Multiple Pattern Matching in LZW Compressed Text*

Takuya Kida Masayuki Takeda Ayumi Shinohara
Masamichi Miyazaki Setsuo Arikawa
{kida, takeda, ayumi, masamich, arikawa}@i.kyushu-u.ac.jp
Department of Informatics, Kyushu University 33
Fukuoka 812-8581, Japan

Abstract

In this paper we address the problem of searching in LZW compressed text directly, and present a new algorithm for finding multiple patterns by simulating the move of the Aho-Corasick pattern matching machine. The new algorithm finds all occurrences of multiple patterns whereas the algorithm proposed by Amir, Benson, and Farach finds only the first occurrence of a single pattern. The new 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 length of the total length of the patterns, and r is the number of occurrences of the patterns. We implemented a simple version of the algorithm, and showed that it is approximately twice faster than a decompression followed by a search using the Aho-Corasick machine.

1 Introduction

Recent years there has been a remarkable explosion of machine readable text data. Such text data are often stored in compressed forms. We therefore need a fast pattern matching technique for searching the compressed text directly. Several researchers tackled this problem. Eilam-Tsoreff and Vishkin[6] addressed the run-length compression, and Amir, Landau, and Vishikin[5], and Amir and Benson[2, 3] addressed its two-dimensional version. Farach and Thorup[7] and Gąsieniec, *et al.*[8] addressed the LZ77 compression[14]. Amir, Benson, and Farach[4] addressed the LZW compression[13]. Karpinski, *et al.*[9] and Miyazaki, *et al.*[12] addressed the straight-line programs. For a fast pattern matching when the text is compressed, we need algorithms which are faster than a decompression followed by a simple search. However, these studies seem to be done mainly from the theoretical viewpoints. Concerning the practical aspect of this problem, Manber [11] pointed out as below.

> It is not clear, for example, whether in practice the compressed search in [4] will indeed be faster than a regular decompression followed by a fast search.

*This research was supported in part by Grant-in-Aid for Scientific Research from the Ministry of Education, Science, Sports and Culture, Japan, (A) No. 07558051 and (C) No. 09680343.

In this paper we shall give a positive answer to this question.

In [4], Amir, Benson, and Farach presented a series of algorithms having various time and space complexities. From the viewpoint of speeding up the pattern matching, the most attractive one is the $O(n + m^2)$ time and space algorithm, where n is the length of the compressed text and m is the length of the pattern. It can be viewed as two functions which simulate the move of the KMP automaton[10]. This view enables us to simplify, improve, and then extend the algorithm to the multiple pattern matching problem.

In this paper, we give a new algorithm for finding multiple patterns in LZW compressed text. The new algorithm simulates the move of the Aho-Corasick pattern matching machine. Whereas the algorithm in [4] finds only the first occurrence of a single pattern, the new algorithm simultaneously recognizes all occurrences of multiple patterns. The new 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 for reporting the positions of occurrences of the patterns. The experimental result shows that the new algorithm is approximately twice faster than an LZW decompression followed by a search using the Aho-Corasick machine. Thus, the algorithm is shown to be efficient from the practical viewpoint.

2 Preliminaries

This section briefly sketches the LZW compression and the Aho-Corasick pattern matching machine.

2.1 LZW compression

An LZW compressed text is a sequence of integers. Each integer j indicates the node numbered j in a *dictionary trie*. A node of the dictionary trie represents the string that is spelled out by the path from the root to it. The set of strings represented by the nodes of the dictionary trie, denoted by D, is called *dictionary*. The dictionary trie is adaptively and incrementally built when compressing a text. Figure 1 shows the dictionary trie when the text is *abababbabcababcabab*, assuming the alphabet $\Sigma = \{a, b, c\}$. For example, the integer 9 of the compressed text represents the string *bc*. 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. It can be recovered from the compressed text. In the decoding phase, the original text is obtained with the aid of the recovered dictionary trie. This decoding process 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.

Figure 1: Dictionary trie.

