Multiple Pattern Matching in LZW Compressed Text

TAKUYA KIDA[1] , MASAYUKI TAKEDA, AYUMI SHINOHARA, MASAMICHI MIYAZAKI[2], SETSUO ARIKAWA, *Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan,* {kida, takeda, ayumi, masamich, arikawa}@i.kyushu-u.ac.jp

ABSTRACT: In this paper we address the problem of searching in LZW compressed texts directly. We present an algorithm that finds all occurrences of multiple patterns in comparison with the algorithm proposed by Amir *et al.* [4] that finds the occurrence of a single pattern. Our algorithm runs in $O(n+m^2+r)$ time using $O(n+m^2)$ space, where n is the length of a given LZW compressed text, m is the total length of a given patterns, and r is the number of pattern occurrences. Technically, it simulates the move of the Aho-Corasick pattern matching machine on an LZW compressed text. We also present another algorithm for a single pattern that is based on bit-parallelism. It is indeed fast when the pattern length is not greater than the word length, which is 32 or 64 in current architecture. After an $O(m)$ time and space preprocessing of a pattern, it scans an LZW compressed text in $O(\lceil m/w \rceil(n+r))$ time and reports all occurrences of the pattern, where w is the word length. We implemented both algorithms and verified that they are about twice faster than a decompression followed by a search with **agrep**.

Keywords: bit-parallelism, compressed string matching, combinatorial pattern matching, LZ family compression, mutiple pattern,

1 Introduction

Recently, the *compressed pattern matching* problem has attracted special concern where the goal is to find a pattern in a compressed text without decompressing it. The problem was first defined by Amir and Benson [3], and several researchers have tackled this problem for various compression methods (see an excellent survey paper [23]).

Amir, Benson, and Farach[4] addressed the LZW compression[27] and presented a series of algorithms having various time and space complexities ($O(n+m^2)$ time and space, $O(n \log m + m)$ time and $O(n+m)$ space, and so on, where n is the length of compressed text and m is the length of pattern). Among them, we focus on the $O(n+m^2)$ time and space algorithm as it runs in linear time proportional to the compressed text length. The algorithm can be viewed as two functions that simulate

[1]The first author is a Research Fellow of the Japan Society for the Promotion of Science (JSPS). The author's research is partly supported by Grant-in-Aid for JSPS research fellows (12000410).

[2]Currently working at NEC.

the move of the KMP automaton [19]. This view enables us to simplify the algorithm and then extend it to the multiple pattern matching problem.

In this paper, we give an algorithm for finding multiple patterns in an LZW compressed text. It simulates the move of the Aho-Corasick multipattern matching machine [2]. The algorithm runs in $O(n + m^2 + r)$ time using $O(n + m^2)$ space, where n is the length of the compressed text, m is the total length of the patterns, and r is the number of occurrences of the patterns. The $O(r)$ time is devoted only to reporting the positions of the pattern occurrences. It is worth mentioning that this is the first compressed pattern matching algorithm that deals with multiple patterns.

We also present another algorithm that uses *bit-parallelism*. It is indeed fast when the pattern length is not greater than the word length in bits, which is 32 or 64 in current architecture.

The experimental results show that the proposed algorithms run faster than a decompression followed by an ordinary search. Especially, they run even about twice faster than the combination of **gunzip** and the exact match routine of the software package **agrep** [28], known as the fastest pattern matching tool.

2 Related works

In this paper, we consider the compressed pattern matching problem for the LZW compression. As well-known, the LZW compression is a variation of the Ziv-Lempel family (LZ77[30], LZ78[31], LZW, and so on). Amir, Benson, and Farach[4] presented algorithms for searching an LZW compressed text for a single pattern. We presented in [18] an extension of [4] to multiple pattern searching, together with the first experimental results in this area.

Moreover, we introduced in [16] a unifying framework, named *collage system*, which abstracts various dictionary-based methods. We showed a general pattern matching algorithm for text string described in terms of collage system. The algorithm can be applied to the problem for any compression methods, such as the Ziv-Lempel family, the RE-PAIR [20], and static dictionary-based methods. Of course, it cannot be applied to non-dictionary based methods, such as the one using anti-dictionaries [8] and the Burrows Wheeler Transform [7]. However, Shibata, *et al.* [26]presented an algorithm based on a similar technique for the compression using anti-dictionaries.

As other works on the Ziv-Lempel family, Farach and Thorup[11] and Gąsieniec, *et al.* [13] addressed the LZ77 compression. Bit-parallel realization of [4] was independently proposed in [17, 22] and proved to be fast in practice for a short pattern.

Recently, new practical results appeared. Miyazaki, *et al.* [21] addressed the Huffman encoding. Moura, *et al.* [9, 10] addressed a new compression scheme that uses a word-based Huffman encoding with a byte-oriented code. Shibata, *et al.* [24, 25] addressed the byte-pair encoding [12], which is a simple version of the RE-PAIR. Their algorithms run even faster than pattern matching in uncompressed texts.

This paper is based on [18] and [17].

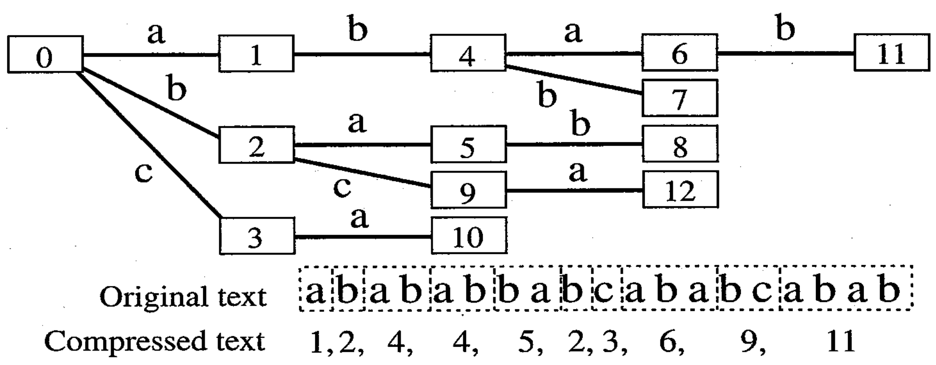
0
1
2
3
4
5
6
7
8
9
10
11
12
a
b
c
Original text a b a b a b b a b c a b a b c a b a b
Compressed text 1, 2, 4, 4, 5, 2, 3, 6, 9, 11

FIG. 1. Dictionary trie.

3 Preliminaries

In the following subsections we briefly sketch the LZW compression, and review the Aho-Corasick pattern matching machine and the generalized suffix trie[15]. These data structure are used in our algorithm. First, we introduce some notation. Let Σ be a finite set of characters, called an *alphabet*, and Σ^* be the set of strings over Σ. We denote the length of $u \in \Sigma^*$ by $|u|$. We call the string whose length is 0 the *null string*, and denote it by ε. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. We denote by $u[i]$ the ith character of a string u, and by $u[i:j]$ the string $u[i]u[i+1]...u[j]$, $1 \le i \le j \le |u|$. When $j < i$, let $u[i:j] = \varepsilon$. When a string u can be written as $u = xyz$, the strings x,y, and z are called a *prefix*, a *factor*, and a *suffix* of u, respectively. Let *Prefix*(u) be the set of prefixes of a string u, and let *Prefix*$(S) = \bigcup_{u \in S}$ *Prefix*(u) for a set S of strings. We also define the sets *Suffix* and *Factor* in a similar way. We denote the cardinality of a set V by $|V|$.

3.1 LZW compression

The LZW compression [27] is a very popular compression method. It is adopted as the **compress** command of UNIX, for instance. It parses a text into *phrases* and replaces them with pointers to a *dictionary*. The dictionary initially consists of the characters in Σ. The compression procedure repeatedly finds the longest match at the current position and updates the dictionary by adding the concatenation of the match and the next character. The dictionary is implemented as a trie structure, in which each node represents a phrase in it. The matches are encoded as integers associated with the corresponding nodes of the dictionary trie. The update of the dictionary is executed in $O(1)$ time by creating a new node labeled by the next character as a child of the node corresponding to the current match.

FIG. 1 shows the dictionary trie for the text *ababababbabcababcabab*, assuming the alphabet $\Sigma = \{a, b, c\}$. Hereafter, we identify the string u with the integer representing it, if no confusion occurs.

The dictionary trie is removed after the compression is completed since it can be reconstructed from the compressed text. In the decompression, the original text is obtained with the aid of the reconstructed dictionary trie. This decompression takes linear time proportional to the length of the original text. However, if the original text is not required, the dictionary trie can be built only in $O(n)$ time, where n is the length of the compressed text. The algorithm for constructing the dictionary trie from an LZW compressed text is summarized in FIG. 2.

```
Input.    An LZW compressed text $u_1 u_2 \dots u_n$.
Output.   Dictionary $D$ represented in the form of a trie.
Method.
begin
    $D := \Sigma$;
    for $i := 1$ to $n - 1$ do begin
        if $u_{i+1} \leq |D|$ then
            let $a$ be the first character of $u_{i+1}$
        else
            let $a$ be the first character of $u_i$;
        $D := D \cup \{u_i \cdot a\}$
    end
end.
```

FIG. 2. Reconstruction of dictionary trie.

3.2 *Aho-Corasick pattern matching machine*

The Aho-Corasick pattern matching machine [2] (AC machine for short) is a finite state machine which simultaneously recognizes all occurrences of multiple patterns in a single pass through a text.

