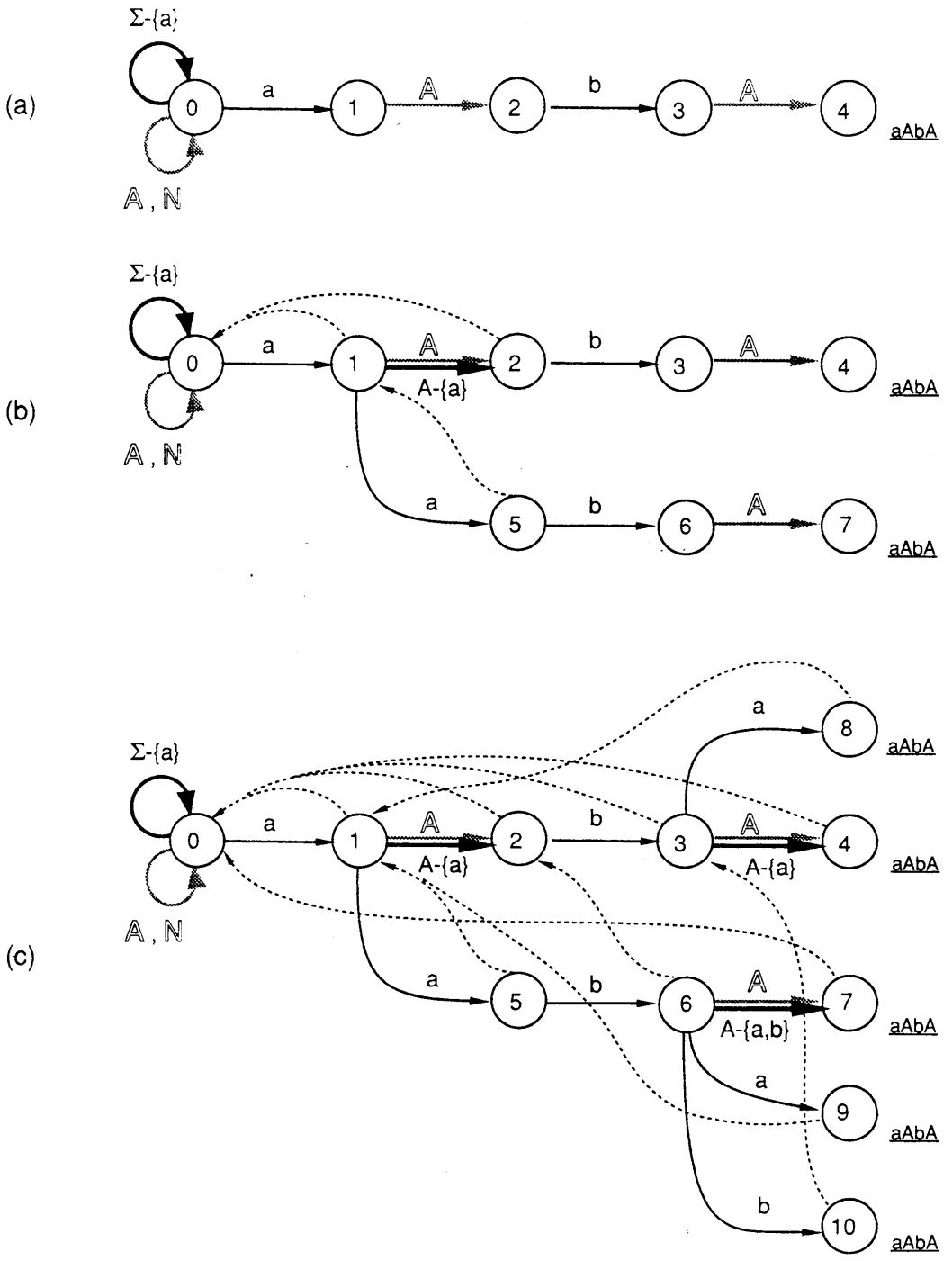


Fig. 5. Construction of the PMM for the pattern aAbA

EXAMPLE 1. Suppose $\mathbf{a}A\mathbf{b}A$ is the pattern. There are two parts to the construction. In the first part, we treat each picture as a symbol, and obtain the graph of Fig. 5(a), where we use gray arrow labeled by outlined letter to distinguish it from the ordinary goto edges. In the second part, we construct the failure function f 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 AC 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 value of the failure function should be 1; Otherwise, it should be 0. We therefore make the edge branch off only for "**a**" (Fig. 5(b)). Note that the subtree of 1 with a root 2 has been copied to new state 5. Continuing in this fashion, we obtain the PMM of Fig. 5(c).