2.2 AC machine

The Aho-Corasick pattern matching machine[1] (AC machine for short) is a finite state machine which simultaneously locates all occurrences of multiple patterns in a single pass through a text. The construction of the AC machine takes only linear time proportional to the sum of the lengths of the patterns.

Let Σ be an alphabet. Let $\Pi \subset \Sigma^+$ be a finite set of patterns. Let Q be the set of states, and $q_0 \in Q$ be the initial state. The AC machine for Π is then specified by the three functions:

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

Figure 2 shows the AC machine for the patterns $\Pi = \{aba, ababb, abca, bb\}$ with $\Sigma = \{a, b, c\}$.

Figure 2: 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.

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

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

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 ε is the empty string, $q \in Q$, $u \in \Sigma^*$, and $a \in \Sigma$.

3 Basic idea

We want to build a pattern matching machine that runs on an LZW compressed text and simulates the behaviors of the AC machine on the original text. For every integer u of the compressed text, the new machine makes one state transition which corresponds to the consecutive state transitions of the AC machine caused by the string u. Let *Next* be the new state transition function. Namely, the function *Next* is the limitation of $\delta : Q \times \Sigma^* \to Q$ to the domain $Q \times D$. By using the function *Next*, 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. Hence the new machine should be a Mealy type sequential machine, with an output function from $Q \times D$ to $2^{\{1,2,\cdots\} \times \Pi}$ defined by

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

That is, $\mathit{Output}(q, u)$ stores all outputs emitted by the AC machine during the state transitions from the state q reading the string u.

Note that the domain of the functions *Next* and *Output* is $Q \times D$, and the set D grows incrementally when reading the compressed text. Therefore the data structures required for the two functions fall into two classes according to whether or not they depend only on the patterns independently of D. Thus the algorithm consists of two parts: Preprocessing the patterns and scanning the compressed text. The functions *Next* and *Output* are partially constructed in the preprocessing phase, and then updated incrementally in the text scanning phase. The pattern matching algorithm in LZW compressed text can be summarized as in Figure 3. Figure 4 shows the move of the new machine on the compressed text of Figure 1.

As shown later, the function *Next* can be realized to answer in $O(1)$ time and the function *Output* can be realized to answer in linear time proportional to the size of the answer 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 Figure 3 runs in $O(n + m^2 + r)$ time using $O(n + m^2)$ space, where r is the number of occurrences of patterns. The following two sections discuss the realizations of the two functions.

Input. Set of patterns Π 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 Π the AC machine, the GST, and the other functions;

    /* Text scanning phase */
    $\ell := 0$;
    $state := q_0$;
    for $i := 1$ **to** n **do begin**
        for each $\langle d, \pi \rangle \in Output(state, u_i)$ **do**
            report an occurrence of pattern π that ends at position $\ell + d$;
        $state := Next(state, u_i)$;
        $\ell := \ell + |u_i|$;
        Update the dictionary trie, *Next* and *Output*
    end
end.

Figure 3: Pattern matching algorithm.

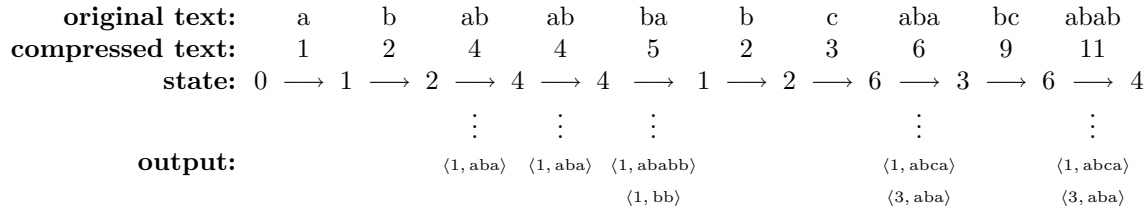

Figure 4: Move of new machine.

4 Realization of state transition function

This section discusses the realization of the state transition function *Next* defined on $Q \times D$.

