IPSJ 40th Anniversary Award Paper

Speeding Up String Pattern Matching by Text Compression: The Dawn of a New Era

Masayuki Takeda,[†] Yusuke Shibata,[†,☆] Tetsuya Matsumoto,[†]
Takuya Kida,[†] Ayumi Shinohara,[†] Shuichi Fukamachi,[††]
Takeshi Shinohara[††] and Setsuo Arikawa[†]

This paper describes our recent studies on string pattern matching in compressed texts mainly from practical viewpoints. The aim is to speed up the string pattern matching task, in comparison with an ordinary search over the original texts. We have successfully developed (1) an AC type algorithm for searching in Huffman encoded files, and (2) a KMP type algorithm and (3) a BM type algorithm for searching in files compressed by the so-called byte pair encoding (BPE). Each of the algorithms reduces the search time at nearly the same rate as the compression ratio. Surprisingly, the BM type algorithm runs over BPE compressed files about 1.2–3.0 times faster than the exact match routines of the software package `agrep`, which is known as the fastest pattern matching tool.

1. Introduction

String pattern matching is one of the most fundamental operations in string processing. The problem is to find all occurrences of a given pattern in a given text. It becomes more important to find a pattern in text files efficiently, as files become large. Recently, the *compressed pattern matching* problem attracts special concern, in which the aim is to find pattern occurrences in compressed text without decompression. It has been extensively studied for various compression methods by many researchers in the last decade from both theoretical and practical viewpoints. See **Table 1**.

One important goal of studies on this problem is to perform a faster search in compressed files in comparison with a decompression followed by an ordinary search (**Goal 1**). Although the prices of storage devices are coming down year by year, the data to be stored increase even more rapidly and we still tend to store them in a compressed form. Typical examples of such situations are mobile devices such as notebook computers and personal digital assistants (PDAs), where a user is often eager to insert any available information up to a possible limitation. Goal 1 is indeed attractive for this reason.

A more ambitious goal is to perform a faster search in compressed files in comparison with an ordinary search in the original files (**Goal 2**). In this case, the aim of compression is not only to reduce disk storage requirement but also to speed up string searching task. Let t_d, t_s, and t_c be the CPU times for a decompression, for searching in uncompressed files, and for searching in compressed files, respectively. Goal 1 aims for $t_d + t_s > t_c$ while Goal 2 for $t_s > t_c$. Thus, Goal 2 is more difficult to achieve than Goal 1.

The searching time is the sum of the file I/O time and the CPU time for pattern matching. In this paper, we focus on the reduction of the CPU time. Of course, text compression reduces the file I/O time at the same rate as the compression ratio, but it may increase the CPU time. When the data transfer is slow, e.g., network environments, the CPU time is negligible compared with the file I/O time. In this case, the elapsed time for searching in compressed files would be shorter than that for searching in the original files[☆☆]. On the other hand, this is not necessarily true when the data transfer is relatively fast, in such situations as a workstation with local disk storage or a notebook personal computer. If we achieve Goal 2, however, we can perform a faster search in compressed files in the elapsed time than an ordinary search in the original files even in such a situation.

† Department of Informatics, Kyushu University
☆ Presently with NTT Comware
†† Department of Artificial Intelligence, Kyushu Institute of Technology

☆☆ Even a naive method of decompression followed by a search can be faster than an ordinary search in the original files.

Table 1 Compressed pattern matching.

Compression method	Compressed pattern matching algorithms
Run-length	Eilam-Tzoreff and Vishkin[10]
Run-length (two dim.)	Amir, et al.[6]; Amir and Benson[2),3)]; Amir, et al.[5]
LZ77	Farach and Thorup[11]; Gąsieniec, et al.[14]
LZ78/LZW	Amir, et al.[4]; Kida, et al.[19]; Navarro and Raffinot[28]; Navarro and Tarhio[29]; Kärkkäinen, et al.[16]
Straight-line programs	Karpinski, et al.[17]; Miyazaki, et al.[25]; Hirao, et al.[15]
Huffman	Fukamachi, et al.[12]; Miyazaki, et al.[24]
Finite state encoding	Takeda[35]
Word based encoding	Moura, et al.[27]
Pattern substitution	Manber[22]; Shibata, et al.[32]
Collage systems	Kida, et al.[18]; Shibata, et al.[33]; Matsumoto, et al.[23]

Let n and N denote the compressed text length and the original text length, respectively. Theoretically, the best compression has $n = \sqrt{N}$ for the Lempel-Ziv-Welch (LZW) encoding[37], and $n = \log N$ for LZ77[39]. Thus an $O(n)$ time algorithm for searching directly in compressed text is considered to be better than a simple $O(N)$ time algorithm for searching in the original text☆. However, in practice n is linearly proportional to N for real text files. For this reason, an elaborate $O(n)$ time algorithm for searching in compressed text is often slower than a simple $O(N)$ time algorithm running on the original text. For example, as shown in Refs. 19), 28), searching in LZW compressed files is slower than searching in the original files, although it is faster than a decompression followed by an ordinary search.

In order to achieve the above two goals, especially Goal 2, we have to choose an appropriate compression method paying attention to the constant factors hidden behind the O-notation. Thus, we shall re-estimate the performances of the existing compression methods in the light of the new criterion: *efficiency of compressed pattern matching*. It is considered that a compression method which has received little attention against the traditional criteria (i.e., the compression ratio and the compression/decompression time) will be estimated good against the new criterion.

In this paper we describe our recent studies on this research theme. Text compression methods fall into two classes: character-wise compression and dictionary-based compression. For the former class, we focused on the Huffman encoding and the higher-order Huffman encoding, and developed run-time efficient algorithms for searching files compressed by these methods. The one dealing with Huffman encoded files runs faster than the Aho-Corasick (AC)[1] algorithm over the original files at the same rate as the compression ratio.

For the latter class, we introduced *collage system*, a unifying system which abstracts various dictionary-based compression methods. We showed two general pattern matching algorithms for text strings in terms of collage system, which simulate the Knuth-Morris-Pratt (KMP)[20] and the Boyer-Moore (BM)[8] algorithms, the most important algorithms for searching in uncompressed files. Within the framework of collage system, we re-estimated the existing dictionary-based methods from the viewpoint of compressed pattern matching, and concluded that the byte-pair encoding (BPE)[13] is most suitable for our purpose. We specialized the algorithms for dealing with BPE compressed files. Experimental results show that each of the obtained algorithms runs faster than the one being simulated. The searching time is reduced at nearly the same rate as the compression ratio. Surprisingly, the one simulating the move of the BM algorithm is about 1.2–3.0 times faster than the exact match routines of the software package `agrep`[38], which is known as the fastest pattern matching tool. Text compression thus accelerates string matching.

It should be mentioned that Manber[22] proposed a simple compression scheme that accelerates the string matching. The approach is rather straightforward: to encode a given pattern and to apply any search routine in order to find the encoded pattern within compressed files. The reductions of space and search time are not very good compared with BPE.

☆ The $O(n)$ time algorithm requires an extra $O(r)$ time in order to report all pattern occurrences, where r is the number of them, and r could be linear in N. But, we here ignore the $O(r)$ factor.