The AC machine for a finite set $\Pi \subseteq \Sigma^+$ of patterns is specified by the three functions:

goto function $g : Q \times \Sigma \to Q \cup \{\mathit{fail}\}$,
failure function $f : Q \to Q$, and
output function $o : Q \to 2^{\Pi}$,

where Σ is an alphabet, Q is the set of states, and *fail* is a special value not in Q. FIG. 3 shows the AC machine for the patterns $\Pi = \{aba, ababb, abca, bb\}$ with $\Sigma = \{a, b, c\}$.

Define the state transition function $\delta : Q \times \Sigma \to Q$ by

$$\delta(q, a) = \begin{cases} g(q, a) & \text{if } g(q, a) \neq \mathit{fail}, \\ g(f(q), a) & \text{otherwise,} \end{cases}$$

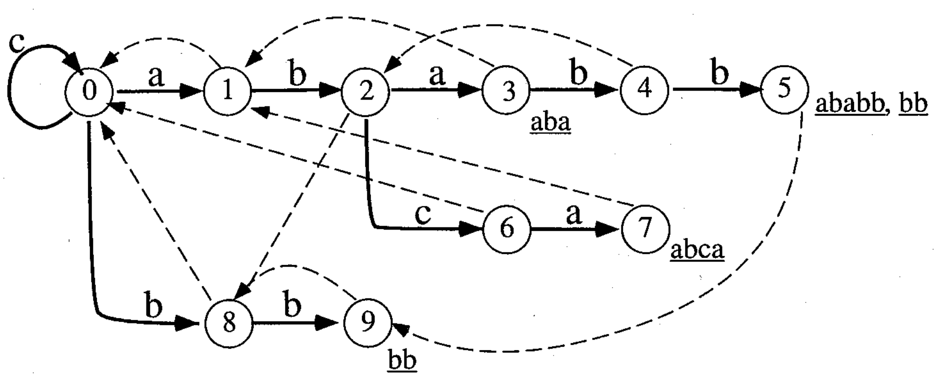

FIG. 3. Aho-Corasick machine for $\Pi = \{aba, ababb, abca, bb\}$.
The solid and the broken arrows represent the goto and the failure functions, respectively. The underlined strings adjacent to the states mean the outputs from them.

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

$$\delta(q, \varepsilon) = q \quad \text{and} \quad \delta(q, ua) = \delta(\delta(q, u), a),$$

where $q \in Q$, $u \in \Sigma^*$, and $a \in \Sigma$.

It should be noted that the states of the AC machine have a one-to-one correspondence with the prefixes of the patterns. For example, the initial state 0 corresponds to the empty string ε and the state 4 corresponds to the string *abab* in FIG. 3. Hereafter, we identify a pattern prefix with the state representing it. Thus we can identify Q with *Prefix*(Π).

The following lemma characterizes the state transition function δ of the AC machine. This is a modified version of Lemma 3 in [2].

LEMMA 3.1
Let $q \in Q = \mathit{Prefix}(\Pi)$, $u \in \Sigma^*$, and let $p = \delta(q, u)$. Then, the string p is the longest string in the set $\mathit{Suffix}(qu) \cap Q$.

3.3 Generalized suffix trie

A *generalized suffix trie* [15] for a set Π of strings (GST, for short) is a trie, which represents the set of suffixes of the strings in Π. It is an extension of the suffix trie for a single string. FIG. 4 shows the GST for $\Pi = \{aba, ababb, abca, bb\}$.

Note that each node of the GST for Π corresponds to a string in *Factor*(Π). A node of GST for Π is said to be *explicit* if and only if either it represents a suffix of some pattern in Π or its out-degree is more than one. The nodes that are not explicit are said to be *implicit*. Note that the number of explicit nodes in the GST for Π is $O(m)$, whereas the number of all nodes is $O(m^2)$, where m is the total length of the strings in Π. The construction of the GST for Π takes $O(m^2)$ time and space.

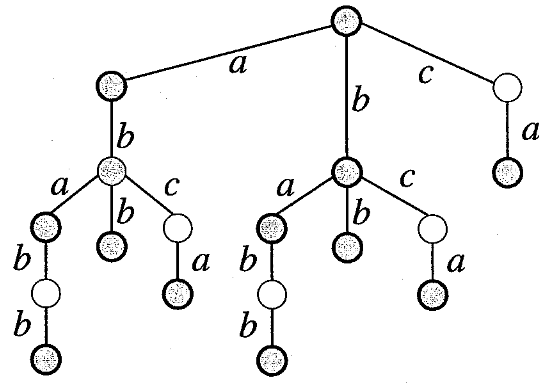

FIG. 4. Generalized Suffix Trie for $\Pi = \{aba, ababb, abca, bb\}$.
The explicit nodes are shaded, and the nodes which represent suffixes of patterns in Π are indicated by the thick circles.

4 Basic idea

The basic idea of our algorithm is to build a pattern matching machine that runs on an LZW compressed text and simulates the behavior of the AC machine on the original text. That is, for every integer u of the compressed text such machine makes one state transition that corresponds to the consecutive state transitions of the AC machine caused by the string u. Let *Jump* be the new state transition function. Namely, the function *Jump* is the limitation of $\delta : Q \times \Sigma^* \to Q$ to the domain $Q \times D$, where D is the set of phrases in the dictionary of the LZW compression. By using the function *Jump*, we can simulate the state transitions of the AC machine. However, the AC machine being simulated may pass through states with outputs in one step of the new machine. To avoid missing such outputs the new machine should be a Mealy type sequential machine (Section 2.7 of [14]), with an output function from $Q \times D$ to $2^{\{1,2,\cdots\}\times\Pi}$ defined as

$$Output(q, u) = \{\langle i, \pi\rangle \mid 1 \le i \le |u| \text{ and } \pi \in o(\delta(q, u[1..i]))\}.$$

That is, $Output(q, u)$ stores all outputs emitted by the AC machine during the state transitions from the state q reading the string u. Though the above idea is similar to that of Amir *et al.* [4], our definition of *Output* differs from their definition in that it is designed to report *all* occurrences of *multiple* patterns.

Note that the domains of the functions *Jump* and *Output* are both $Q \times D$, and the set D grows incrementally when reading the compressed text. Therefore the data structures required for *Jump* and *Output* fall into two classes according to whether they depend only on the patterns but not on D. Thus the algorithm consists of two parts: preprocessing the patterns and scanning the compressed text. The functions *Jump* and *Output* are partially constructed in the preprocessing phase, and then updated incrementally in the text scanning phase. The proposed algorithm can be summarized as in FIG. 5. FIG. 6 shows the move of the new machine on the compressed text of FIG. 1.

As shown later, the function *Jump* can be built to return its value in $O(1)$ time, and

```
Input.    Set of patterns $\Pi$ and LZW compressed text $u_1 u_2 \dots u_n$.
Output.   All positions of the original text at which pattern ends.
Method.
begin
    /* Preprocessing phase */
    Construct from $\Pi$ the AC machine and the GST;
    Initialize the dictionary trie and the functions Jump and Output;

    /* Text scanning phase */
    $\ell := 0$; $state := 0$;
    for $i := 1$ to $n$ do begin
        for each $\langle d, \pi \rangle \in Output(state, u_i)$ do
            report an occurrence of pattern $\pi$ that ends at position $\ell + d$;
        $state := Jump(state, u_i)$;
        $\ell := \ell + |u_i|$;
        Update the dictionary trie and the functions Jump and Output
    end
end.
```

FIG. 5. Pattern matching algorithm.

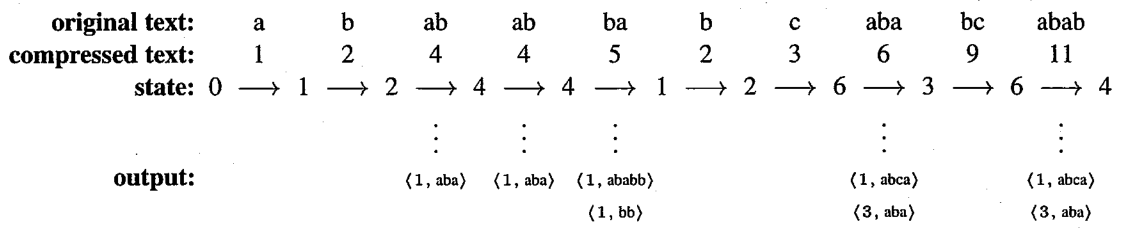
original text: a b ab ab ba b c aba bc abab
compressed text: 1 2 4 4 5 2 3 6 9 11
state: 0 1 2 4 4 1 2 6 3 6 4
output:
⟨1, aba⟩
⟨1, aba⟩
⟨1, ababb⟩ ⟨1, bb⟩
⟨1, abca⟩ ⟨3, aba⟩
⟨1, abca⟩ ⟨3, aba⟩

FIG. 6. Move of the new machine.

the procedure for enumerating the elements of *Output* can be built to enumerate them in linear time proportional to the number of the elements, using $O(n + m^2)$ time and space, where n is the length of the compressed text and m is the total length of the patterns. Thus, the algorithm of FIG. 5 runs in $O(n + m^2 + r)$ time using $O(n + m^2)$ space, where r is the number of occurrences of patterns.

5 Algorithm in detail

In this section, we discuss how to compute *Jump* and *Output*. Our goal is to prove the following two theorems.

THEOREM 5.1
The state transition function *Jump* defined on $Q \times D$ can be built to return its value in $O(1)$ time, using $O(|D| + m^2)$ time and space.