First of all, we introduce some notations. When the string w can be written as $w = xyz$, the strings x,y, and z are called a *prefix*, a *factor*, and a *suffix* of w, 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 of strings S. We also define *Suffix* and *Factor* in a similar way. It should be noted that there is one-to-one correspondence between the states of the AC machine and 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 Figure 2. Hereafter, we identify a pattern prefix with the state representing it. Thus we can identify Q with *Prefix*(Π).

To realize the function *Next*, we use two data structures: the AC machine and the generalized suffix trie for Π. A *generalized suffix trie* for a set of strings (GST, for short) is a trie which represents the set of suffixes of the strings. It is an extension of

Table 1: Function N_1.

The blanks indicate undefined.

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

the suffix trie for a single string. Note that each node of the GST for Π corresponds to a string in $\mathit{Factor}(\Pi)$. A node of GST for Π is said to be *explicit* if and only if it represents a suffix of some pattern in Π, or its out-degree is more than one. The construction of the AC machine for Π takes $O(m)$ time and space, and the construction of the GST for Π takes $O(m^2)$ time and space, where m is the total length of patterns in Π.

Now, we consider the realization of *Next*. The following lemma characterizes the state transition function δ of the AC machine. This is a modified version of Lemma 3 in [1].

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

Definition 1 Let $\Pi \subset \Sigma^+$ be a finite set of patterns, and let $u \in \Sigma^+$. Define

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

The set $\mathit{Occ}(\Pi, u)$ means the set of states in $Q = \mathit{Prefix}(\Pi)$ at which string u occurs. For example, $\mathit{Occ}(\Pi, a) = \{0, 2, 6\}$, $\mathit{Occ}(\Pi, bab) = \{1\}$, and $\mathit{Occ}(\Pi, aa) = \emptyset$ in the AC machine of Figure 2 for $\Pi = \{aba, ababb, abca, bb\}$.

Definition 2 For any $(q, u) \in Q \times \Sigma^*$, let $N_1(q, u)$ be the longest string in $\mathit{Suffix}(q) \cap \mathit{Occ}(\Pi, u)$. If no such string, $N_1(q, u)$ is undefined.

Since $u \notin \mathit{Factor}(\Pi)$ implies that $N_1(q, u)$ is undefined, it is sufficient to consider only the values of N_1 for $Q \times \mathit{Factor}(\Pi)$. Table 1 shows the function N_1 for the AC machine of Figure 2 where $\Pi = \{aba, ababb, abca, bb\}$. Note that the values of N_1 are stored as the states of the AC machine. For example, $N_1(4, ab) = 2$ since $\mathit{Suffix}(4) \cap \mathit{Occ}(\Pi, ab) = \{0, 2\}$ 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 1, is an extension of the idea in [4] to the multiple pattern problem.

Lemma 2 *Let $(q, u) \in Q \times D$. Then,*

$$\mathit{Next}(q, u) = \begin{cases} N_1(q, u) \cdot u & \text{if } \mathit{Suffix}(q) \cap \mathit{Occ}(\Pi, u) \neq \emptyset \\ \delta(\varepsilon, u) & \text{otherwise.} \end{cases}$$

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 the nodes of it. 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 create a new node, and add a new edge labelled a from the node representing u to the new node which 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, and is stored in the new node. This requires only $O(1)$ time.

The function N_1 can be represented as a table of size $|Q| \times |Factor(\Pi)| = O(m^3)$, where m is the total length of patterns in Π. The table size, moreover, can be reduced to $O(m^2)$.

Definition 3 For any $u \in Factor(\Pi)$, let

$$[u] = \{ux \mid x \in \Sigma^* \text{ and } Occ(\Pi, u) = Occ(\Pi, ux)\}.$$

String $v \in [u]$ is said to be *maximal* if $va \notin [u]$ for all $a \in \Sigma$.

For example, $[ab] = \{ab\}$, $[c] = \{c, ca\}$, and $[ba] = \{ba, bab, babb\}$ when $\Pi = \{aba, ababb, abca, bb\}$.