It should also be emphasized that Moura, et al.[27] proposed a compression scheme that uses a word-based Huffman encoding with a byte-oriented code. They presented an algorithm which runs twice faster than `agrep`. However, the compression method is not applicable to such texts as DNA sequences, which cannot be segmented into words. For the same reason, it cannot be used for natural language texts written in Japanese in which we have no blank symbols between words.

2. Preliminaries

Let Σ be a finite set of character symbols, called an *alphabet*. Denote by Σ^* the set of strings over Σ. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string $u = xyz$, respectively. The length of a string u is denoted by $|u|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. The ith symbol of a string u is denoted by $u[i]$ for $1 \leq i \leq |u|$, and the factor of a string u that begins at position i and ends at position j is denoted by $u[i:j]$ for $1 \leq i \leq j \leq |u|$. For a convenience, let $u[i:j] = \varepsilon$ for $i > j$. Let u be a string in Σ^*, and let i be a non-negative integer. Denote by ${}^{[i]}u$ (resp. $u^{[i]}$) the string obtained by removing the length i prefix (resp. suffix) from u.

3. For Character-wise Compression Methods

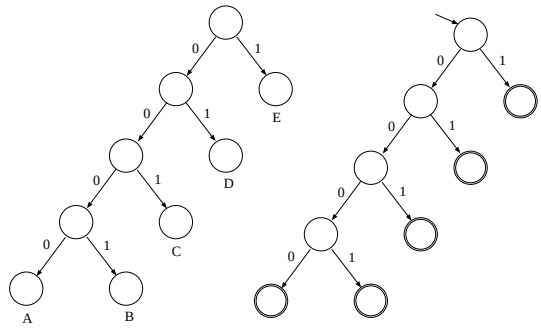
Text compression methods fall into two categories: the *character-wise compression* and the *dictionary-based compression*. In this section, we discuss the problem of compressed pattern matching in which text files are compressed using character-wise compression methods, such as the Huffman encoding.

3.1 Pattern Matching in Huffman Encoded Text

Pattern matching in Huffman encoded text seems not so difficult. A naive solution would be to encode a given pattern and to apply any favorite string search routine to find the encoded pattern within the compressed text. Searching with this approach, however, is very slow because of the following reasons: (1) An extra work for 'synchronization' is needed, i.e., the starting bit of each encoded symbol needs to be determined; (2) Bit-wise processing is rather slow. These problems can be solved as stated in the sequel.

3.1.1 Synchronization

As shown in Ref. 12), the first problem can be overcome by incorporating a deterministic finite-state automaton (DFA) that accepts the set of codewords into the pattern-matching machines (PMMs, in short) of the AC algorithm. Assume that $\Sigma = \{A, B, C, D, E\}$ and text strings over Σ are encoded according to the Huffman tree of **Fig. 1** (a). Figure 1 (b) is a DFA accepting the set of codewords, which is obtained directly from the Huffman tree. The smallest DFA accepting the same language can be built in only linear time with respect to the size of the Huffman tree, by using the minimization technique[31].



(a) Huffman tree (b) DFA

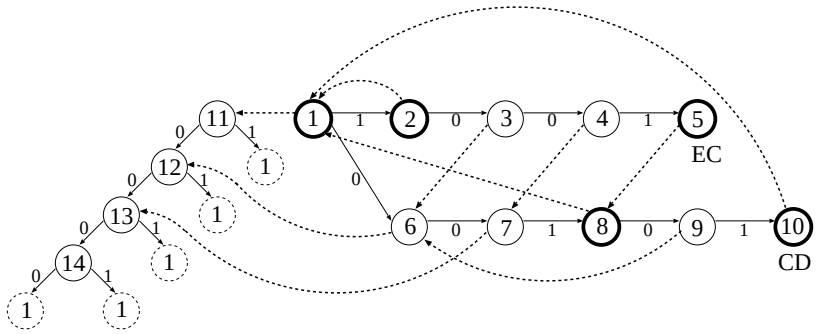
Fig. 1 Huffman tree and DFA accepting the set of codewords.

Suppose that we are given two patterns EC and CD. PMM for finding the encoded patterns 1001 and 00101 within a text compressed by the Huffman code is shown in **Fig. 2** (a). For a comparison, we also show PMM without DFA for synchronization in Fig. 2 (b), which may lead a false-detection of patterns.

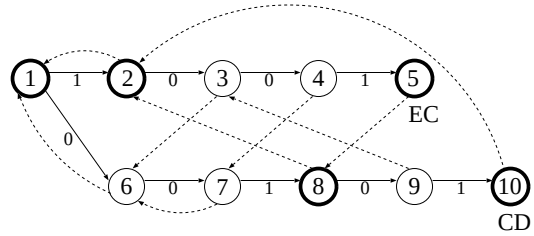
By using PMM with DFA for synchronization, we can process a Huffman encoded text bit-by-bit without any extra work for determining the starting bit of each encoded symbol. The size of PMM, namely, the number of states in it, is approximately $m \cdot R + \ell$, where m is the total length of patterns, R is the average codeword length, and ℓ is the size of DFA for synchronization, respectively.

3.1.2 Avoiding Bit-wise Processing

The second problem can be avoided by converting PMM into a new one so that it runs in byte-by-byte manner, as shown in Ref. 24). Although the machine size is larger than that of the usual PMM, the two-dimensional array implementation can be adopted. The new PMM runs on a Huffman encoded file faster than the usual PMM running on the original file, as will be shown in Section 6. The searching time is



(a) PMM with DFA for synchronization.



(b) PMM without DFA for synchronization.

Fig. 2 PMMs for searching in Huffman encoded text. The circles denote the states. The solid and the broken arrows represent the goto and the failure functions, respectively. The strings adjacent to the states mean the outputs from them. The thick line circles correspond to the starting bits of encoded symbols.

reduced at nearly the same rate as the compression ratio.

3.2 Searching in Files Compressed by Finite-state Encoder

Since the compression ratio is not very good in the Huffman encoding, we shall use the finite-state model in which the probability distribution is conditioned by the *current state*, often referred to as the *context*. The compression based on this model can be formalized as a finite-state encoder (FSE, in short). **Figure 3** shows an example of FSE with two states, where $\Sigma = \{A, B, C, D\}$ and the code alphabet is $\Delta = \{0, 1\}$. The code-trees for the two states 1 and 2 are shown in **Fig. 4**.

3.2.1 Compression Using FSE

An FSE is formally a 6-tuple $\langle Q, q_0, \Sigma, \Delta, \delta, \lambda \rangle$, where Q is a finite set of states; q_0 in Q is the initial state; Σ is a source alphabet; Δ is a code alphabet; $\delta : Q \times \Sigma \to Q$ is a state-transition function; and $\lambda : Q \times \Sigma \to \Delta^*$ is a coding function which satisfies the condition that, for any $q \in Q$ and any $a, b \in \Sigma$, if $\lambda(q, a)$ is a prefix of $\lambda(q, b)$, then $a = b$. Define for a state $q \in Q$ the function $\varphi_q : \Sigma \to \Delta^*$ by $\varphi_q(a) = \lambda(q, a)$ $(a \in \Sigma)$. Then, the above condition implies that φ_q is a one-to-one mapping and the set of

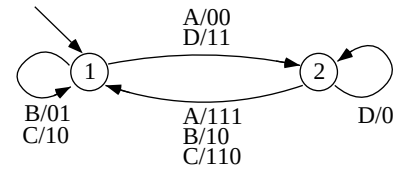
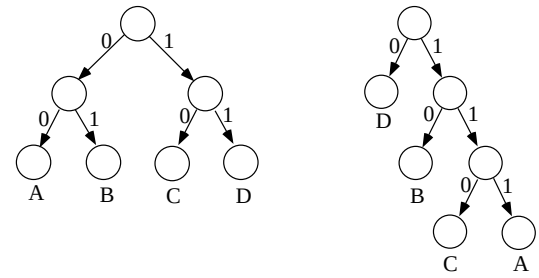


Fig. 3 Finite-state encoder with two states.



for state 1 for state 2

Fig. 4 Code-trees.

codewords $Codeword_q = \{\varphi_q(a) | a \in \Sigma\}$ has the *prefix property* for any state q.