TABLE 1. Table N_1 for $\Pi = \{aba, ababb, abca, bb\}$.
The blanks indicate *nil*.

state	a	b	c	ab	ba	bb	bc	ca	aba	abb	abc	bab	bca	abab	abca	babb	ababb
0	0	0		0		0			0		0			0	0		0
1	0	1		0	1	0	1		0		0	1	1	0	0	1	0
2	2	8	2	2		0		2	0	2	0			0	0		0
3	0	3		0	1	3	1		0		0	1	1	0	0	1	0
4	2	4	2	2		0		2	0	2	0			0	0		0
5	0	8		0		0			0		0			0	0		0
6	6	0		0		0			0		0			0	0		0
7	0	1		0	1	0	1		0		0	1	1	0	0	1	0
8	0	8		0		0			0		0			0	0		0
9	0	8		0		0			0		0			0	0		0

THEOREM 5.2
The procedure that takes as input $q \in Q$ and $u \in D$ and enumerates the set *Output*, can be built in $O(|D| + m^2)$ time and space, so that it runs in linear time proportional to $|Output(q, u)|$.

5.1 *Computation of Jump*

DEFINITION 5.3
Let $\pi \in \Sigma^+$ be a pattern, and let $u \in \Sigma^+$. Define

$$Occ(\pi, u) = \{q \in Prefix(\pi) \mid qu \in Prefix(\pi)\}.$$

Also define, for a set $\Pi \subset \Sigma^+$ of patterns,

$$Occ(\Pi, u) = \bigcup_{\pi \in \Pi} Occ(\pi, u).$$

For example, $Occ(\Pi, a) = \{\varepsilon, ab, abc\}$, $Occ(\Pi, bab) = \{a\}$, and $Occ(\Pi, aa) = \emptyset$ for $\Pi = \{aba, ababb, abca, bb\}$. Recall that $Q = Prefix(\Pi)$ (see Section 3.2). We can regard each element of $Occ(\Pi, u)$ as a state of the AC machine. For example, $Occ(\Pi, a) = \{0, 2, 6\}$, $Occ(\Pi, bab) = \{1\}$, and $Occ(\Pi, aa) = \emptyset$ in the AC machine of FIG. 3 for $\Pi = \{aba, ababb, abca, bb\}$.

DEFINITION 5.4
For any $(q, u) \in Q \times Factor(\Pi)$, let $N_1(q, u)$ be the longest string in $Suffix(q) \cap Occ(\Pi, u)$. If no such string, let $N_1(q, u) = nil$.

TABLE 1 shows the table N_1 for the AC machine of FIG. 3 where $\Pi = \{aba, ababb, abca, bb\}$. Note that the entries of N_1 are stored as integers that represent states of the AC machine. For example, $N_1(4, ab) = 2$ since $Suffix(4) \cap Occ(\Pi, ab) = \{0, 2\}$ and the string ab represented by state 2 is longer than ε represented by state 0. The next lemma, which can be proved by using Lemma 3.1, is an extension of the idea in [4] to the multiple pattern problem.

LEMMA 5.5
Let $(q,u) \in Q \times D$. Then,

$$Jump(q,u) = \begin{cases} N_1(q,u) \cdot u & \text{if } u \in \mathit{Factor}(\Pi) \text{ and } N_1(q,u) \neq \mathit{nil}, \\ \delta(\varepsilon, u) & \text{otherwise.} \end{cases}$$

PROOF. Let $p = \mathit{Jump}(q,u) = \delta(q,u)$. By Lemma 3.1, the string p is the longest string in the set

$$\mathit{Suffix}(qu) \cap Q.$$

From the definition of N_1, the string $N_1(q,u)$ is the longest string in the set

$$\mathit{Suffix}(q) \cap \mathit{Occ}(\Pi, u).$$

(Recall that $N_1(q,u) = \mathit{nil}$ if this is empty.) It is not difficult to show that, between the above two sets, the following relation holds: for any $p' \in \Sigma^*$,

$$p'u \in \mathit{Suffix}(qu) \cap Q \quad \Leftrightarrow \quad p' \in \mathit{Suffix}(q) \cap \mathit{Occ}(\Pi, u).$$

Therefore, when $\mathit{Suffix}(q) \cap \mathit{Occ}(\Pi, u) \neq \emptyset$, it holds that $\mathit{Jump}(q,u) = N_1(q,u) \cdot u$. On the other hand, when $\mathit{Suffix}(q) \cap \mathit{Occ}(\Pi, u) = \emptyset$, we can show that $\mathit{Suffix}(qu) \cap Q = \mathit{Suffix}(u) \cap Q$. Hence the string $\mathit{Jump}(q,u)$ is the longest string in $\mathit{Suffix}(\varepsilon \cdot u) \cap Q$, and this implies $\mathit{Jump}(q,u) = \delta(\varepsilon, u)$. ∎

By using the GST for Π, we can determine for any $u \in D$ whether $u \in \mathit{Factor}(\Pi)$. Namely, if there exists a node in the GST that represents u, then $u \in \mathit{Factor}(\Pi)$. In order to build the function *Jump*, we will construct two tables that store respectively the values of

- $\delta(\varepsilon, u) \quad (u \in D)$, and
- $\hat{N_1}(q,u) = N_1(q,u) \cdot u \quad (q \in Q, u \in D)$.

LEMMA 5.6
The table that stores the values of $\delta(\varepsilon, u)$ for the strings $u \in D$ can be computed in $O(|D|)$ time using $O(|D|)$ space.

PROOF. The values of $\delta(\varepsilon, u)$ for $u \in D$ can be computed incrementally when constructing the dictionary trie from the compressed text, and stored in nodes of the dictionary trie. The computation is as follows. Suppose that the values of $\delta(\varepsilon, v)$ are already computed for all existing nodes v of the dictionary trie. Suppose also that we have created a new node, and added a new edge labeled a from the node representing u to the new node that represents the string ua. Then, the value of $\delta(\varepsilon, ua)$ is obtained as $\delta(\delta(\varepsilon, u), a)$ by performing one state transition of the AC machine. This requires only $O(1)$ time. Since it obviously requires $O(1)$ space to store each value, the total complexities are $O(|D|)$ time and space. ∎

On the other hand, the values of $\hat{N_1}$ can be stored in a table whose size is $|Q| \times |\mathit{Factor}(\Pi)| = O(m^3)$. The table size can be reduced to $O(m^2)$ as shown below.

DEFINITION 5.7
For a node u in GST for Π, let $\overline{u}$ denote the nearest descendant of u that is explicit. Note that if u is explicit, then $\overline{u} = u$.

The table that stores the value $\overline{u}$ for every u can be built using $O(m^2)$ time and space by traversing over the GST. The next lemma follows directly from Definition 5.3 and Definition 5.7.

LEMMA 5.8
Let u be a node in GST for Π. Then, for any $\pi \in \Pi$,

$$Occ(\pi, u) = Occ(\pi, \overline{u}).$$

From Definition 5.4 and Lemma 5.8, we can prove the next lemma.

LEMMA 5.9
Let u be a node in GST for Π. Then, $N_1(q, u) = N_1(q, \overline{u})$ for any $q \in Q$.

For example, we see in TABLE 1 that $N_1(q, abc) = N_1(q, abca)$ and $N_1(q, bab) = N_1(q, babb)$ for any $q \in Q$.

The above lemma implies that the size of table N_1 can be reduced to $O(m^2)$. However, we need the values of $\hat{N_1}$(not of N_1). In the single pattern case, we can number each state $0, 1, ..., m$ as the length of pattern prefix it represents and holds, $\hat{N_1}(q, u) = N_1(q, u) + |u| = N_1(q, \overline{u}) + |u|$, as pointed out in [4]. Thus, it suffices to store the entries of N_1 only for the domain $Q \times \{\overline{u} \mid u \in \mathit{Factor}(\Pi)\}$. In the multiple pattern case, however, we need some additional effort. Recall that each entry of $\hat{N_1}$ corresponds to a state of the AC machine, that is, a node of the trie for Π. We see that in the trie, the node $\hat{N_1}(q, u)$ is an ancestor of the node $\hat{N_1}(q, \overline{u})$, and the distance between the two nodes is $|\overline{u}| - |u|$, where the distance is the length of the path from the ancestor to the descendant. Denote by $Ancestor(q, k)$ the ancestor of the node q with distance $k \geq 0$. Then,

$$\hat{N_1}(q, u) = Ancestor(\hat{N_1}(q, \overline{u}), |\overline{u}| - |u|).$$

Although the table that stores the values $Ancestor(q, k)$ can be built using $O(m^2)$ time and space, we need not to built it. We will give here a more efficient method. From Lemma 5.8, for any $q \in Occ(\pi, u)$ the path from the node qu to the node $q\overline{u}$ in the AC machine is non-branching and does not contain a node that corresponds to the end of a pattern. Hence, the nodes on such path have consecutive numbers if the trie is constructed according to the standard way shown in FIG. 7. This observation enables us to simplify the computation, namely,

$$Ancestor(\hat{N_1}(q, \overline{u}), |\overline{u}| - |u|) = \hat{N_1}(q, \overline{u}) - (|\overline{u}| - |u|).$$

For example, for $q = 5$ and $u = abc$, $\hat{N_1}(5, abc) = Ancestor(\hat{N_1}(5, abca), |abca| - |abc|) = \hat{N_1}(5, abca) - (|abca| - |abc|) = 7 - 1 = 6$.