The next is an example for multiple patterns.

EXAMPLE 2. Suppose that $A1$, $\mathbf{a}A\mathbf{c}$, **ab** are the patterns. In the first part, we obtain the graph of Fig. 6(a). In the second part, we initially set the graph as in Fig. 6(b). Note that the subtree of 0 with a root 1 has been copied to state 3. We check the goto edge from 1 and get $f(2) = 0$. Now, we check the edges from 3. Since "**b**" is in A, we shall copy to state 6 the subtree of 3 with a root 4. We then check each edge labeled by a symbol, and obtain $f(6) = 1$ and $f(7) = 0$. We next check each edge labeled by a picture, and we set $f(4) = 1$. We now determine the next state from 3 for each symbol in the picture A. For the symbol "**a**", the value of failure function should be 3, we therefore make the edge branch off (Fig. 6(c)). For "**b**", there already exists an edge labeled by "**b**" to state 6. For the other symbols in A, the next states should be all 4. Continuing in this fashion, we can complete the PMM as shown in Fig. 6(d).

Now, we describe the algorithm more precisely. In the first part we simply build a graph by treating each picture as a symbol. In the second part we compute from the graph the failure and output functions, and branch off the goto edges if necessary. Remember the recursive construction of the failure function in [3]. Consider a state r with $r \neq 0$ and a symbol/picture σ. Let $r_0, r_1, \ldots, r_n$ be the sequence of states such that

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

We then denote by $F(r, \sigma)$ the value of $g(r_n, \sigma)$. Let $TREE(s)$ denote the tree whose root is state s, $s \neq 0$. The second part of the construction can then be summarized as follows.

Basis. Perform the following actions:

(1) Consider each picture P in Δ such that $g(0, P) = s \neq 0$. For each symbol σ in P such that $g(0, \sigma) = t \neq 0$, add to state t a copy of $TREE(s)$. Namely, new states and edges are added so that there will be a copy of $TREE(s)$ in $TREE(t)$. The outputs of states in $TREE(s)$ are also added.

(2) For each symbol σ in Σ such that $g(0, \sigma) = s \neq 0$, set $f(s) := 0$.