Extend δ into the function from $Q \times \Sigma^*$ to Q by

$$\delta(q, \varepsilon) = q,$$
$$\delta(q, xa) = \delta(\delta(q, x), a),$$

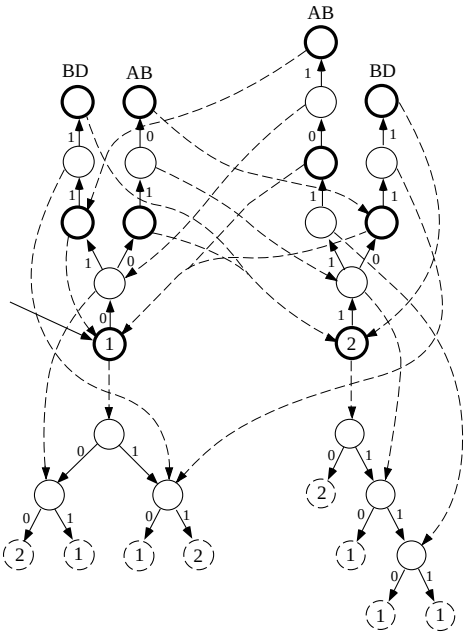


Fig. 5 PMM for searching in FSE compressed files.

and then extend λ into the function from $Q \times \Sigma^*$ to Δ^* by

$$\lambda(q, \varepsilon) = \varepsilon,$$
$$\lambda(q, xa) = \lambda(q, x) \cdot \lambda(\delta(q, x), a),$$

where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$. The encoding of a text $T \in \Sigma^*$ by an FSE is then defined to be the string $\lambda(q_0, T)$. For example, FSE of Fig. 3 takes as input the text $DBBDCAB$, makes state-transitions of $1 \to 2 \to 1 \to 1 \to 2 \to 1 \to 2 \to 1$, and emits as output the string $\lambda(1, DBBDCAB) = 11\,10\,01\,11\,110\,00\,10$.

3.2.2 PMMs for Searching in FSE Compressed Text

Suppose that we are given two patterns AB and BD. In state 1 we shall search for the strings $\lambda(1, AB) = 00\,10$ and $\lambda(1, BD) = 01\,11$, and in state 2 the strings $\lambda(2, AB) = 11101$ and $\lambda(2, BD) = 10\,11$. As shown in Ref. 35), PMM for searching in FSE compressed text can be constructed from the code-trees of Fig. 4 and from the tries representing the encodings of the patterns for two states. See **Fig. 5**.

The size of PMM is linearly proportional to the number of states of the underlying FSE. Therefore, we cannot use a large FSE. The problem is how to build a reasonably small FSE which has a relatively high compression ratio.

Miyazaki[26] proposed an approach to reduce the size of FSE, based on *alphabet indexing*. He showed a local search method which finds not the best but a 'good' alphabet indexing which yields a relatively high compression ratio.

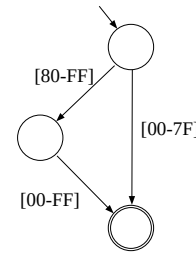


Fig. 6 DFA accepting Japanese texts (for EUC). Although the first and second 8-bits of each 16-bits codeword are in fact in [A1-FE], this DFA is enough for a correct Japanese text.

3.3 Searching in Japanese Text Files

Since an 8-bit code such as the ASCII code cannot represent many characters used in Japanese texts, a 16-bit code is used to represent such characters. Thus, a text file written in Japanese is a mixture of 8-bit codewords and 16-bit codewords. Automata-oriented approach to the string pattern matching in Japanese text seems to be unrealistic because the alphabet size is very large. However, we can build a PMM which runs on a Japanese text in a byte-by-byte manner.

In both of the Extended-Unix-Code (EUC) and the Shifted-JIS (SJIS), the first 8-bits of each of the 16-bit codewords is *not* identical to any of the 8-bit codewords. Thus the set of codewords satisfies the prefix property. Assuming the code alphabet is $\Delta = \{\mathtt{00}, \mathtt{01}, \ldots, \mathtt{FF}\}$ (i.e., the set of byte codes), DFA accepting the set of codewords is as shown in **Fig. 6**. By incorporating this DFA into PMMs, we can process Japanese text files in a byte-by-byte manner. The synchronization technique mentioned in Section 3.1.1 is thus applicable to any code satisfying the prefix property.

4. For Dictionary-based Methods

In this section we discuss the problem of compressed pattern matching for the dictionary-based methods. In a dictionary-based compression, a text string is described by a pair of a *dictionary* and a sequence of *tokens*, each of which represents a phrase defined in the dictionary. We introduced in Ref. 18) a unifying framework, named *collage system*, which abstracts various dictionary-based compression methods, such as the Lempel-Ziv family, the SEQUITUR[30], the Re-Pair[21], and the static dictionary methods. In Ref. 18) we presented a general compressed

pattern matching algorithm for texts described in terms of collage system. Consequently, any of the compression methods covered by the framework has a compressed pattern matching algorithm as an instance. The algorithm essentially simulates the move of the KMP algorithm. On the other hand, we also presented in Ref. 33) a BM type algorithm for collage systems.

4.1 Collage System

A *collage system* is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ defined as follows: $\mathcal{D}$ is a sequence of assignments $X_1 = expr_1;\ X_2 = expr_2;\ \cdots\ ; X_n = expr_n$, where each X_k is a variable (or a token) and $expr_k$ is any of the form:

a for $a \in \Sigma \cup \{\varepsilon\}$, (*primitive assignment*)
$X_i X_j$ for $i, j < k$, (*concatenation*)
$^{[j]}X_i$ for $i < k$ and an integer j, (*prefix truncation*)
$X_i^{[j]}$ for $i < k$ and an integer j, (*suffix truncation*)
$(X_i)^j$ for $i < k$ and an integer j. (j *times repetition*)

Each variable represents a string obtained by evaluating the expression as it implies. We identify a variable X_i with the string represented by X_i in the sequel. The *size* of $\mathcal{D}$ is the number n of assignments and denoted by $||\mathcal{D}||$. The *syntax tree* of a variable X in $\mathcal{D}$, denoted by $\mathcal{T}(X)$, is defined inductively as follows. The root node of $\mathcal{T}(X)$ is labeled by X and has:

no subtree, if $X = a \in \Sigma \cup \{\varepsilon\}$,
two subtrees $\mathcal{T}(Y)$ and $\mathcal{T}(Z)$, if $X = YZ$,
one subtree $\mathcal{T}(Y)$, if $X = Y^i, {}^{[i]}Y$, or $Y^{[i]}$.