LEMMA 5.10
Let $(q, u) \in Q \times D$. Then,

$$Jump(q, u) = \begin{cases} \hat{N_1}(q, \overline{u}) - (|\overline{u}| - |u|) & \text{if } u \in \mathit{Factor}(\Pi) \text{ and } N_1(q, \overline{u}) \neq nil, \\ \delta(\varepsilon, u) & \text{otherwise.} \end{cases}$$

```
Input.    Set of patterns $\Pi \subset \Sigma^*$.
Output.   Trie representing $\Pi$.
Method.
begin
    $newstate := 0$;
    for each $\pi \in \Pi$ do call $enter(\pi)$
end.

procedure $enter(\pi[1:\ell])$;
begin
    $state := 0$;
    for $i := 1$ to $\ell$ do begin
        if $g(state, \pi[i]) = fail$ then
            begin
                $newstate := newstate + 1$;
                $g(state, \pi[i]) := newstate$
            end;
        $state := g(state, \pi[i])$
    end
end;
```

FIG. 7. Construction of trie of AC machine.

LEMMA 5.11
The table that stores the values of $\hat{N}_1$ can be computed in $O(m^2)$ time using $O(m^2)$ space.

PROOF. We give a sketch of the algorithm for computing the table that stores the values of $\hat{N}_1$. The algorithm has two stages. In the first stage, for every pair (q, u) such that $N_1(q, u) = q$, let $\hat{N}_1(q, u) := qu$. A naive computation requires $O(m^3)$ time. Using the compacted GST in which the implicit nodes are eliminated, we can perform the computation in $O(m^2)$ time and space. In the second stage, we fill the entries for the other pairs (q, u) using the failure function f of the AC machine. For each node u in the compacted GST, perform the following action. For each node q in the trie of the AC machine in the breadth first order, if $\hat{N}_1(q, u)$ is not filled, then let $\hat{N}_1(q, u) := \hat{N}_1(f(q), u)$. The computation of this stage requires $O(m^2)$ time and space. Thus, the total time and space complexities of the algorithm are $O(m^2)$. ∎

We can prove Theorem 5.1 by using Lemmas 5.6, 5.10, and 5.11.

5.2 *Computation of Output*

It follows from the definition of *Output* that

$$Output(q, u) = \{\langle i, \pi\rangle \mid 1 \le i \le |u|, \pi \in \Pi, \text{ and } \pi \text{ is a suffix of string } q \cdot u[1..i]\}.$$

DEFINITION 5.12
Let $lps(u)$ be the longest prefix v of a string u such that $v \in Suffix(\Pi)$.

We partition the set $Output(q, u)$ into two sets according to the following lemma.

LEMMA 5.13
For any $q \in Q$ and any $u \in D$, $Output(q, u) = Output(q, lps(u)) \cup Output(\varepsilon, u)$.

PROOF. It is obvious that $Output(q, u) \supseteq Output(q, lps(u)) \cup Output(\varepsilon, u)$. Assume for a contradiction that there exists $\langle j, \pi \rangle \in Output(q, u) \setminus Output(q, lps(u)) \cup Output(\varepsilon, u)$. Since $\langle j, \pi \rangle \in Output(q, u) \setminus Output(q, lps(u))$, $|lps(u)| < j \leq |u|$ and π is a suffix of string $q \cdot u[1..j]$. If $|\pi| \leq j$, then $u[j - |\pi| + 1..j] = \pi$, which contradicts $\langle j, \pi \rangle \notin Output(\varepsilon, u)$. So we have $j < |\pi|$. Hence $u[1..j]$ is a prefix of u that is also a suffix of π. This contradicts the definition of $lps(u)$. ■

We will build three tables that store respectively the values of

- $lps(u) \quad (u \in D)$,
- $Output(\varepsilon, u) \quad (u \in D)$, and
- $Output(q, u) \quad (q \in Q, u \in Suffix(\Pi))$.

LEMMA 5.14
The table storing the values of $lps(u)$ for $u \in D$ can be computed in $O(|D|)$ time and space.

PROOF. The values of $lps(u)$ for $u \in D$ can be computed incrementally when constructing the dictionary trie from the compressed text and stored in the nodes of dictionary trie. The computation is as follows. Suppose that the values of $lps(v)$ are already computed for all existing nodes v of the dictionary trie. Suppose also that we have created a new node, and added a new edge labeled a from the node representing u to the new node that represents the string ua. Then, by traversing one edge of GST, the value of $lps(ua)$ is obtained as

$$lps(ua) = \begin{cases} ua & \text{if } ua \in Suffix(\Pi), \\ lps(u) & \text{otherwise.} \end{cases}$$

This requires only $O(1)$ time. The proof is complete. ■

LEMMA 5.15
The procedure that takes as input $u \in D$ and enumerates the set $Output(\varepsilon, u)$, can be built in $O(|D|)$ time using $O(|D|)$ space, so that it runs in linear time proportional to $|Output(\varepsilon, u)|$.

PROOF. The set $Output(\varepsilon, u)$ can be represented as $Output(\varepsilon, u) = Output(\varepsilon, prev(u)) \cup \{|u| \mid \delta(\varepsilon, u) \in \Pi\}$, where $prev(u)$ is the longest proper prefix of a string u whose suffix is in Π. The function $prev(u)$ can be computed incrementally when constructing the dictionary trie and stored in its nodes, in a similar way to $lps(u)$. Thus, we need to store for any $u \in D$ the value $prev(u)$ and the information about whether $\delta(\varepsilon, u) \in \Pi$. The enumeration procedure is summarized in FIG. 8. The proof is now complete. ■

```
**procedure** $enumOutput_1(offset, u)$
**begin**
    $v := u$;
    **while** $|lps(u)| < |v|$ **do begin**
        $q := \delta(\varepsilon, v)$;
        **for each** $\pi \in o(q)$ **do**
            report an occurrence of pattern $\pi$ that ends at position $offset + |v|$;
        $v := prev(v)$
    **end**
**end**;
```

FIG. 8. Enumeration of *Output*(ε, u).

TABLE 2. Table *Next* for $\Pi = \{aba, ababb, abca, bb\}$.
The asterisks indicate the pairs (p, v) with $o(p) \neq \emptyset$.

state	aba	ba	a	ababb	babb	abb	bb	b	abca	bca	ca
0	(1,ba)	(8,a)	(1,ε)	(1,babb)	(8,abb)	(1,bb)	(8,b)	(8,ε)	(1,bca)	(8,ca)	(0,a)
1	(1,ba)	(2,a)	(1,ε)	(1,babb)	(2,abb)	(1,bb)	(2,b)	(2,ε)	(1,bca)	(2,ca)	(0,a)
2	(3,ba)*	(9,a)*	(3,ε)*	(3,babb)*	(9,abb)*	(3,bb)*	(9,b)*	(9,ε)*	(3,bca)*	(9,ca)*	(6,a)
3	(1,ba)	(4,a)	(1,ε)	(1,babb)	(4,abb)	(1,bb)	(4,b)	(4,ε)	(1,bca)	(4,ca)	(0,a)
4	(3,ba)*	(5,a)*	(3,ε)*	(3,babb)*	(5,abb)*	(3,bb)*	(5,b)*	(5,ε)*	(3,bca)*	(5,ca)*	(6,a)
5	(1,ba)	(9,a)*	(1,ε)	(1,babb)	(9,abb)*	(1,bb)	(9,b)*	(9,ε)*	(1,bca)	(9,ca)*	(0,a)
6	(7,ba)*	(8,a)	(7,ε)*	(7,babb)*	(8,abb)	(7,bb)*	(8,b)	(8,ε)	(7,bca)*	(8,ca)	(0,a)
7	(1,ba)	(2,a)	(1,ε)	(1,babb)	(2,abb)	(1,bb)	(2,b)	(2,ε)	(1,bca)	(2,ca)	(0,a)
8	(1,ba)	(9,a)*	(1,ε)	(1,babb)	(9,abb)*	(1,bb)	(9,b)*	(9,ε)*	(1,bca)	(9,ca)*	(0,a)
9	(1,ba)	(9,a)*	(1,ε)	(1,babb)	(9,abb)*	(1,bb)	(9,b)*	(9,ε)*	(1,bca)	(9,ca)*	(0,a)

We have to store the values of *Output*(q, u) for $(q, u) \in Q \times$ *Suffix*(Π), and the enumeration of the set *Output*(q, u) should be performed in linear time proportional to $|$*Output*$(q, u)|$. In the following discussion, we give such a data structure and an enumeration procedure.