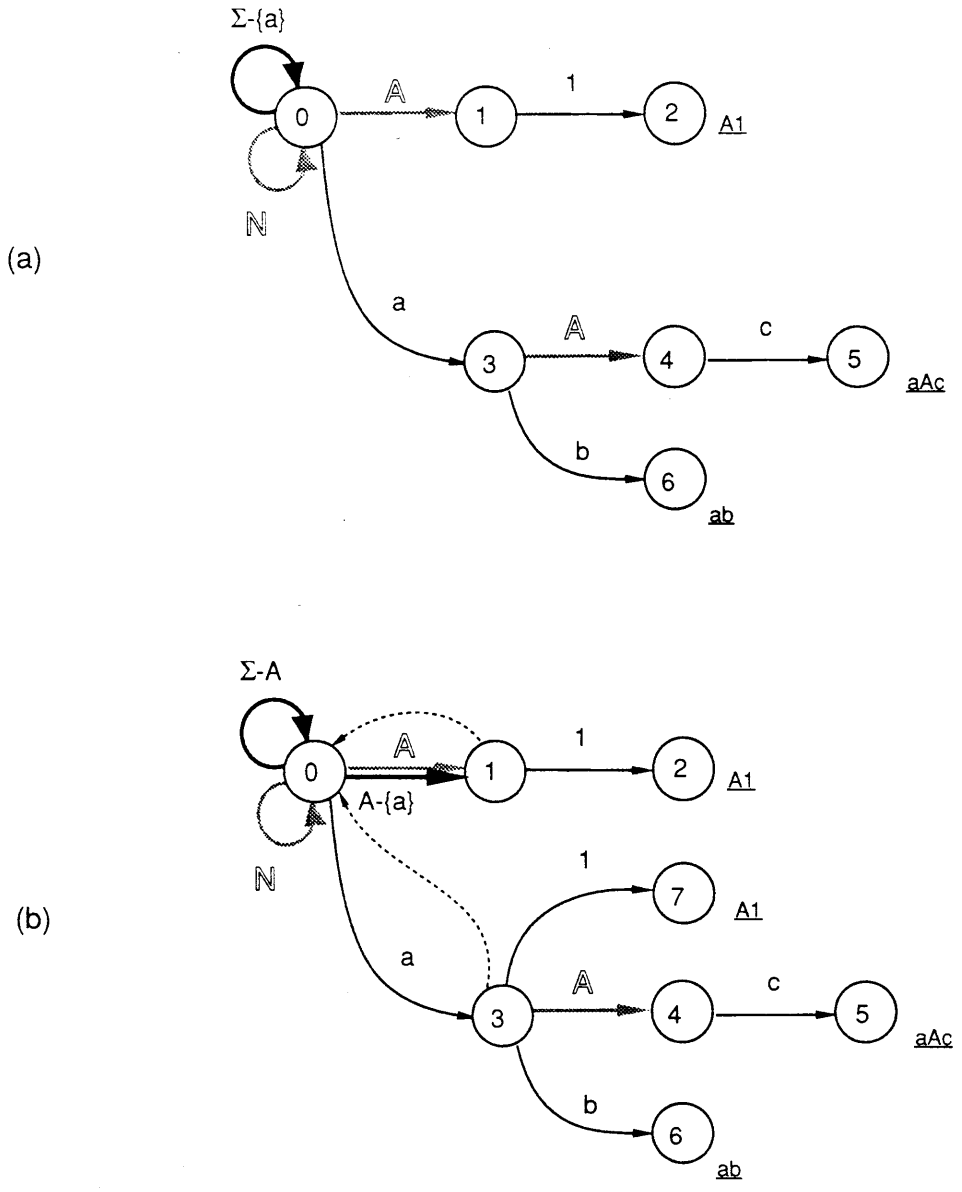
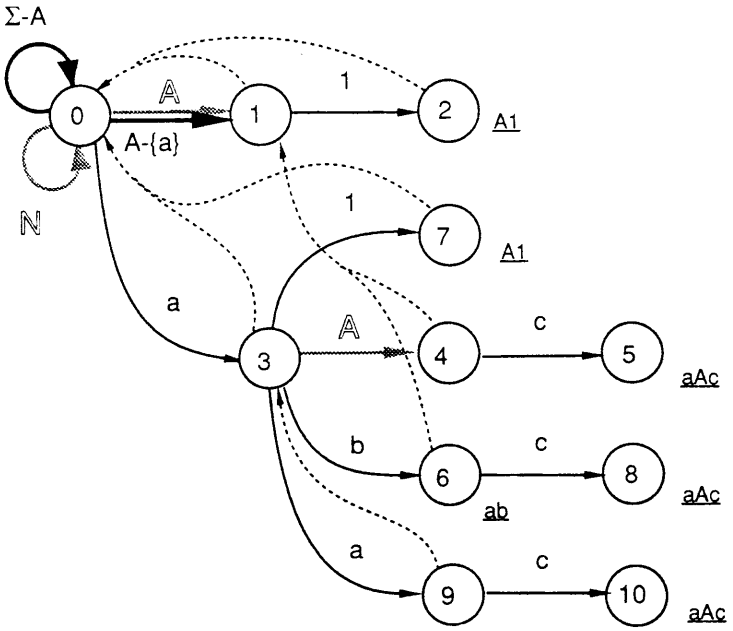


Fig. 6. Construction of the PMM for the patterns {A1, aAc, ab}

(c)



(d)

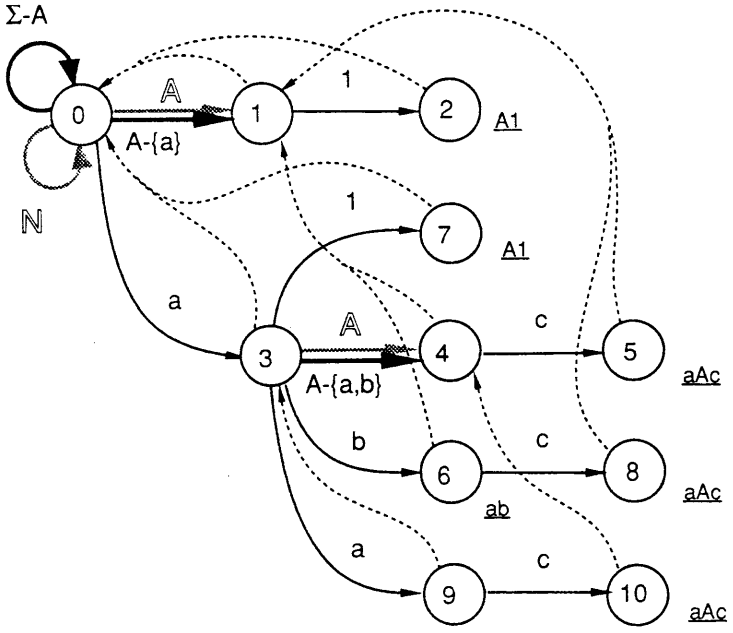


Fig. 6. (continued)