Define the *height* of a variable X to be the height of the syntax tree $\mathcal{T}(X)$. The *height* of $\mathcal{D}$ is defined by $height(\mathcal{D}) = \max\{height(X) \mid X \text{ in } \mathcal{D}\}$. It expresses the maximum dependency of the variables in $\mathcal{D}$.

On the other hand, $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_k}$ is a sequence of variables defined in $\mathcal{D}$. We denote by $|\mathcal{S}|$ the number k of variables in $\mathcal{S}$. The collage system represents a string obtained by concatenating strings $X_{i_1}, X_{i_2}, \ldots, X_{i_k}$. Essentially, we can convert any collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ into the one where $\mathcal{S}$ consists of a single variable, by adding a series of concatenation operations into $\mathcal{D}$. The fact may suggest that $\mathcal{S}$ is unnecessary. However, by separating a dictionary $\mathcal{D}$ which only defines phrases, from $\mathcal{S}$ which intends for a sequence of phrases, we can capture a variety of compression methods naturally. Both $\mathcal{D}$ and $\mathcal{S}$ can be encoded in various ways. The compression ratios therefore depend upon the encoding sizes of $\mathcal{D}$ and $\mathcal{S}$ rather than upon $||\mathcal{D}||$ and $|\mathcal{S}|$.

A collage system is said to be *truncation-free* if $\mathcal{D}$ contain no truncation operation. A collage system is said to be *regular* if $\mathcal{D}$ contain neither repetition nor truncation operation.

4.2 KMP Type Algorithm for Collage Systems

Let us denote by $t.u$ the phrase represented by a token t. The problem is:

Given: a pattern $\pi = \pi[1:m]$ and a collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ with $\mathcal{S} = \mathcal{S}[1:n]$.
Find: all locations at which π occurs within the original text $\mathcal{S}[1].u \cdot \mathcal{S}[2].u \cdots \mathcal{S}[n].u$.

In Ref. 18) we presented a KMP type algorithm solving this problem. **Figure 7** gives an overview of the algorithm, which processes $\mathcal{S}$ token-by-token. The algorithm simulates the move of the KMP automaton running on the original text, by using two functions $Jump_{\mathrm{KMP}}$ and $Output_{\mathrm{KMP}}$, both take as input a state and a token. The former is used to substitute just one state transition for the consecutive state transitions of the KMP automaton caused by each of the phrases, and the latter is used to report all pattern occurrences found during the state transitions. Thus the definitions of the two functions are as follows.

$$Jump_{\mathrm{KMP}}(q, t) = \delta(q, t.u),$$
$$Output_{\mathrm{KMP}}(q, t) = \left\{ |v| \;\middle|\; \begin{array}{l} v \text{ is a non-empty prefix of } t.u \\ \text{such that } \delta(q, v) \text{ is the final state} \end{array} \right\},$$

where δ is the state transition function of the KMP automaton.

This idea is essentially based on the algorithm for searching in LZW compressed text due to Amir, et al.[4] which finds only the leftmost pattern occurrence. The extension to find all pattern occurrences was achieved by Kida, et al.[19], together with an extension to the multiple pattern problem.

Let $||\mathcal{D}||$ and $height(\mathcal{D})$ respectively denote the number of assignments in $\mathcal{D}$ and the maximum dependency in $\mathcal{D}$. Let r be the number of all occurrences of π in the text. The time and space complexities of the algorithm are as follows.

Lemma 1 (Kida, et al.[18]) The function $Jump_{\mathrm{KMP}}$ can be realized in $O(||\mathcal{D}|| \cdot height(\mathcal{D}) +$

```
Input:  Pattern π and collage system consisting of D and S = S[1 : n].
Output: All occurrences of π in the original text.
begin
/* Preprocessing */
    Compute the functions Jump_KMP and Output_KMP from the pattern π
      and the dictionary D;

/* Main routine */
    state := 0;   ℓ := 0;
    for i := 1 to n do begin
        for each d ∈ Output_KMP(state, S[i]) do
            Report a pattern occurrence that ends at position ℓ + d;
        state := Jump_KMP(state, S[i]);   ℓ := ℓ + |S[i].u|
    end
end.
```

Fig. 7 KMP Type algorithm for searching in a collage system.

m^2) time using $O(||\mathcal{D}|| + m^2)$ space, so that it responds in constant time. For a truncation-free collage system, the time complexity becomes $O(||\mathcal{D}|| + m^2)$.

Lemma 2 (Kida, et al.[18]) The procedure to enumerate the set $Output_{\mathrm{KMP}}(q, t)$ can be realized in $O(||\mathcal{D}|| \cdot height(\mathcal{D}) + m^2)$ time using $O(||\mathcal{D}|| + m^2)$ space, so that it responds in $O(height(t) + \ell)$ time, where ℓ is the size of the set. For a truncation-free collage system, it can be realized in $O(||\mathcal{D}|| + m^2)$ time and space, so that it runs in $O(\ell)$ time.

Theorem 1 (Kida, et al.[18]) The problem of compressed pattern matching can be solved in $O((||\mathcal{D}|| + n) \cdot height(\mathcal{D}) + m^2 + r)$ time using $O(||\mathcal{D}|| + m^2)$ space. For a truncation-free collage system, the time complexity becomes $O(||\mathcal{D}|| + n + m^2 + r)$.

The idea can also be applied to compressed pattern matching for other compression methods that are not contained in the collage system. For instance, we developed in Ref. 34) an algorithm, which is based on the similar idea, for searching in texts compressed using anti-dictionaries[9].

4.3 BM Type Algorithm for Collage Systems

We first briefly sketches the BM algorithm, and then show a BM type algorithm for searching in collage systems.

4.3.1 BM Algorithm on Uncompressed Text

The BM algorithm performs the character comparisons in the right-to-left direction, and slides the pattern to the right using the so-called shift function when a mismatch occurs.

```
Input:  Pattern π and text T = T[1 : N].
Output: All occurrences of π in T.
begin
/* Preprocessing */
    Compute the functions g and σ from the pattern π;

/* Main routine */
    T[0] := $; /* $ never occurs in pattern */
    i := m;
    while i ≤ N do begin
        state := 0;   ℓ := 0;
        while g(state, T[i − ℓ]) is defined do begin
            state := g(state, T[i − ℓ]);      ℓ := ℓ + 1
        end;
        if state = m then report a pattern occurrence;
        i := i + σ(state, T[i − ℓ])
    end
end.
```

Fig. 8 BM algorithm on uncompressed text.

The algorithm for searching in text $T[1 : N]$ is shown in **Fig. 8**. Note that the function g is the state transition function of the (partial) automaton that accepts the reversed pattern, in which state j represents the length j suffix of the pattern ($0 \le j \le m$).