DEFINITION 5.16
Let $q \in Q =$ *Prefix*(Π) and $u \in$ *Suffix*(Π) with $u \neq \varepsilon$. Denote by *Next*(q, u) the pair $(\delta(q, u[1]), u[2 : |u|])$. A sequence of pairs

$$(p_0, v_0) \to (p_1, v_1) \to \cdots \to (p_{|u|}, v_{|u|}),$$

where $p_i \in Q$ and $v_i \in$ *Suffix*(Π) for $i = 0, \ldots, |u|$, is said to be the *P-S sequence w.r.t.* (q, u) if and only if

$$\begin{aligned} &(p_0, v_0) = (q, u), \\ &(p_i, v_i) = Next(p_{i-1}, v_{i-1}) \quad \text{for } i = 1, 2, \ldots, |u|. \end{aligned}$$

TABLE 2 shows the table *Next* for the running example. For instance, the P-S sequence w.r.t. $(2, ababb)$ is

$$(2, ababb) \rightarrow (3, babb) \rightarrow (4, abb) \rightarrow (3, bb) \rightarrow (4, b) \rightarrow (5, \varepsilon).$$

Now we mention the construction of the table *Next*(q, u). The arguments $q \in$ *Prefix*(Π) and $u \in$ *Suffix*(Π) are given as a node of the AC machine and a node of the GST, respectively. The number of entries is $O(m^2)$ and each entry occupies only $O(1)$ space. To compute each entry, we need a mechanism to determine from the node representing u the character $u[1]$ and the node representing the string $u[2 : |u|]$ for every $u \in$ *Suffix*(Π). Such mechanism can be embedded in the GST by using $O(m^2)$ time and space. Therefore, the table *Next* can be built in $O(m^2)$ time and space.

By repeated references to the table *Next*, we can get for any $q \in Q$ and any $u \in$ *Suffix*(Π) the P-S sequence $(q_0, v_0) \rightarrow (q_1, v_1) \rightarrow \cdots \rightarrow (q_{|u|}, v_{|u|})$ w.r.t. (q, u). Thus we can enumerate all elements of *Output*$(q, u) = \bigcup_{i=1}^{|u|} \{\langle i, \pi \rangle \mid \pi \in o(q_i)\}$. However, the enumeration takes linear time proportional to the length of the string u. To make the enumeration time linear in the number of elements, we need the subsequence of the P-S sequence in which the states q_i have non-empty outputs.

DEFINITION 5.17
Let $(q_0, v_0) \rightarrow (q_1, v_1) \rightarrow \ldots \rightarrow (q_{|u|}, v_{|u|})$ be the P-S sequence w.r.t. (q, u). Define *Next*$^*(q, u) = (q_j, u[j + 1 : |u|])$ where j is the smallest integer such that $1 \leq j \leq |u|$ and $o(q_j) \neq \emptyset$.

TABLE 3 shows the values of *Next** for the running example. The table *Next** can be constructed from the table *Next* using $O(m^2)$ time and space by using the procedure in FIG. 9. Although the procedure has three-nested loops, the table can be built in $O(m^2)$ time since each entry of the table is calculated only once.

Using the table *Next** we can get the desired subsequence in linear time proportional to the length of it. In the running example, the obtained subsequence is

$$(2, ababb) \rightarrow (3, babb) \rightarrow (3, bb) \rightarrow (5, \varepsilon).$$

The procedure for enumerating *Output*(q, u) for $q \in Q$ and $u \in$ *Suffix*(Π) is summarized in FIG. 10. We thus have the following lemma.

LEMMA 5.18
The procedure that takes as input $q \in Q$ and $u \in$ *Suffix*(Π), and enumerates the set *Output*(q, u), can be built using $O(m^2)$ time and space, so that it runs in linear time proportional to $|$*Output*$(q, u)|$.

We can prove Theorem 5.2 by using Lemmas 5.13, 5.15, and 5.18.

6 Bit-parallel implementation

In this section we present an efficient implementation of [4] by using the technique called *bit-parallelism*[6]. This technique takes advantage of the fact that the processor

```
**procedure** ComputeNext* $(q, u)$
/* *Assume that Next$(q,u)$ is already computed.* */
*Prepare a stack S for a pair of q and u.*
**begin**
    **for each** $q \in Q$ **do**
        **for each** $u \in \mathit{Suffix}(\Pi)$ **do**
            **if** $\mathit{Next}^*(q,u)$ is already calculated **then**
                **continue**
            **else begin**
                $\mathit{push}(q,u)$;   /* *push the pair into S.* */
                $(q', u') := \mathit{Next}(q,u)$;
                **while** $o(q') \neq \emptyset$ **do begin**
                    $\mathit{push}(q', u')$;
                    $(q', u') := \mathit{Next}(q', u')$
                **end**;
                $(\mathit{neq}, \mathit{neu}) := (q', u')$;
                **while** $S$ is not empty **do begin**
                    $(q', u') := \mathit{pop}()$;   /* *pop the pair from S.* */
                    $\mathit{Next}^*(q', u') := (\mathit{neq}, \mathit{neu})$
                **end**
            **end**
**end**;
```

FIG. 9. Computation of *Next** from *Next*.

TABLE 3. Table *Next** for $\Pi = \{aba, ababb, abca, bb\}$.
The asterisks indicate the pairs (p, v) with $o(p) \neq \emptyset$.

state	aba	ba	a	ababb	babb	abb	bb	b	abca	bca	ca
0	(3,ε)*	(1,ε)	(1,ε)	(3,bb)*	(9,ε)*	(9,ε)*	(9,ε)*	(8,ε)	(7,ε)*	(1,ε)	(1,ε)
1	(3,ε)*	(3,ε)*	(1,ε)	(3,bb)*	(3,bb)*	(9,ε)*	(9,ε)*	(2,ε)	(7,ε)*	(7,ε)*	(1,ε)
2	(3,ba)*	(9,a)*	(3,ε)*	(3,babb)*	(9,abb)*	(3,bb)*	(9,b)*	(9,ε)*	(3,bca)*	(9,ca)*	(7,ε)*
3	(3,ε)*	(3,ε)*	(1,ε)	(3,bb)*	(3,bb)*	(9,ε)*	(5,ε)*	(4,ε)	(7,ε)*	(7,ε)*	(1,ε)
4	(3,ba)*	(5,a)*	(3,ε)*	(3,babb)*	(5,abb)*	(3,bb)*	(5,b)*	(5,ε)*	(3,bca)*	(5,ca)*	(7,ε)*
5	(3,ε)*	(9,a)*	(1,ε)	(3,bb)*	(9,abb)*	(9,ε)*	(9,b)*	(9,ε)*	(7,ε)*	(9,ca)*	(1,ε)
6	(7,ba)*	(1,ε)	(7,ε)*	(7,babb)*	(9,ε)*	(7,bb)*	(9,ε)*	(8,ε)	(7,bca)*	(1,ε)	(1,ε)
7	(3,ε)*	(3,ε)*	(1,ε)	(3,bb)*	(3,bb)*	(9,ε)*	(9,ε)*	(2,ε)	(7,ε)*	(7,ε)*	(1,ε)
8	(3,ε)*	(9,a)*	(1,ε)	(3,bb)*	(9,abb)*	(9,ε)*	(9,b)*	(9,ε)*	(7,ε)*	(9,ca)*	(1,ε)
9	(3,ε)*	(9,a)*	(1,ε)	(3,bb)*	(9,abb)*	(9,ε)*	(9,b)*	(9,ε)*	(7,ε)*	(9,ca)*	(1,ε)

```
procedure $enumOutput_2(offset, q, u)$
/* Third argument is limited to $u \in Suffix(\Pi)$. */
begin
    $(p, v) := (q, u)$;
    while $v \neq \varepsilon$ do begin
        $(p, v) := Next^*(p, v)$;
        $d := |u| - |v|$;
        for each $\pi \in o(q)$ do
            report an occurrence of pattern $\pi$ that ends at position $offset + d$
    end
end;
```

FIG. 10. Enumeration of *Output*(q, u).

works in parallel on all the bits of word length, say 32 or 64. By using the technique, we can simplify the calculation of *Jump* and *Output*.

Independently, Navarro and Raffinot[22] proposed a more general pattern matching algorithm on LZ family compressed text by using bit-parallelism.

6.1 Bit-parallel approach to uncompressed pattern matching

Pattern matching algorithms utilizing bit-parallelism were proposed independently by Abrahamson [1], Baeza-Yates and Gonnet [5], and Wu and Manber [29]. Here, we review the algorithm following the notation in [1]. Let $\pi = \pi[1 : m]$ be a pattern of length m and $\mathcal{T} = \mathcal{T}[1 : N]$ be a text of length N. For $k = 0, 1, \ldots, N$, let $R_k = \{1 \leq i \leq m \mid i \leq k \text{ and } \pi[1 : i] = \mathcal{T}[k - i + 1 : k]\}$, and for any $a \in \Sigma$, let $M(a) = \{1 \leq i \leq m \mid \pi[i] = a\}$. For a set A of integers and an integer k, let $A \oplus k = \{i + k \mid i \in A\}$ and $k \ominus A = \{k - i \mid i \in A\}$.

DEFINITION 6.1
Define the function $F : 2^{\{1,2,\ldots,m\}} \times \Sigma \to 2^{\{1,2,\ldots,m\}}$ by

$$F(S, a) = \big((S \oplus 1) \cup \{1\}\big) \cap M(a).$$

Using this function we can compute the values of R_k for $k = 1, 2, \ldots, N$ by

$$R_0 = \emptyset \quad \text{and} \quad R_{k+1} = F(R_k, \mathcal{T}[k+1]) \qquad (k \geq 0).$$

For $k = 1, 2, \ldots, N$, the algorithm reads the k-th character of the text, computes the value of R_k, and then examines whether $m \in R_k$. If $m \in R_k$, then $\mathcal{T}[k - m + 1 : k] = \pi$, that is, there is a pattern occurrence at position $k - m + 1$ of the text. Note that we can regard R_k as the states of the KMP automaton, and F acts as the state transition function.

$$
\begin{array}{rl|cccccccccccccccc}
\textbf{original text:} & & & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{c} \\
 & a & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
 & b & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
R_k: & a & 0 \to & 0 \to & 0 \to & 1 \to & 0 \to & 1 \to & 0 \to & 0 \to & 0 \to & 0 \to & 0 \to & 0 \to & 0 \to & 1 \to & 0 \to & 0 \\
 & b & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \underline{1} \\
 & & & & & & & & & & & & & & & & & \triangle
\end{array}
$$

FIG. 11. Behavior of the Shift-And algorithm.