(3) Consider each picture P in Δ such that $g(0, P) = s \neq 0$. Set $f(s) := 0$. For each symbol σ in P such that $g(0, \sigma) = 0$, set $g(0, \sigma) := s$.

Recursive step. Suppose that the computation has been done for all states of depth less than $d (d > 1)$. Consider each state r of depth $d - 1$, and perform the following actions:

(1) Consider each picture P in Δ such that $g(r, P) = s \neq$ **fail**. For each symbol σ in P such that $g(r, \sigma) = t \neq$ **fail**, add to state t a copy of $TREE(s)$.

(2) For each symbol σ in Σ such that $g(r, \sigma) = s \neq$ **fail**, do as follows: Compute $F(r, \sigma)$, and then set $f(s) := F(r, \sigma)$ and $out(s) := out(s) \cup out(f(s))$.

(3) For each picture P in Δ such that $g(r, P) = s \neq$ **fail**, do as follows: Compute $F(r, P)$, and then set $f(s) := F(r, P)$ and $out(s) := out(s) \cup out(f(s))$. Now, for each σ in P such that $g(r, \sigma) =$ **fail**, compute $F(r, \sigma)$, and do as follows:
- If $F(r, \sigma) = F(r, P)$, then set $g(r, \sigma) := s$.
- Otherwise, create a copy of $TREE(s)$ with a root s', and let $g(r, \sigma) := s'$. In addition, set $f(s') := F(r, \sigma)$ and $\text{out}(s') := out(s') \cup out(f(s'))$.

Stage 3 in the recursive step above should be improved as follows: A symbol σ in Σ is said to be *critical* iff σ appears in some pattern in Γ. It is easy to see that if σ in P is not critical, then $F(r, \sigma) = F(r, P)$. Thus we can omit the computation of $F(r, \sigma)$ and simply set $g(r, \sigma) := s$, for all symbols σ that are not critical.

5. Correctness of the Algorithm

A PMM M is said to be *valid* for a set of patterns Γ when M indicates that pattern π ends at position i of text T iff there exists $y \in \pi$ such that $T = uyv$ and $|uy| = i$. This section shows the PMM produced by our algorithm is valid.

It is easy to see that:

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

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

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

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

Then we define sets and a mapping as follows: For a set of patterns $\Gamma = \{\pi_1, \ldots, \pi_k\}$, let $K = \pi_1 \cup \ldots \cup \pi_k$. Note that K is the set of all strings to be searched for. Define a mapping *state* from $PRE(K)$ into the set of non-negative integers by

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

The mapping *state* is *well-defined* because of Remark 1. Let $Q = \{state(u) | u \in PRE(K)\}$, which is the set of all states reachable from state 0. Let $PATH(s) = \{u \in PRE(K) | state(u) = s\}$, for all $s \in Q$. Note that $PRE(K) = \bigcup_{s \in Q} PATH(s)$ and for any $s, t \in Q$, $s \neq t$ implies $PATH(s) \cap PATH(t) = \emptyset$.

We are now ready to characterize the goto, failure, and output functions.

LEMMA 1. *Let $\pi = X_1 \ldots X_m$ be any pattern in Γ. Then, for any j with $1 \leq j \leq m$,*

there uniquely exists a nonempty subset I of Q such that $X_1 \ldots X_j = \bigcup_{s \in I} PATH(s)$ *and* $depth(s) = j$ *for any* $s \in I$, *where* $depth(s)$ *denotes the depth of* s.

PROOF. By induction on j.

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

LEMMA 2. *Let* $s \in Q$ *with* $s \neq 0$. *Let* u *be any string in* $PATH(s)$, *and let* v *be the longest string in* $(SUF(u) - \{u\}) \cap PRE(K)$. *Then,* $v \in PATH(f(s))$.