Although there are many variations of the shift function, they are basically designed to shift the pattern to the right so as to align a text substring with its rightmost occurrence within the pattern. Let

$$rightmost\_occ(w) = \min \left\{ \ell > 0 \;\middle|\; \begin{array}{l} \pi[m - \ell - |w| + 1 : m - \ell] = w, \\ \text{or } \pi[1 : m - \ell] \text{ is a suffix of } w \end{array} \right\}.$$

The following definition, given by Uratani and Takeda[36] (for multiple pattern case), is the one which utilizes all information gathered in one execution of the inner-while-loop in the algo-

Input: Pattern π and collage system consisting of $\mathcal{D}$ and $\mathcal{S} = \mathcal{S}[1:n]$.
Output: All occurrences of π in the original text.
begin
/* Preprocessing */
Compute the functions $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$ and Occ from the pattern π and the dictionary $\mathcal{D}$;

/* Main routine */
focus := an appropriate value;
while *focus* $\leq n$ **do begin**
Step 1: *Report all pattern occurrences that are contained in the phrase $\mathcal{S}[focus].u$ by using Occ;*
Step 2: *Find all pattern occurrences that end within the phrase $\mathcal{S}[focus].u$ by using $Jump_{\mathrm{BM}}$ and $Output_{\mathrm{BM}}$;*
Step 3: *Compute a possible shift Δ based on information gathered in Step 2;*
focus := *focus* + Δ
end
end.

Fig. 9 Overview of BM type compressed pattern matching algorithm.

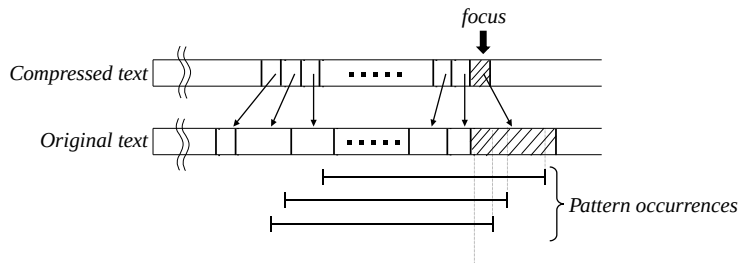


Fig. 10 Pattern occurrences.

rithm of Fig. 8.

$\sigma(j, a) = rightmost\_occ(a \cdot \pi[m-j+1:m])$. The two-dimensional array realization of this function requires $O(|\Sigma| \cdot m)$ memory, but it becomes realistic due to recent progress in computer technology. Moreover, the array can be shared with the goto function g. This saves not only memory requirement but also the number of table references.

4.3.2 Algorithm for Collage System

Now, we show a BM type algorithm for searching in collage systems. **Figure 9** gives an overview of our algorithm. For each iteration of the while-loop, we report in Step 1 all the pattern occurrences that are contained in the phrase represented by the token we focus on, determine in Step 2 the pattern occurrences that end within the phrase, and then shift our focus to the right by Δ obtained in Step 3. Let us call the token we focus on the *focused token*, and the phrase it represents the *focused phrase*. For Step 1, we shall compute during the preprocessing, for every token t, the set $Occ(t)$ of all pattern occurrences contained in the phrase $t.u$. The time and space complexities of this computation are as follows.

Lemma 3 (Kida, et al.[18]) We can build in $O(height(\mathcal{D}) \cdot ||\mathcal{D}|| + m^2)$ time using $O(||\mathcal{D}|| + m^2)$ space a data structure by which the enumeration of the set $Occ(t)$ is performed in $O(height(t) + \ell)$ time, where $\ell = |Occ(t)|$. For a truncation-free collage system, it can be built in $O(||\mathcal{D}|| + m^2)$ time and space, and the enumeration requires only $O(\ell)$ time.

In the following we discuss how to realize Step 2 and Step 3.

Figure 10 illustrates pattern occurrences that end within the focused phrase. A candidate for pattern occurrence is a non-empty prefix of the focused phrase that is also a proper suffix of the pattern. There may be more than one candidate to be checked. One naive method is to check all of them independently, but here we take another approach. We shall

```
procedure Find_pattern_occurrences(focus : integer);
begin
    if Jump_BM(0, S[focus]) is undefined then return;
    state := Jump_BM(0, S[focus]);    d := state;    ℓ := 1;
    repeat
        while Jump_BM(state, S[focus − ℓ]) is defined do begin
            state := Jump_BM(state, S[focus − ℓ]);    ℓ := ℓ + 1
        end;
        if Output_BM(state, S[focus − ℓ]) = true then report a pattern occurrence;
        d := d − (state − f(state));    state := f(state)
    until d ≤ 0
end;
```

Fig. 11 Finding pattern occurrences in Step 2.

start with the longest one. For the case of uncompressed text, we can do it by using the partial automaton for the reversed pattern stated in Section 4.3.1. When a mismatch occurs, we change the state by using the failure function and try to proceed into the left direction. The process is repeated until the pattern does not have an overlap with the focused phrase. In order to perform such processing over compressed text, we use the two functions $Jump_{\mathrm{BM}}$ and $Output_{\mathrm{BM}}$ defined in the sequel.

Let $lpps(w)$ denote the longest prefix of a string w that is also a proper suffix of the pattern π. Extend the function g into the domain $\{0,\ldots,m\} \times \Sigma^*$ by $g(j, aw) = g(g(j, w), a)$, if $g(j, w)$ is defined and otherwise, $g(j, aw)$ is undefined, where $w \in \Sigma^*$ and $a \in \Sigma$. Let $f(j)$ be the largest integer k $(k < j)$ such that the length k suffix of the pattern is a prefix of the length j suffix of the pattern. Note that f is the same as the failure function of the KMP automaton. Define the functions $Jump_{\mathrm{BM}}$ and $Output_{\mathrm{BM}}$ by

$$Jump_{\mathrm{BM}}(j,t) = \begin{cases} g(j,t.u), & \text{if } j \neq 0; \\ |lpps(t.u)|, & \text{if } j=0 \text{ and } lpps(t.u) \neq \varepsilon; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$Output_{\mathrm{BM}}(j,t) = \begin{cases} true, & \text{if } g(j,w) = m \text{ and } w \text{ is} \\ & \quad \text{a proper suffix of } t.u; \\ false, & \text{otherwise.} \end{cases}$$

The procedure for Step 2 is shown in **Fig. 11**.

We now discuss how to compute the possible shift Δ of the focus. Let

$$Shift(j,t) = rightmost\_occ(t.u \cdot \pi[m-j+1:m]).$$

Assume that starting at the token $\mathcal{S}[focus]$, we encounter a mismatch against a token t in state j. Find the minimum integer $k > 0$ such that

$$Shift(0, \mathcal{S}[focus]) \leq \sum_{i=1}^{k} \Big|\mathcal{S}[focus+i].u\Big|, \quad \text{or} \tag{1}$$

$$Shift(j,t) \leq \sum_{i=0}^{k} \Big|\mathcal{S}[focus+i].u\Big| - \Big|lpps(\mathcal{S}[focus].u)\Big|. \tag{2}$$

Note that the shift due to Eq. (1) is possible independently of the result of the procedure of Fig. 11. When returning at the first if-then statement of the procedure in Fig. 11, we can shift the focus by the amount due to Eq. (1). Otherwise, we shift the focus by the amount due to both Eq. (1) and Eq. (2) for $j = state$ and $t = \mathcal{S}[focus - \ell]$ just after the execution of the while-loop at the first iteration of the repeat-until loop.