The symbol $\triangle$ indicates that a pattern occurrence is found at that position.

When $m \leq 32$ (or $m \leq 64$), we can represent the sets R_k and $M(a)$ as m-bit integers. Then, we can calculate the integers R_k by

$$R_0 = 0, \quad \text{and} \quad R_{k+1} = ((R_k \ll 1) \mid 1) \,\&\, M(\mathcal{T}[k+1]) \qquad (k \geq 0),$$

where '$\ll \ell$' is the bit-shift operation which assigns the i-th bit to the $(i+\ell)$-th setting the first ℓ bits to zero, '$|$' is the bitwise-or of the computer word, and '&' is the bitwise logical product of the computer word. The bitwise logical product, the bit-shift, and the arithmetic operations can be performed in constant time. We find a pattern occurrence if $R_k \,\&\, 2^{m-1} \neq 0$. For example, the values of R_k for $k = 0, 1, \ldots$ are shown in FIG.11, where $\mathcal{T} = ababab babcababc$ and $\pi = ababc$.

The time complexity of this algorithm is $O(\lceil m/w \rceil N)$, where w is the word length in bits. We can regard the time complexity as $O(N)$ if $m \leq w$ (in fact such a case occurs very often).

6.2 *Application to LZW compressed pattern matching*

Assume that the text is parsed as $u_1 u_2 \ldots u_n$ by the LZW compression. Let $k_i = |u_1 u_2 \ldots u_i|$ for $i = 0, 1, \ldots, n$. Our idea is to compute only the values of R_{k_i} for $i = 1, 2, \ldots, n$, to achieve a linear time complexity which is proportional to the compressed text length n (not to the original text length N).

Extend F into the function from $2^{\{1,2,\cdots,m\}} \times \Sigma^*$ to $2^{\{1,2,\cdots,m\}}$ by

$$F(S, \varepsilon) = S \quad \text{and} \quad F(S, ua) = F(F(S, u), a),$$

where $S \subseteq \{1, \cdots, m\}$, $u \in \Sigma^*$ and $a \in \Sigma$. Then, the next lemma holds.

LEMMA 6.2
Suppose that the text is $\mathcal{T} = xuy$ with $x, u, y \in \Sigma^*$ and $u \neq \varepsilon$. Then,

$$R_{|xu|} = F(R_{|x|}, u).$$

If we have the values of F for the domain $2^{\{1,\cdots,m\}} \times D$, we can compute the value $R_{k_{i+1}} = F(R_{k_i}, u_{i+1})$ from R_{k_i} and u_{i+1} for each $i = 0, 1, \ldots, n-1$. As shown later, we can perform the computation of F only in $O(1)$ time by executing the bit-shift and the bitwise logical operations, using the function $\widehat{M}$ defined as follows.

DEFINITION 6.3
For any $u \in \Sigma^*$, let $\widehat{M}(u) = F(\{1, \ldots, m\}, u)$.

LEMMA 6.4
For any $S \subseteq \{1, \ldots, m\}$ and any $u \in \Sigma^*$,

$$F(S, u) = ((S \oplus |u|) \cup \{1, 2, \ldots, |u|\}) \cap \widehat{M}(u). \tag{6.1}$$

PROOF. By induction on $|u|$. It is easy for $u = \varepsilon$. Suppose $u = u'a$ with $u' \in \Sigma^*$ and $a \in \Sigma$. We have, from the induction hypothesis,

$$F(S, u') = ((S \oplus |u'|) \cup \{1, 2, \ldots, |u'|\}) \cap \widehat{M}(u').$$

It follows from the definition of F that, for any $S_1, S_2 \subseteq \{1, 2, \ldots, m\}$ and for any $a \in \Sigma$, $F(S_1 \cap S_2, a) = F(S_1, a) \cap F(S_2, a)$ and $F(S_1 \cup S_2, a) = F(S_1, a) \cup F(S_2, a)$. Then,

$$\begin{aligned} F(S, u) &= (F(S \oplus |u'|, a) \cup F(\{1, 2, \ldots, |u'|\}, a)) \cap F(\widehat{M}(u'), a) \\ &= ((S \oplus |u|) \cup \{1, 2, \ldots, |u|\}) \cap \widehat{M}(u). \end{aligned}$$

∎

Our remaining work for computing $F(S, u)$ is how to compute the value of the function $\widehat{M}(u)$.

LEMMA 6.5
The function that takes as input $u \in D$ and returns in $O(1)$ time the m-bit representation of the set $\widehat{M}(u)$, can be built in $O(\lceil m/w \rceil |D| + m)$ time using $O(\lceil m/w \rceil |D|)$ space.

PROOF. Since $\widehat{M}(u) \subseteq \{1, \ldots, m\}$, we can store $\widehat{M}(u)$ as an m-bit integer in the node u of the dictionary trie D. Suppose $u = u'a$ with $u' \in D$ and $a \in \Sigma$. $\widehat{M}(u)$ can be computed in $O(1)$ time from $\widehat{M}(u')$ and $M(a)$ when the node u is added to the dictionary trie since $\widehat{M}(u) = F(\widehat{M}(u'), a) = ((\widehat{M}(u') \oplus 1) \cup \{1\}) \cap M(a)$. Since the table $M(a)$ is computed in $O(\lceil m/w \rceil |\Sigma| + m)$ time using $O(\lceil m/w \rceil |\Sigma|)$ space and $\Sigma \subseteq D$, the total time and space complexities are $O(\lceil m/w \rceil |D| + m)$ and $O(\lceil m/w \rceil |D|)$, respectively. ∎

Now we have the following theorem from Lemma 6.2, Lemma 6.4, and Lemma 6.5.

THEOREM 6.6
The function which takes as input $(S, u) \in 2^{\{1, \ldots, m\}} \times D$ and returns in $O(1)$ time the m-bit representation of the set $F(S, u)$, can be built in $O(\lceil m/w \rceil |D| + m)$ time using $O(\lceil m/w \rceil |D|)$ space.

Equation (6.1) can be translated into

$$F(S,u) = \big((S \ll |u|) \mid ((1 \ll |u|) - 1)\big) \;\&\; \widehat{M}(u).$$

The function *Jump* can be computed using the above equation. In practice, this computation is faster than the one described in the previous section.

Now let us turn to *Output*. For $S \subseteq \{1, \ldots, m\}$ and $u \in D$, let $\mathit{Output}(S,u) = \big\{1 \le i \le |u| \;\big|\; m \in F(S, u[1:i])\big\}$, and let $U(u) = \big\{1 \le i \le |u| \;\big|\; i < m \text{ and } m \in \widehat{M}(u[1:i])\big\}$. Then, we have the following lemma.

LEMMA 6.7
For any $S \subseteq \{1, \ldots, m\}$ and any $u \in \Sigma^*$,

$$\mathit{Output}(S,u) = \big((m \ominus S) \cap U(u)\big) \cup \mathit{Output}(\emptyset, u).$$

PROOF.

$$\begin{aligned}
\mathit{Output}(S,u) &= \{1 \le i \le |u| \mid i < m \text{ and } m \in (S \oplus i) \cap \widehat{M}(u[1:i])\} \\
&\qquad \cup \{1 \le i \le |u| \mid m \le i \text{ and } m \in \widehat{M}(u[1:i])\} \\
&= \big((m \ominus S) \cap U(u)\big) \cup \mathit{Output}(\emptyset, u).
\end{aligned}$$

■

Since $\mathit{Output}(\emptyset, u)$ is the same as the one described in the previous section, the enumeration of it can be performed using the algorithm of FIG. 8. The remaining problem is how to enumerate the set $\big((m \ominus S) \cap U(u)\big)$.

Since $U(u) \subseteq \{1, \ldots, m\}$, we can store the set $U(u)$ as an m-bit integer in the node u of the dictionary trie D.

LEMMA 6.8
The function which takes as input $u \in D$ and returns in $O(1)$ time the m-bit representation of $U(u)$, can be built in $O(\lceil m/w \rceil |D| + m)$ time using $O(\lceil m/w \rceil |D|)$ space.

PROOF. By the definition of U, for any $u = u'a$ with $u' \in \Sigma^*$ and $a \in \Sigma$,

$$U(u) = U(u') \cup \big\{|u| \;\big|\; |u| < m \text{ and } m \in \widehat{M}(u)\big\}.$$

Then, we can prove the lemma in a similar way to the proof of Lemma 6.5. ■

To eliminate the cost of performing the operation $\ominus$ in $(m \ominus S) \cap U(u)$, we store the set $U'(u) = m \ominus U(u)$ instead of $U(u)$. Then, we can obtain the integer representing the set $S \cap U'(u)$ by one execution of the bitwise logical product operation. For an enumeration of a set represented as an m-bit integer, we need an operation to find the leftmost bit of the integer that is one. Since such operation takes $O(\lceil m/w \rceil)$ time [3], we can enumerate the set in $O(\lceil m/w \rceil \ell)$ time, where ℓ is the cardinality of the set.

From the above discussion, we have the following theorem.