PROOF. By induction on the depth of s. ■

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

LEMMA 3. *For all* $s \in Q$ *with* $s \neq 0$,

$$\begin{aligned} out(s) &= \{\pi \in \Gamma | PATH(s) \subseteq \Sigma^*\pi\} \\ &= \{\pi \in \Gamma | PATH(s) \cap \Sigma^*\pi \neq 0\}. \end{aligned}$$

PROOF. It suffices to prove the following:

(1) $out(s) \subseteq \{\pi \in \Gamma | PATH(s) \subseteq \Sigma^*\pi\}$.

(2) $\{\pi \in \Gamma | PATH(s) \cap \pi^*\pi \neq \emptyset\} \subseteq out(s)$.

We shall first prove (1) by induction on the depth of s. It follows from the construction of *out* and Lemma 1 that

$$out(s) = \{\pi \in \Gamma | PATH(s) \subseteq \pi\} \cup out(f(s)),$$

so it clearly suffices to show that $out(f(s)) \subseteq \{\pi \in \Gamma | PATH(s) \subseteq \Sigma^*\pi\}$. By the induction hypothesis,

$$out(f(s)) \subseteq \{\pi \in \Gamma | PATH(f(s)) \subseteq \Sigma^*\pi\}.$$

Since $PATH(s) \subseteq \Sigma^* PATH(f(s))$ by Lemma 2, if $PATH(f(s)) \subseteq \Sigma^*\pi$ then $PATH(s) \subseteq \Sigma^*\pi$. Thus we complete the proof of (1).

We shall then prove (2) by induction on the depth of s. Since

$$out(s) = \{\pi \in \Gamma | PATH(s) \cap \pi \neq \emptyset\}\ out(f(s))$$

and by the induction hypothesis, it suffices to see that, for any $\pi \in \Gamma$, if $PATH(s) \cap \Sigma^*\pi \neq \emptyset$ then $PATH(s) \cap \pi \neq \emptyset$ or $PATH(f(s)) \cap \Sigma^*\pi \neq \emptyset$. Suppose that $PATH(s) \cap \Sigma^*\pi \neq \emptyset$. Let $u \in PATH(s) \cap \Sigma^*\pi$, and let $u = u'\alpha$ with $\alpha \in \pi$. If $u' = \varepsilon$, $u = \alpha \in PATH(s) \cap \pi$; Otherwise, choose v for u as in Lemma 2. Then, since $\alpha \in (SUF(u) - \{u\} \cap PRE(K)$, α must be a suffix of v, hence $v \in PATH(f(s)) \cap \Sigma^*\pi$. Thus we complete the proof of (2). ■

The following lemma characterizes the behavior of the PMM M in processing a text $T = a_1 a_2 \ldots a_n$.

LEMMA 4. *After jth operating cycle, M will be in state s iff PATH(s) contains the longest string in* $SUF(a_1a_2 \ldots a_j) \cap PRE(K)$.

PROOF. By induction on j. ∎

We now have the following theorem.

THEOREM 1. *The PMM M produced by our algorithm is valid.*

PROOF. By Lemmas 3 and 4. ∎

6. Theoretical Analysis

The text processing takes only linear time proportional to the text length. For actual run-time efficiency, the goto function should be stored in a two-dimensional array. Furthermore, if we convert a PMM into a deterministic finite state machine, and realize the state transition table by the table look-at technique [18, 6], then the machine runs on texts remarkably fast. However, when the patterns contain both pictures and characters, the number of states could grow exponentially in the worst case. In practice it is reasonable to store the states of depth less than d as direct access tables for some appropriate positive integer d. The other states can be stored as linear lists. For example, let us see the PMM of Example 2. The thick arrow from 3 to 4 labeled $A - \{\mathbf{a}, \mathbf{b}\}$ can be omitted if we have the gray arrow from 3 to 4 labeled by picture A together with the two arrows each labeled by "**a**" and "**b**". Thus, the linear lists will not be very long.

We then discuss the time complexity of the construction of PMM. Clearly, the first part of the construction takes only linear time proportional to the sum of the lengths of the patterns. The second part does so if the patterns consist only of symbols, or only of pictures. However, it is not so simple when the patterns contain both symbols 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 Methods 1 and 2 described in Section 3.