Lemma 4 (Shibata, et al.[33]) The functions $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$, and *Shift* can be built in $O(height(\mathcal{D}) \cdot ||\mathcal{D}|| + m^2)$ time and $O(||\mathcal{D}||+m^2)$ space, so that they answer in $O(1)$ time. The factor $height(\mathcal{D})$ can be dropped if the collage system is truncation-free.

Theorem 2 (Shibata, et al.[33]) The algorithm of Fig. 9 runs in $O(height(\mathcal{D}) \cdot (||\mathcal{D}|| + n) + n \cdot m + m^2 + r)$ time, using $O(||\mathcal{D}|| + m^2)$ space. For a truncation-free collage system, the time complexity becomes $O(||\mathcal{D}|| + n \cdot m + m^2 + r)$.

4.4 Practical Aspects

Theorem 1 suggests that a compression method described as a collage system with no truncation might be suitable for the speed-up of pattern matching. In fact the collage systems

for LZ77 have truncation and LZ77 is not suitable as shown in Ref. 28). On the other hand, the collage systems for LZW are truncation-free. However, searching in LZW compressed files are rather slow in comparison with searching in the original files, as shown in Refs. 19), 28). We have two reasons. One is that in LZW the dictionary $\mathcal{D}$ is not encoded explicitly: it will be incrementally re-built from $\mathcal{S}$. The preprocessing of $\mathcal{D}$ is therefore merged into the main routine (see Fig. 7 again). The other reason is as follows. Although $Jump_{\mathrm{KMP}}$ can be realized using only $O(||\mathcal{D}|| + m)$ space so that it answers in constant time, the constant factor is relatively large. The two-dimensional array implementation of $Jump_{\mathrm{KMP}}$ would improve this, but it requires $O(||\mathcal{D}|| \cdot m)$ space, which is unrealistic because $||\mathcal{D}||$ is linear with respect to n in the case of LZW.

From the above observations the desirable properties for compressed pattern matching can be summarized as follows.

- The dictionary $\mathcal{D}$ contains no truncation.
- The dictionary $\mathcal{D}$ is encoded separately from the sequence $\mathcal{S}$.
- The size of $\mathcal{D}$ is small enough.
- The tokens of $\mathcal{S}$ are encoded using a fixed length code.

The compression scheme called the byte pair encoding (BPE)[13] is the one which satisfies all of the properties. It is regarded as a simplified version of the compression method called Re-Pair[21]. The basic operation of the compression is to substitute a single character which did not appear in the text for a pair of consecutive two characters which frequently appears in the text. This operation will be repeated until either all characters are used up or no pair of consecutive two characters appears frequently. Thus, $||\mathcal{D}|| \leq 256$. In the next section, we show compressed pattern matching algorithms for BPE compressed files, which are very fast in practice.

5. Searching in BPE Compressed Files

We specialized the KMP type and the BM type algorithms, presented in the last section, for dealing with only BPE compressed files.

5.1 KMP (AC) Type Algorithm

We can take the two-dimensional array realization of $Jump_{\mathrm{KMP}}$, which is realistic since the array size is only $256 \cdot (m+1)$.

Lemma 5 (Shibata, et al.[32]) For a regular collage system, the two-dimensional tables storing $Jump_{\mathrm{KMP}}$ and $Output_{\mathrm{KMP}}$ can be built in $O(||\mathcal{D}|| \cdot m)$ time and space.

The searching time is reduced at almost the same rate as the compression ratio, as will be seen in Section 6. BPE compresses Japanese texts nearly the same ratio as English texts, and therefore the method can be applied to searching in BPE compressed Japanese texts. This is a big advantage of this method. Moreover, the algorithm can be extended to the multipattern searching problem so that it simulates the AC algorithm.

5.2 BM Type Algorithm

Lemma 6 (Shibata, et al.[33]) For a regular collage system, the tables storing $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$, and *Shift* can be built in $O(||\mathcal{D}|| \cdot m)$ time and space.

By using the BM type algorithm presented in the previous section, we cannot shift the pattern without knowing the total length of phrases corresponding to skipped tokens. This slows down the algorithm in practice. To do without such information, we assume that the skipped phrases are all of length C, the maximum phrase length in $\mathcal{D}$, and divide the shift value by C. The value of C is crucial in this approach.

We estimated the change of compression ratios depending on C. The text files we used are:

Medline. A clinically-oriented subset of Medline, consisting of 348,566 references. The file size is 60.3 Mbyte and the entropy[☆] is 4.9647.

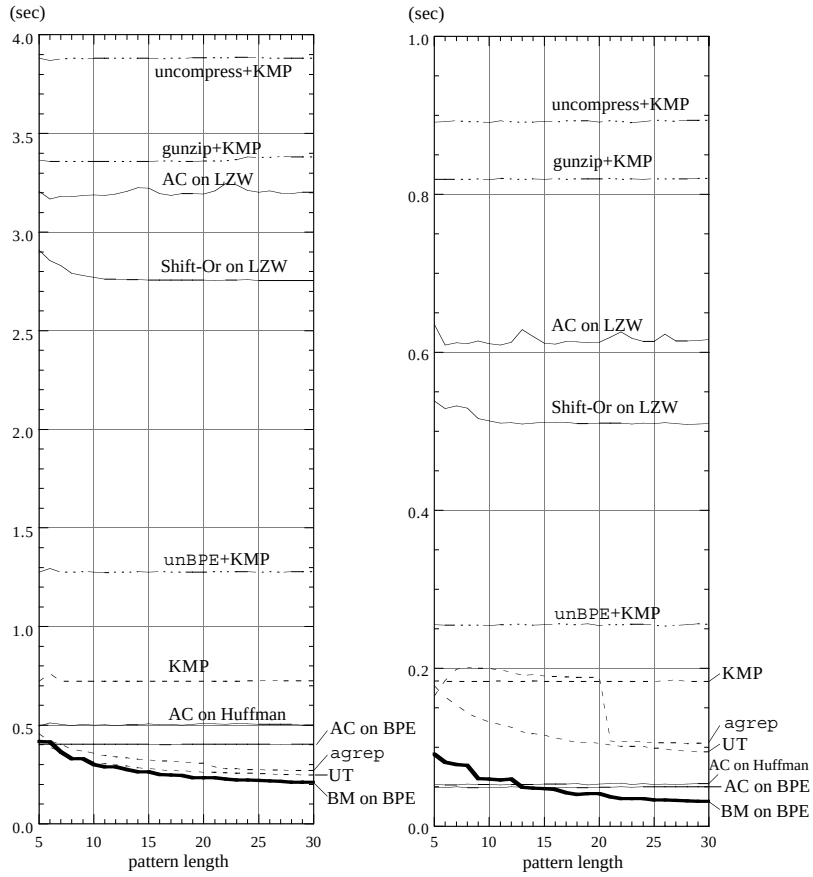
Genbank. The file consisting only of accession numbers and nucleotide sequences taken from a data set in Genbank. The file size is 17.1 Mbyte and the entropy is 2.6018.