[3]The operation can be done by using some bitwise logical operations and one logarithm operation. Since w is a constant number, the logarithm operation requires only $O(1)$ time if $m \le w$.

THEOREM 6.9
The procedure which takes as input $(S, u) \in 2^{\{1,\cdots,m\}} \times D$ and enumerates the set *Output*(S, u), can be realized in $O(|D| + m)$ time using $O(|D|)$ space, so that it runs in $O(\lceil m/w \rceil |\mathit{Output}(S, u)|)$ time.

Now we can simulate the behavior of the Shift-And algorithm on the original text completely. The algorithm is summarized as in FIG. 12. The behavior of the algorithm is illustrated in FIG. 13. This algorithm, moreover, can be extended to the problems of generalized pattern matching [1], pattern matching with k mismatches [4], and multiple pattern matching, like the Shift-And algorithm [5, 29].

7 Experimental results

We estimated the performances of the following programs:

- *Decompression followed by a search with the AC machine.*
  We tested this approach for **compress** and **gzip**, which are the famous UNIX tools based on the LZW and the LZ77 compressions, respectively. We combine the decompression programs and the AC machine using the UNIX 'pipe'. These methods are abbreviated as **uncompress+AC** and **gunzip+AC**, respectively. On the other hand, we embedded the search routine of the AC machine in the decompression programs, so that the AC machine processes the decoded characters 'on the fly'. These methods are abbreviated as **uncomAC** and **gunzipAC**, respectively.
- *Decompression followed by a search with* **agrep**.
  We tested this approach for **compress** and **gzip** again. We combine the decompression programs and **agrep** using the UNIX 'pipe'. The programs are abbreviated as **uncompress+agrep** and **gunzip+agrep**, respectively.
- *Compressed pattern matching.*
  Our algorithms, which are proposed in Sections 4 and 5. They are abbreviated as **AC on LZW** and **Bit-Parallel on LZW**, respectively.

Our experiment was carried out on an AlphaStation XP1000 with an Alpha21264 processor at 667MHz running Tru64 UNIX operating system V4.0F. The text files we used are:

Genbank. A subset of the GenBank database, which is an annotated collection of all publicly available DNA sequences. However, all fields other than accession number and nucleotide sequence were removed. The file size is about 17.1 Mbyte.

Medline. A clinically-oriented subset of Medline, consisting of 348,566 references. The file size is about 60.3 Mbyte.

Table 4 shows the compression ratios of these texts for **compress** and **gzip**. We selected patterns of lengths 5 to 30 from the texts randomly. We tested for 10 patterns of each length and took an average, where the same patterns were used for all methods.

Recall that the time complexities of our algorithms are linear with respect to the compressed text size n, not to the original size N. This property is desirable since the

[4] Navarro and Raffinot [22] also have achieved this extension.

```
Input.   An LZW compressed text $u_1 u_2 ... u_n$ and a pattern $\mathcal{P}$.
Output.  All positions at which $\mathcal{P}$ occurs.
begin
    /* Preprocessing */
    Construct the table $M$ from $\mathcal{P}$;
    $D := \emptyset$;  $U'(\varepsilon) := \emptyset$;  $prev(\varepsilon) := \varepsilon$;
    for each $a \in \Sigma$ do call $Update(\varepsilon, a)$;

    /* Text scanning */
    $k := 0$;    $R := \emptyset$;
    for $\ell := 1$ to $n$ do begin
        call $Update(u_{\ell-1}, u_\ell)$;   /* We assume $u_0 = \varepsilon$.*/
        for each $p \in \big(R \cap U'(u_\ell)\big) \cup Output(\emptyset, u_\ell)$ do
            report an occurrence of pattern $\pi$ that ends at position $k + p$;
        $R := \big((R \oplus |u_\ell|) \cup \{1, 2, \ldots, |u_\ell|\}\big) \cap \widehat{M}(u_\ell)$;
        $k := k + |u_\ell|$
    end
end.

procedure $Update(u, v)$
begin
    if $v \leq |D|$ then
        let $a$ be the first character of $v$
    else
        let $a$ be the first character of $u$;
    $D := D \cup \{u \cdot a\}$;
    $\widehat{M}(u \cdot a) := ((\widehat{M}(u) \oplus 1) \cup \{1\}) \cap M(a)$;
    if $|u \cdot a| < m$ then
        if $m \in \widehat{M}(u \cdot a)$ then
            $U'(u \cdot a) := U'(u) \cup \{m - |u \cdot a|\}$
        else
            $U'(u \cdot a) := U'(u)$
    else begin
        $U'(u \cdot a) := \emptyset$;
        if $m \in \widehat{M}(u)$ then
            $prev(u \cdot a) := u$
        else
            $prev(u \cdot a) := prev(u)$
    end
end;
```

FIG. 12. Bit-parallel approach to LZW compressed pattern matching

$$\begin{array}{rcccccccccccccccccccc}
\textbf{original text:} & & \text{a} & & \text{b} & & \text{ab} & & \text{ab} & & \text{ba} & & \text{b} & & \text{c} & & \text{aba} & & \text{bc} & \\
\textbf{compressed text:} & & 1 & & 2 & & 4 & & 4 & & 5 & & 2 & & 3 & & 6 & & 9 & \\
a & 0 & & 1 & & 0 & & 0 & & 0 & & 1 & & 0 & & 0 & & 1 & & 0 \\
b & 0 & & 0 & & 1 & & 1 & & 1 & & 0 & & 1 & & 0 & & 0 & & 0 \\
R_k\colon\ a & 0 & \longrightarrow & 0 & \longrightarrow & 0 & \longrightarrow & 0 & \longrightarrow & 0 & \longrightarrow & 0 & \longrightarrow & 0 & \longrightarrow & 0 & \longrightarrow & 1 & \longrightarrow & 0 \\
b & 0 & & 0 & & 0 & & 1 & & 1 & & 0 & & 0 & & 0 & & 0 & & 0 \\
c & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 1 \\
 & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
\textit{Output}(R_k, u_\ell)\colon & & \emptyset & & \emptyset & & \emptyset & & \emptyset & & \emptyset & & \emptyset & & \emptyset & & \emptyset & & \{2\} &
\end{array}$$

FIG. 13. Behavior of algorithm of Fig. 12.

TABLE 4. Compression ratio (%).

text (original size)	**compress**	**gzip**
Genbank (17.1Mbyte)	26.80	23.15
Medline (60.3Mbyte)	42.34	33.35

best LZW compression gives $n = \sqrt{2N}$. However, the LZW compression normally gives n that is linear with respect to N. In fact, the LZW compressions of Genbank and Medline give $n \approx 0.18N$ and $n \approx 0.22N$, respectively. Thus the constant factor is crucial.

FIG.14 shows the running times (CPU time), where the preprocessing times were included. We observed the following facts.

1. The difference between **uncompress+AC** and **gunzip+AC** corresponds to the difference between the decompression times of **uncompress** and **gunzip**. The difference between **uncompress+agrep** and **gunzip+agrep** is the same.
2. The approach that uses the UNIX pipe is slower than the others. Even **gunzip+agrep** is slower than **uncomAC**.
3. The difference between the approaches using **AC** and **agrep** is small in comparison with the difference between **gunzipAC** and **Bit-Parallel on LZW**. Thus, even if we could embed the search routine of the **agrep** in **gunzip**, it must be slower than **Bit-Parallel**.
4. The proposed algorithms are faster than the other approaches. Especially, it runs about 1.8 times faster than **gunzip+agrep** for Medline, and about twice faster for Genbank.

In general, the searching time is the sum of (1) the file I/O time and (2) the CPU time consumed for compressed pattern matching. Text compression reduces the file I/O time at the same ratio as the compression ratio while it may increase the CPU time. When the data transfer is slow, we have to give a weight to the reduction of the file I/O time, and a good compression ratio leads to a fast search. Therefore, we also performed the experiment in the following two different situations, where we used Medline as the text to be searched.