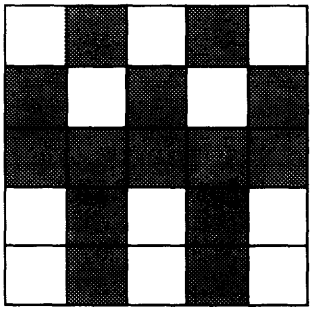
Suppose that $\Gamma = \{\pi_1, \pi_2, \ldots, \pi_k\}$ is the set of patterns to be searched for. Method 1 takes to construct the PMM linear time proportional to

$$\sum_{i=1}^{k} \left(\sum_{x \in \pi_i} |x| \right) = \sum_{i=1}^{k} \#(\pi_i) \cdot |\pi_i|,$$

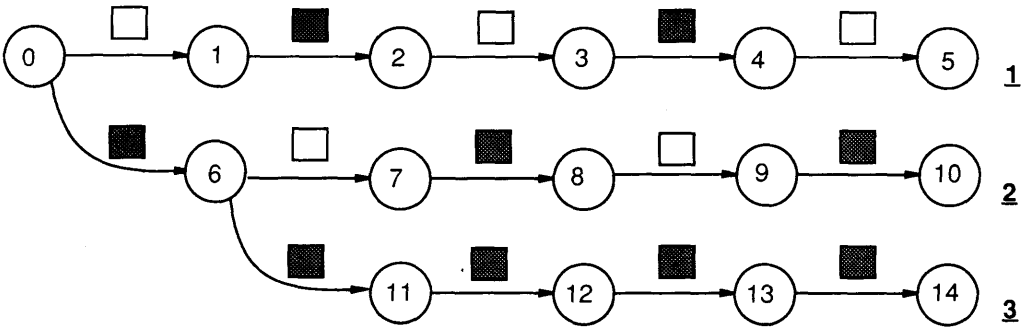
where $\#(\pi_i)$ denotes the number of strings in π_i. We then consider Method 2. Let c_i be, for each $i = 1, \ldots, p$, the number of distinct critical symbols in the picture A_i. Let $d(X) = 1$, if $X \in \Sigma$; $c_i + 1$, if $X = A_i$. Let $d(\pi) = d(X_1) \cdot d(X_2) \cdot \ldots \cdot d(X_m)$, for a pattern $\pi = X_1X_2 \ldots X_m$. Then, the cost of Method 2 is linearly proportional to

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

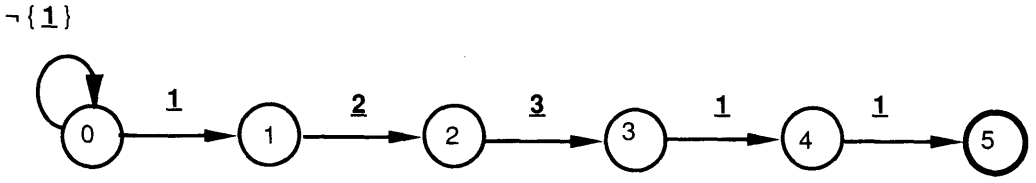
Unless a large number of different symbols appear in the patterns, $d(\pi_i)$ is much



(a) Pattern

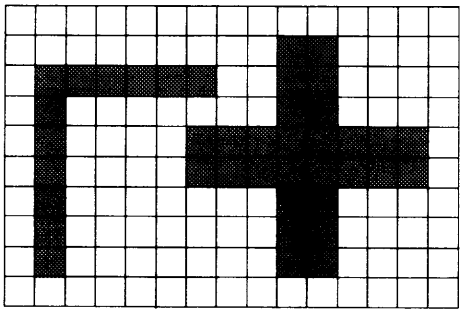


(b) The PMM for row-matching

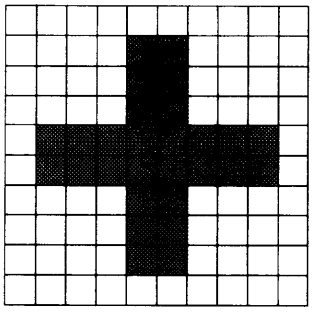


(c) The PMM for column-matching

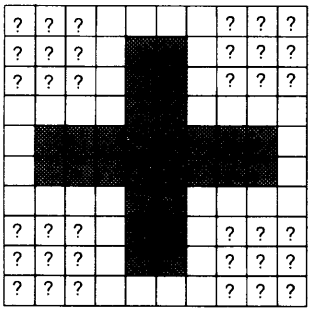
Fig. 7. Two-dimensional pattern matching by the Bird algorithm



(a) Text



(b) Rectangular pattern



(c) Pattern with pictures

Fig. 8. Searching for pattern with pictures

smaller than $\#(\pi_i)$. Moreover, if many symbols appear in the patterns, accordingly the number of occurrences of pictures in the patterns is small, thus $d(\pi_i)$ will be 1 frequently.