Table 2 shows the compression ratios of these texts for BPE, together with those for the Huffman encoding, `gzip`, and `compress`, where the last two are well-known compression tools based on LZ77 and LZW, respectively. Remark that the change of compression ratios depending on C is non-monotonic. The reason for this is that the BPE compression routine we used builds a dictionary $\mathcal{D}$ in a greedy manner only from the first block of a text file. It is observed that we can restrict C with no great sacrifice of compression ratio. Thus we decided to use the BPE compressed file of Medline for $C = 3$, and

[☆] By entropy is meant the order-0 entropy, namely the classical Shannon entropy.

Table 2 Compression ratios (%).

	Huffman	BPE							`compress`	`gzip`
		$C=3$	$C=4$	$C=5$	$C=6$	$C=7$	$C=8$	unlimit.		
Medline	62.41	59.44	58.46	58.44	58.53	58.47	58.58	59.07	42.34	33.35
Genbank	33.37	36.93	32.84	32.63	32.63	32.34	32.28	32.50	26.80	23.15



(a) Medline (b) Genbank

Fig. 12 Running times (in CPU time).

that of Genbank for $C = 4$ in our experiment in the next section.

For the BPE compression, we observed that putting a restriction on C makes no great sacrifice of compression ratio even for $C = 3, 4$.

6. Experimental results

We estimated the performances of the following programs:

(A) *Decompression followed by ordinary search.* We tested this approach with the KMP algorithm for the compression methods: `gzip`, `compress`, and BPE. We did not combine the decompression programs and the KMP search program using the Unix 'pipe' because it is slow. Instead, we embedded the KMP routine in the decompression programs, so that the KMP automaton processes the decoded characters 'on the fly'. The programs are abbreviated as `gunzip`+KMP, `uncompress`+KMP, and `unBPE`+KMP, respectively.

(B) *Ordinary search in original text.* KMP, UT (the Uratani-Takeda variant[36] of BM), and `agrep`.

(C) *Compressed pattern matching.* AC on LZW[19], Shift-Or on LZW[19], AC on Huffman[24], AC on BPE[32], and BM on BPE[33].

The automata in KMP, UT, AC on Huffman, AC on BPE, and BM on BPE were realized as two-dimensional arrays of size $\ell \times 256$, where ℓ is the number of states. The texts used are Medline and Genbank mentioned in Section 5, and the patterns searched for are text substrings randomly gathered from them. Our experiment was carried out on an AlphaStation XP1000 with an Alpha21264 processor at 667 MHz running Tru64 UNIX operating system V4.0F. **Figure 12** shows the running times (CPU time). We excluded the preprocessing times since they are negligible compared with the running times. We observed the following facts.

- The differences between the running times of KMP and the three programs of (A) correspond to the decompression times. Decompression tasks for LZ77 and LZW are thus time-consuming compared with pattern matching task. Even when we use the BM algorithm instead of KMP, the approach (A) is still slow for LZ77 and LZW.
- BM on BPE is faster than all the others. Especially, it runs about 1.2 times faster than `agrep` for Medline, and about 3 times faster for Genbank.

7. Concluding Remarks

The aim of compression was traditionally to save space requirement of data files. Instead of saving space, we must bear an extra time for expanding files when processing them. For this reason, we did not want to compress data files in the case that the available space was sufficient. However, we now have a new reason to compress data files, in which the compression is regarded as a means of speeding up the exact string matching task, which we call Goal 2. The results presented in this paper thus have raised the curtain on a new era in the history of string pattern matching and data compression.

The work by Manber[22] (1994) is recognized as the first attempt to Goal 2. However, an attempt was made by our research group in Ref. 12) early in 1992. In this work, we incorporated the Huffman tree into the pattern matching machine for syncronization, as described in Section 3. This idea was obtained as a generalization of the one used for bytewise-processing of Japanese text files, which are mixtures of one-byte and two-byte codes. A naive method of processing such a text file would be to convert the text into a sequence of 16-bit integers, each of which represents one character, and then process it by an ordinary search routine. Notice that the naive method is similar to the decompression-then-search method, and on the other hand, the bytewise-processing seems like a compressed pattern matching. The requirement to efficiently process Japanese text files forced us to distinguish a text from its representation. Most interestingly, we have proposed in 1984 in Ref. 7) an efficient realization of the pattern matching machine which processes a text file 4-bit by 4-bit, in order to reduce the size of the state-transition table into 1/8. Thus, before the 1990's we had almost finished preparing for studies on the compressed pattern matching.

The string pattern matching is simple but the basis of other string processings. To speed up more complicated string processings, such as the approximate string matching or the regular expression matching, by using text compression will be our future work.

References

1) Aho, A.V. and Corasick, M.: Efficient string matching: An aid to bibliographic search, *Comm. ACM*, Vol.18, No.6, pp.333–340 (1975).
2) Amir, A. and Benson, G.: Efficient two-dimensional compressed matching, *Proc. Data Compression Conference*, p.279 (1992).
3) Amir, A. and Benson, G.: Two-dimensional periodicity and its application, *Proc. 3rd Ann. ACM-SIAM Symp. on Discrete Algorithms*, pp.440–452 (1992).
4) Amir, A., Benson, G. and Farach, M.: Let sleeping files lie: Pattern matching in Z-compressed files, *J. Computer and System Sciences*, Vol.52, pp.299–307 (1996).
5) Amir, A., Benson, G. and Farach, M.: Optimal two-dimensional compressed matching, *J. Algorithms*, Vol.24, No.2, pp.354–379 (1997).
6) Amir, A., Landau, G.M. and Vishkin, U.: Efficient pattern matching with scaling, *J. Algorithms*, Vol.13, No.1, pp.2–32 (1992).
7) Arikawa, S. and Shinohara, T.: A run-time efficient realization of Aho-Corasick pattern matching machines, *New Generation Computing*, Vol.2, No.2, pp.171–186 (1984).
8) Boyer, R.S. and Moore, J.S.: A fast string searching algorithm, *Comm. ACM*, Vol.20, No.10, pp.62–72 (1977).
9) Crochemore, M., Mignosi, F., Restivo, A. and Salemi, S.: Text compression using antidictionaries, *Proc. 26th Internationial Colloquim on Automata, Languages and Programming*, pp.261–270, Springer-Verlag (1999).
10) Eilam-Tzoreff, T. and Vishkin, U.: Match-