Lemma 3 *Let $u \in Factor(\Pi)$. Then, $N_1(q, u) = N_1(q, x)$ for any $x \in [u]$.*

Note that the set $[u]$ may contain more than one maximal element. Let us denote by $\overline{u}$ one of the maximal elements of $[u]$. The function which returns the string $\overline{u}$ for a string u can be realized to answer in $O(1)$ time, and stored in the nodes of GST for Π. It can be computed in $O(m^2)$ time using $O(m^2)$ space. We can see that any maximal element in $[u]$ for any $u \in Factor(\Pi)$ is represented by an explicit node of GST. Since the number of the explicit nodes of GST is $O(m)$, the size of the set $\{\overline{u} \mid u \in Factor(\Pi)\}$ is $O(m)$. Therefore the function N_1 defined on $Q \times Factor(\Pi)$ can be represented as an $O(m^2)$ size table by using the function $\overline{u}$.

Let $\hat{N}_1(q, u) = N_1(q, u) \cdot u$. It should be mentioned that each value of $\hat{N}_1$ corresponds to a state of the AC machine, that is, a node of the trie for Π. We can 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 $|\overline{u}| - |u|$. Assuming the function $Ancestor(q, k)$ which returns the ancestor of the node q with distance k, the value of $\hat{N}_1(q, u)$ can be obtained by

$$\hat{N}_1(q, u) = Ancestor(\hat{N}_1(q, \overline{u}), |\overline{u}| - |u|).$$

The function *Ancestor* can be realized to answer in $O(1)$ time as an $O(m^2)$ size table which is constructed in $O(m^2)$ time. Thus, we can get the value of the function *Next* by

$$Next(q, u) = \begin{cases} Ancestor(\hat{N}_1(q, \overline{u}), |\overline{u}| - |u|) & \text{if } u \in Factor(\Pi), \\ \delta(\varepsilon, u) & \text{otherwise.} \end{cases}$$

The construction of the table which stores the values of $\hat{N}_1(q, \overline{u})$ can be constructed in $O(m^2)$ time using $O(m^2)$ space in a manner which is basically the same as the algorithm proposed by Amir, Benson, and Farach[4]. We now have the following theorem.

Theorem 1 *The state transition function Next defined on $Q \times D$ can be realized to answer in $O(1)$ time, using $O(|D| + m^2)$ time and space.*

5 Realization of output function

This section discusses the realization of the function *Output* defined on $Q \times D$. It follows from the definition of *Output* that

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

Now, we partition the set $\mathit{Output}(q, u)$ into two sets according to the following lemma.

Lemma 4 *Let $\tilde{u}$ be the longest prefix of u such that $\tilde{u} \in \mathit{Suffix}(\Pi)$. Let*

$$A(u) = \{\langle i, \pi\rangle \mid \pi \in \Pi, |\tilde{u}| < i \le |u|, \mathit{and}\ u[i - |\pi| + 1..i] = \pi\}.$$

Then, $\mathit{Output}(q, u) = \mathit{Output}(q, \tilde{u}) \cup A(u)$.

The set $A(u)$ can be represented by the two functions $Prev(u)$ and $\tilde{u}$, where $Prev(u)$ is the longest proper prefix of a string u whose suffix is in Π. Since these functions can be computed incrementally when constructing the dictionary trie and stored in its nodes, the representation of $A(u)$ requires $O(n)$ time and space. Now we have only to store the values of *Output* for $Q \times \mathit{Suffix}(\Pi)$ because of $\tilde{u} \in \mathit{Suffix}(\Pi)$.