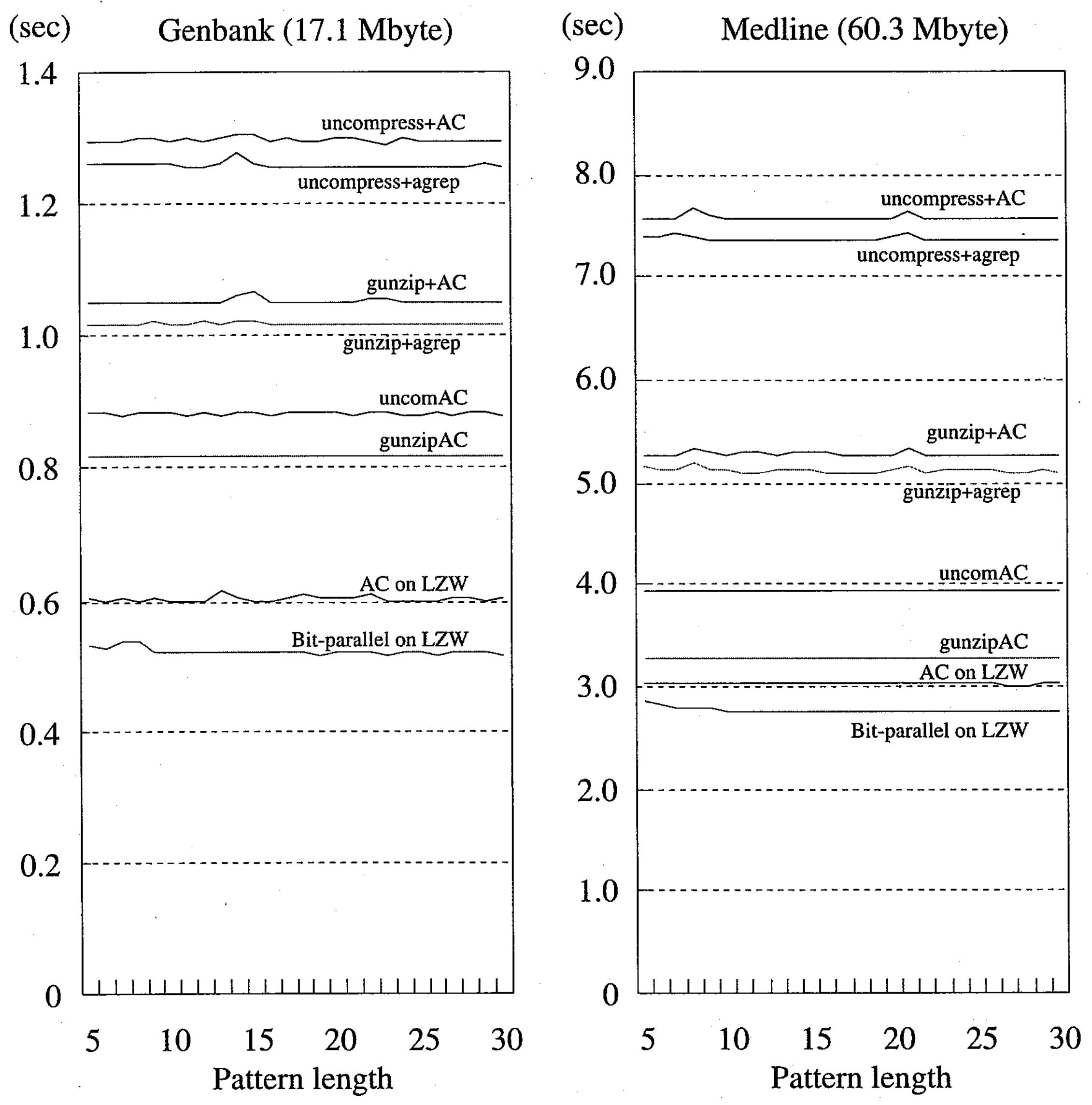

FIG. 14. Search time (CPU time).

Local. Workstation (AlphaStation XP1000) with local disk storage. The file transfer ratio is 24.9 Mbyte/sec.

Remote. Workstation (AlphaStation XP1000) with remote disk storage. The file transfer ratio is 6.57 Mbyte/sec.

The results are shown in TABLE 5. Even a decompression followed by an ordinary search was faster than the uncompressed search in Situation 2, whereas it wasn't in Situation 1. Thus we conclude that the compressed search can be faster than the uncompressed search when the data transfer is relatively slow, e.g. network environments.

We measured total search times including the preprocessing times above. However, the preprocessing times vary according to the pattern length m, especially the one of

TABLE 5. Elapsed time comparison.

method	Local	Remote
agrep on the original text	2.39	9.03
uncomAC	4.10	6.93
gunzipAC	3.38	6.49
AC on LZW	3.29	5.51
Bit-parallel on LZW	3.01	5.49

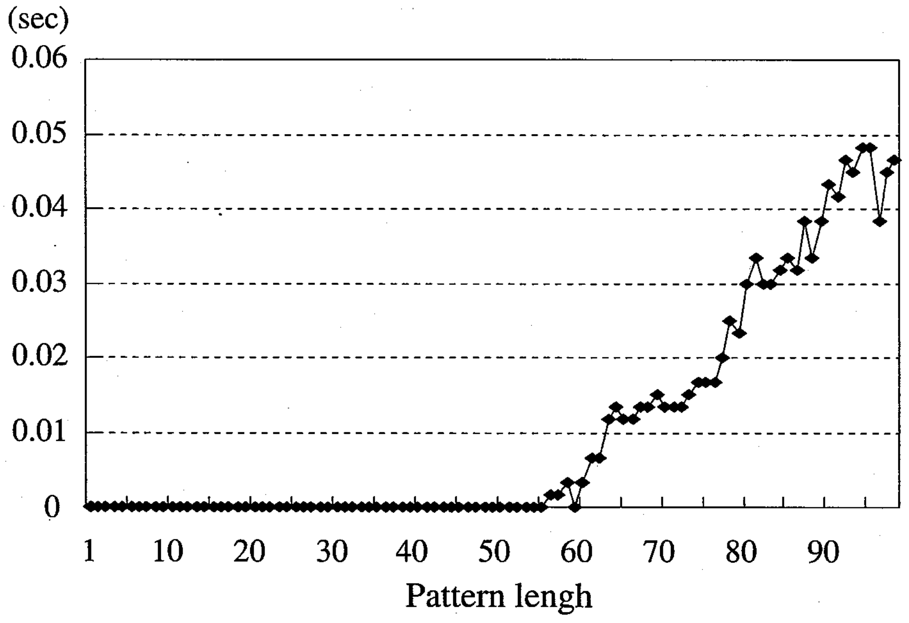

FIG. 15. Preprocessing time.

AC on LZW is $O(m^2)$. FIG.15 shows the preprocessing time of **AC on LZW** from $m = 5$ to $m = 100$. We observe that the preprocessing time is very small compared with the search time.

Practically, we used about $16|D| + (|\Sigma| + 25)m^2$ byte and $12|D| + 16$ byte memory space for the implimentations of **AC on LZW** and **Bit-parallel on LZW**, respectivly. Recall that $|D| = O(n)$. Since n is the compressed text length, the memory requirements seem vary large. However, the size of the dictionary trie is usually restricted to at most $2^{16} - 1 = 65535$ in the LZW compression, thus the memory requirements are about $1 \sim$ 2Mbyte at most.

8 Conclusion

In this paper we addressed the problem of searching in LZW compressed text directly, and presented the first algorithm that deals with multiple patterns. The algorithm simulates the move of the AC machine by scanning a compressed text in linear time proportional to the compressed text size. It should be emphasized that the number of patterns affects only the preprocessing time, which is very small in comparison with the scanning time. From the practical viewpoint, this property is desirable when we want to search many patterns at once. We also proposed another algorithm using bit-parallelism that is suitable for the case that the pattern length is not greater than the word length. Experimental results show that both algorithms are indeed faster than a decompression followed by an ordinary search, like the AC machine and **agrep**. Moreover, our algorithms are also faster than a fast search in the original text when the text data is stored in a remote disk device.

We have proved in this paper that compressed pattern matching is not only of theoretical interest but also of practical importance. The importance is rising recently because of a remarkable explosion of machine readable text files, which are often stored in a compressed form.

References

[1] K. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, December 1987.

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

[3] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. Data Compression Conference*, page 279, 1992.

[4] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1996.

[5] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Comm. ACM*, 35(10):74–82, 1992.

[6] R. Baeza-Yetes. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume 1, pages 465–476. Elsevier Science, September 1992.

[7] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, SRC Research Report, 130 Lytton Avenue Palo Alto, California 94301, May 1994. gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z.

[8] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *Proc. 26th International Colloquium on Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 261–270. Springer-Verlag, 1999.

[9] E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. 5th International Symp. on String Processing and Information Retrieval*, pages 90–95. IEEE Computer Society, 1998.

[10] E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast sequencial searching on compressed texts allowing errors. In *Proc. 21st Ann. International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 298–306. York Press, 1998.

[11] M. Farach and M. Thorup. String-matching in Lempel-Ziv compressed strings. In *27th ACM STOC*, pages 703–713, 1995.

[12] P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

[13] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 4th Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403. Springer-Verlag, 1996.

[14] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[15] L. C. K Hui. Color set size problem with application to string matching. In *Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer-Verlag, 1992.

[16] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *6th International Symposium on String Processing and Information Retrieval*, pages 89–96. IEEE Computer Society, 1999.

[17] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, pages 1–13. Springer-Verlag, 1999.

[18] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In James A. Atorer and Martin Cohn, editors, *Proc. Data Compression Conference '98*, pages 103–112. IEEE Computer Society, 1998.

[19] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput*, 6(2):323–350, 1977.

[20] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference '99*, pages 296–305. IEEE Computer Society, 1999.

[21] M. Miyazaki, S. Fukamachi, M. Takeda, and T. Shinohara. Speeding up the pattern matching machine for compressed texts. *Transactions of Information Processing Society of Japan*, 39(9):2638–2648, 1998. (in Japanese).

[22] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, pages 14–36. Springer-Verlag, 1999.

[23] W. Rytter. Algorithms on compressed strings and arrays. In *Proc. 26th Ann. Conf. on Current Trends in Theory and Practice of Infomatics*. Springer-Verlag, 1999.

[24] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *4th Italian Conference on Algorithms and Complexity*, Lecture Notes in Computer Science, pages 306–315. Springer-Verlag, 2000.

[25] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A boyer-moore type algorithm for compressed pattern matching. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science. Springer-Verlag, 2000. to appear.

[26] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, pages 37–49. Springer-Verlag, 1999.

[27] T. A. Welch. A technique for high performance data compression. *IEEE Comput.*, 17:8–19, June 1984.

[28] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, 1992.

[29] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35(10):83–91, October 1992.

[30] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inform. Theory*, IT-23(3):337–349, May 1977.

[31] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Inform. Theory*, 24(5):530–536, Sep 1978.

Received November 15, 1999.