ing patterns in strings subject to multi-linear transformations, *Theoretical Computer Science*, Vol.60, No.3, pp.231–254 (1988).
11) Farach, M. and Thorup, M.: String-matching in Lempel-Ziv compressed strings, *Algorithmica*, Vol.20, No.4, pp.388–404 (1998). (Previous version in: *STOC'95*).
12) Fukamachi, S., Shinohara, T. and Takeda, M.: String pattern matching for compressed data using variable length codes (in Japanese), *Proc. Symposium on Informatics 1992*, pp.95–103 (1992).
13) Gage, P.: A new algorithm for data compression. *The C Users Journal*, Vol.12, No.2 (1994).
14) Gąsieniec, L., Karpinski, M., Plandowski, W. and Rytter, W.: Efficient algorithms for Lempel-Ziv encoding, *Proc. 4th Scandinavian Workshop on Algorithm Theory*, pp.392–403, Springer-Verlag (1996).
15) Hirao, M., Shinohara, A., Takeda, M. and Arikawa, S.: Fully compressed pattern matching algorithm for balanced straight-line programs, *Proc. 7th International Symp. on String Processing and Information Retrieval*, pp.132–138, IEEE Computer Society (2000).
16) Kärkkäinen, J., Navarro, G. and Ukkonen, E.: Approximate string matching over Ziv-Lempel compressed text, *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*, pp.195–209, Springer-Verlag (2000).
17) Karpinski, M., Rytter, W. and Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions, *Nordic Journal of Computing*, Vol.4, pp.172–186 (1997).
18) Kida, T., Shibata, Y., Takeda, M., Shinohara, A. and Arikawa, S.: A unifying framework for compressed pattern matching, *Proc. 6th International Symp. on String Processing and Information Retrieval*, pp.89–96, IEEE Computer Society (1999).
19) Kida, T., Takeda, M., Shinohara, A., Miyazaki, M. and Arikawa, S.: Multiple pattern matching in LZW compressed text, *J. Discrete Algorithms* (to appear). (Previous versions in: *DCC'98* and *CPM'99*).
20) Knuth, D.E., Morris, J.H. and Pratt, V.R.: Fast pattern matching in strings, *SIAM J. Comput*, Vol.6, No.2, pp.323–350 (1977).
21) Larsson, N.J. and Moffat, A.: Offline dictionary-based compression, *Proc. Data Compression Conference '99*, pp.296–305, IEEE Computer Society (1999).
22) Manber, U.: A text compression scheme that allows fast searching directly in the compressed file, *ACM Trans. Information Systems*, Vol.15, No.2, pp.124–136 (1997). (Previous version in: *CPM'94*).
23) Matsumoto, T., Kida, T., Takeda, M., Shinohara, A. and Arikawa, S.: Bit-parallel approach to approximate string matching in compressed texts, *Proc. 7th International Symp. on String Processing and Information Retrieval*, pp.221–228, IEEE Computer Society (2000).
24) Miyazaki, M., Fukamachi, S., Takeda, M. and Shinohara, T.: Speeding up the pattern matching machine for compressed texts (in Japanese), *Trans. IPS Japan*, Vol.39, No.9, pp.2638–2648 (1998).
25) Miyazaki, M., Shinohara, A. and Takeda, M.: An improved pattern matching algorithm for strings in terms of straight-line programs. *J. Discrete Algorithms* (to appear). (Previous version in: *CPM'97*).
26) Miyazaki, T.: Studies on speed-up of string pattern matching by text compression using statistical model (in Japanese), Master's Thesis, Kyushu Institute of Technology (1996).
27) Moura, E., Navarro, G., Ziviani, N. and Baeza-Yates, R.: Fast and flexible word searching on compressed text, *ACM Trans. Information Systems* (2000). (Previous Versions in: *SIGIR'98* and *SPIRE'98*).
28) Navarro, G. and Raffinot, M.: A general practical approach to pattern matching over Ziv-Lempel compressed text, *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pp.14–36, Springer-Verlag (1999).
29) Navarro, G. and Tarhio, J.: Boyer-Moore string matching over Ziv-Lempel compressed text, *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*, pp.166–180, Springer-Verlag (2000).
30) Nevill-Manning, C.G., Witten, I.H. and Maulsby, D.L.: Compression by induction of hierarchical grammars, *DCC94*, pp.244–253, IEEE Press (1994).
31) Revuz, D.: Minimisation of acyclic deterministic automata in linear time, *Theoretical Computer Science*, Vol.92, No.1, pp.181–189 (1992).
32) Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T. and Arikawa, S.: Speeding up pattern matching by text compression, *Proc. 4th Italian Conference on Algorithms and Complexity*, pp.306–315, Springer-Verlag (2000).
33) Shibata, Y., Matsumoto, T., Takeda, M., Shinohara, A. and Arikawa, S.: A Boyer-Moore type algorithm for compressed pattern matching, *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*, pp.181–194, Springer-Verlag (2000).
34) Shibata, Y., Takeda, M., Shinohara, A. and Arikawa, S.: Pattern matching in text compressed by using antidictionaries, *J. Discrete*

Algorithms (to appear). (Previous version in: *CPM'99*).
35) Takeda, M.: Pattern matching machine for text compressed using finite state model, Technical Report DOI-TR-CS-142, Department of Informatics, Kyushu University (Oct. 1997).
36) Uratani, N. and Takeda, M.: A fast string-searching algorithm for multiple patterns. *Information Processing & Management*, Vol.29, No.6, pp.775–791 (1993).
37) Welch, T.A.: A technique for high performance data compression, *IEEE Comput.*, Vol.17, pp.8–19 (1984).
38) Wu, S. and Manber, U.: Agrep – a fast approximate pattern-matching tool, *Usenix Winter 1992 Technical Conference*, pp.153–162 (1992).
39) Ziv, J. and Lempel, A.: A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory*, Vol.23, No.3, pp.337–349 (1977).

(Received June 30, 2000)
(Accepted September 27, 2000)

Masayuki Takeda was born in 1964. He is an Associate Professor in the Department of Informatics at Kyushu University. He received his B.S. in 1987 in Mathematics, his M.S. in 1989 in Information Systems, and his Dr.Eng. degree in 1996 all from Kyushu University. His present research interests include pattern matching algorithms, text compression, discovery science, and information retrieval.

Yusuke Shibata was born in 1975. He receieved his B.E. and M.E. from Kyushu Institute of Technology in 1998 and from Kyushu University in 2000, respectively. His research interests include pattern matching algorithms and text compression. Presently, he works at NTT Comware.

Tetsuya Matsumoto was born in 1977. He is a student of the master course in the Department of Informatics at Kyushu University. He receieved his B.S. in Physics from Kyushu University in (1999). His research interests include pattern matching algorithms and text compression.

Takuya Kida was born in 1974. He is a student of the doctor course in the Department of Informatics at Kyushu University. He obtained his B.S. in 1997 in Physics, his M.A. in 1999 from Kyushu University. His research interests include pattern matching algorithms and text compression.

Ayumi Shinohara was born in 1965. He is an Associate Professor in the Department of Informatics at Kyushu University. He obtained his B.S. in 1988 in Mathematics, his M.S. in 1990 in Information Systems, and his Dr.Sci. degree in 1994 all from Kyushu University. His current research interests include discovery science, bioinformatics, and pattern matching algorithms.

Shuichi Fukamachi was born in 1967. He is a researcher in the Department of Artificial Intelligence at Kyushu Institute of Technology. He received his B.E. and M.E. from Kyushu Institute of Technology in 1991 and 1993, respectively. His research interests include pattern matching algorithms and information retrieval.

Takeshi Shinohara was born in 1955. He is a Professor in the Department of Artificial Intelligence at Kyushu Institute of Technology. He obtained his B.S. in Mathematics from Kyoto University in 1980, and his Dr.Sci. degree from Kyushu University in 1986. His research interests are in computational/algorithmic learning theory, information retrieval, and approximate retrieval of multimedia data.

Setsuo Arikawa was born in 1941. He is a Professor in the Department of Informatics at Kyushu University and the Director of University Library at Kyushu University. He received his B.S. in 1964, his M.S. in 1966 and his Dr.Sci. degree in 1969 all in Mathematics from Kyushu University. His research interests include discovery science, algorithmic learning theory, logic and inference/reasoning in AI, pattern matching algorithms and library science.