7. An Extension to the Two-dimensional Pattern Matching

In the two-dimensional pattern matching problem, both pattern and text are two dimensional arrays of characters. Bird [10] described an algorithm to solve this problem by using the AC method. Consider the pattern of Fig. 7(a). The Bird algorithm, regarding each row as a string pattern, builds a PMM as shown in Fig. 7(b). 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 PMM of Fig. 7(c) 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 $n_1 \times n_2$.

The Bird algorithm, however, has the following problem: Suppose that we wish to detect the cross within the text of Fig. 8(a). If we search for the rectangular pattern of Fig. 8(b) using the Bird algorithm, we cannot find it in the text.

In contrast, if we search for the pattern with pictures "?", as in Fig. 8(c), by our algorithm, then we can find the cross in the text.

8. Concluding Remarks

We have presented a fast matching algorithm for patterns with pictures. Since it is a natural extension of the AC algorithm, it has many possible applications.

Japanese texts consist of both 1-byte and 2-byte characters with shift codes. Shinohara and Arikawa [23] developed an algorithm to build a PMM for Japanese texts. The PMM runs on a Japanese text without loosing the efficiency, taking each byte as an input symbol. If we combine our algorithm with this, we can deal with not only 1-byte pictures but also some 2-byte pictures, such as *kanji*, *hiragana*, etc., by treating them as concatenations of two 1-byte pictures.

When editing texts we are often faced with the need to replace some words by others. Arikawa and Shiraishi [7] devised a multiple key replacement algorithm, which uses a generalized sequential machine produced in nearly the same manner as the AC machine. Our algorithm may improve the space efficiency of this method in some applications [25].

The pattern matching problem is conceptually simple, but is of great theoretical interest and practical importance. Various many works have been done on this problem, some of which should be mentioned in the remainder of this paper. We denote further by m, n the lengths of the pattern and the text, respectively.

The average case running time of the BM algorithm is said to be *sublinear*. Rivest [22] proved that no algorithm could solve the pattern matching problem in sublinear time even in the worst case. Yao [28] showed that the run time efficiency on the average of any matching algorithm could not be better than $O(n(\log m)/m)$, and this lower bound is achieved by Bailey and Dromey's algorithm [9].

The BM algorithm in the worst case takes $O(mn)$ time to locate all occurrences of

the pattern in the text. It can be modified so that the worst case running time is $O(n)$, independently of m [14, 5]. Guibas and Odlyzko [16] proved that the BM algorithm performs only at most $4n$ character comparisons when the pattern does not occur in the text.

The AC algorithm is an extension of the KMP algorithm to the multiple pattern problem. Similarly, fast algorithms for multiple patterns have been devised as extensions of the BM algorithm [12, 20, 27].

Approximate pattern matching has also been studied extensively. Three editing operations for a string are considered: *insertion*, *deletion*, and *substitution* of a letter. The *edit distance* between two strings are defined as the minimum total number of such editing steps needed for converting one of the strings to the other. The problem is, for a given integer $k \geq 0$, to find all substrings of the text that are at edit distance of at most k from the pattern. A straightforward solution based on the dynamic programming techniques takes $O(mn)$ time. Clever algorithms have been proposed [21, 15], which run in time $O(k^2n)$ or $O(kn)$, excluding the time for preprocessing of the pattern. However, no substantial progress have been achieved because the constant factor is large, and in practice, $k = \Theta(m)$. Ukkonen [26] proposed an interesting method: A deterministic finite automaton accepting the set Σ^*L is built, where L is the set of all strings at edit distance of at most k from the pattern. Although the construction of such an automaton is inefficient, this method is effective in some applications since the text scanning requires only $O(n)$ time.

References

[1] ABRAHAMSON, K.: *Generalized string matching*, SIAM J. Comput. **16** (1987) 1039–1051.

[2] AHO, A. V.: *Pattern matching in strings*, R. Book, editor, Formal lanuage theory: Perspectives and open problems, 325–347, Academic Press, New York, (1980).

[3] AHO, A. V. and CORASICK, M. J.: *Efficient string matching: An aid to bibliographic search*, Comm. ACM, **18** (1975), 333–340.