Definition 4 Let $q \in Q$ and $u \in \mathit{Suffix}(\Pi)$ with $u \neq \varepsilon$. A sequence of states $q_1, q_2, \ldots, q_{|u|} \in Q$ is said to be the *state sequence w.r.t.* (q, u) if and only if

$$\begin{aligned} &q_1 = \delta(q, u[1]); \\ &q_i = \delta(q_{i-1}, u[i]) \quad \text{for } i = 2, 3, \ldots, |u|. \end{aligned}$$

Definition 5 Define the function $N_2 : Q \times (\mathit{Suffix}(\Pi) - \{\varepsilon\}) \to Q \times \mathit{Suffix}(\Pi)$ by

$$N_2(q_1, u_1) = (q_2, u_2) \quad \Leftrightarrow \quad a \in \Sigma,\ u_1 = a u_2, \text{ and } \delta(q_1, a) = q_2.$$

Table 2 shows the function N_2 for the AC machine of Figure 2 where $\Pi = \{aba, ababb, abca, bb\}$. By repeated calls of the function N_2, we can get the state sequence $q_1, q_2, \ldots, q_m$ w.r.t. (q, u). Thus we can enumerate all elements of $\mathit{Output}(q, u) = \bigcup_{i=1}^{m} \{\langle i, \pi\rangle \mid \pi \in o(q_i)\}$. However, the enumeration takes linear time proportional to the length of the string u. We need the subsequence of the above state sequence in which the states have non-empty outputs. Define the function N_2^* as follows.

Definition 6 Let $q_1, q_2, \ldots, q_m$ be the state sequence w.r.t. $(q, u) \in Q \times \mathit{Suffix}(\Pi)$. Define $N_2^*(q, u) = (q_j, u[j+1..|u|])$ where j is the smallest integer such that $1 \le j \le m$ and $o(q_j) \neq \emptyset$.

The function N_2^* can be constructed from N_2 using $O(m^2)$ time and space. Using N_2^* we can get the desired subsequence in linear time proportional to the length of it. We now have the following theorem.

Theorem 2 *The output function Output defined on $Q \times D$ can be realized to answer in linear time proportional to the size of the answer, using $O(|D| + m^2)$ time and space.*

Table 2: Function N_2.
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)

Figure 5: Running time.

6 Experimental results

To evaluate our algorithm from the practical viewpoint, we implemented the following three methods in the C++ language on Sun SPARCstation 20.

Method 1 A decompression into a temporary file which is followed by a search using the AC machine.

Method 2 A method in which the AC machine runs on the decompressed substring for each integer of the compressed text (without creating a temporary file).

Method 3 A simple version of the proposed algorithm in which the technique for reducing the size of N_1 is not applied.

Since Method 2 differs from Method 1 only in that it does not create a temporary file, it is clear that Method 2 is faster than Method 1. Ignoring the preprocessing time, the running time of Method 2 is $O(N)$ while that of Method 3 is $O(n)$, where N is the length of the original text and n is the length of the compressed text. The best LZW compression gives $n = \sqrt{2N}$. However, the LZW compression of typical English texts normally gives $n = N/2$. Thus, the constant factor hidden behind the O-notation plays a key role in the competition.

In the experiment, we used the Brown corpus as the text to be searched. The original size is about 6.8MB and the compressed size is about 3.4MB. We measured the CPU time to search for various sets of patterns. We excluded the preprocessing phase because we can ignore it when the total length of patterns m is sufficiently smaller than the text length N. The running time varies depending on the number of pattern occurrences. Let us define *the occurrence rate* to be the ratio of the number of pattern occurrences r to the original text length N. The relation between the CPU time and the occurrence rate is shown in Figure 5. Method 3 defeats both Method 1 and Method 2. It should be emphasized that Method 3 is approximately 1.5 to 1.6 times faster than Method 2.

References

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

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

[3] A. Amir and G. Benson. Two-dimensional periodicity and its application. In *Proc. 3rd Symposium on Discrete Algorithms*, page 440, 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] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms*, 13(1):2–32, 1992.

[6] T. Eilam-Tsoreff and U. Vishkin. Matching patterns in a string subject to multilinear transformations. In *Proc. International Workshop on Sequences, Combinatorics, Compression, Security and Transmission*, 1988.

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

[8] 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.

[9] M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing*, 4:172–186, 1997.

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

[11] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc. Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 113–124. Springer-Verlag, 1994.

[12] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1997.

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

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