[4] AHO, A. V., HOPCROFT, J. E., and ULLMAN, J. D.: *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., (1974).

[5] APOSTOLICO, A. and GIAMCARLO, R.: *The Boyer-Moore-Galil string searching strategies revisited*, SIAM J. Comput., **15** (1986), 98–105.

[6] ARIKAWA, S. and SHINOHARA, T.: *A run-time efficient realization of Aho-Corasick pattern matching machine*, New Generation Comput., **2** (1984), 171–186.

[7] ARIKAWA, S. and SHIRAISHI, S.: *Pattern matching machines for replacing several character strings*, Bull. of Inform. Cybern., **21** (1984), 101–111.

[8] ARIKAWA, S. et al.: *The text database management system SIGMA: An improvement of the main engine*, J. Grabowski, editor, Proc. of the „Berliner Informatik Tage", Akademic der Wissenschaften der DDR, Berlin Sonderausgabe, (1989), 72–81.

[9] BAILEY, T. A. and DROMEY, R. G.: *Fast string searching by finding subkeys in subtext*, Inf. Process. Lett., **11** (1980), 130–133.

[10] BIRD, R. S.: Two dimensional pattern matching, Inf. Process. Lett., **6** (1977), 168–170.

[11] BOYER, R. S. and MOORE, J. S.: *A fast string searching algorithm*, Comm. ACM, **20** (1977), 762–772.

[12] COMMENTZ-WALTER, B.: *A string matching algorithm fast on the average*, H. A. Maurer, editor, Sixth International Colloquim on Automata, Languages and Programming, Lecture Notes in Computer Science, **71**, Springer-Verlag, (1979), 118–132.

[13] FISCHER, M. J. and PATERSON, M. S.: *String-matching and other products*, Complexity of computation, SIAM-AMS Proc., **7**, American Mathematical Society, Providence, R. I., (1974), 113–125.

[14] GALIL, Z.: *On improving the worst case running time of the Boyer-Moore string matching algorithm*, Comm. ACM, **22** (1979), 505–508.

[15] GALIL, Z. and PARK, K.: *An improved algorithm for approximate string matching*, SIAM J. Comput., **19** (1990), 989–999.

[16] GUIBAS, L. J. and ODLYZKO, A. M.: *A new proof of the linearity of the Boyer-Moore string searching algorithm*, SIAM J. Comput., **9** (1980), 672–682.

[17] HIGUCHI, T.: *On Thomas Mann File*, Research Reports in Computer Science, **4** (1987), 37–39, Computer Center, Kyushu University (in Japanese).

[18] KNUTH, D. E.: *The Art of Computer Programming*, **3:** *Sorting and Searching*, Addison-Wedley, (1973).

[19] KNUTH, D. E., MORRIS, J. H., and PRATT, V. R.: *Fast pattern matching in strings*, SIAM J. Comput., **6** (1977), 323–350.

[20] KOWALSKI, G. and MELTZER, A.: *New multi-term high speed text search algorithms*, The First International Conference on Computer and Applications, (1984), 514–521.

[21] LANDAU, G. M. and VISHKIN, U.: *Fast string matching with k differences*, J. Comput. System. Sci., (1988), 63–78.

[22] RIVEST, R. L.: *On the worst-case behavior of string searching algorithm*, SIAM J. Comput, **6** (1977), 669–674.

[23] SHINOHARA, T. and ARIKAWA, S.: *Pattern matching machines for Japanese texts*, Research Report No. 110, Res. Inst. of Fundamental Information Science, Kyushu University, (March 1986).

[24] TAKEDA, M.: *A fast matching algorithm for patterns with pictures*, Technical Report RIFIS-TR-CS-11, Res. Inst. of Fundamental Information Science, Kyushu University, (March 1989).

[25] TAKEDA, M.: *An efficient multiple string replacing algorithm using patterns with pictures*, Advances in Software Science and Technology, **2** (1990), 131–151.

[26] UKKONEN, E.: *Finding approximate patterns in strings*, J. Algorithms, **6** (1985), 132–137.

[27] URATANI, N. and TAKEDA, M.: *A fast string-searching algorithm for multiple patterns*, Information Processing & Management, (To appear).

[28] YAO, A. C.-C.: *The complexity of pattern matching for a random string*, SIAM J. Comput., **8** (1979), 368–387.

Received September 25, 1991
Revised August 30, 1992
Communicated by S. Arikawa