Shift-And Approach to Pattern Matching in LZW Compressed Text

Takuya Kida, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa

Department of Informatics, Kyushu University 33
Fukuoka 812-8581, Japan
{kida, takeda, ayumi, arikawa}@i.kyushu-u.ac.jp

Abstract. This paper considers the Shift-And approach to the problem of pattern matching in LZW compressed text, and gives a new algorithm that solves it. The algorithm is indeed fast when a pattern length is at most 32, or the word length. After an $O(m + |\Sigma|)$ time and $O(|\Sigma|)$ space preprocessing of a pattern, it scans an LZW compressed text in $O(n + r)$ time and reports all occurrences of the pattern, where n is the compressed text length, m is the pattern length, and r is the number of the pattern occurrences. Experimental results show that it runs approximately 1.5 times faster than a decompression followed by a simple search using the Shift-And algorithm. Moreover, the algorithm can be extended to the generalized pattern matching, to the pattern matching with k mismatches, and to the multiple pattern matching, like the Shift-And algorithm.

1 Introduction

Pattern matching in compressed text is one of the most interesting topics in the combinatorial pattern matching. Several researchers tackled this problem. Eilam-Tzoreff and Vishkin [8] addressed the run-length compression, and Amir, Landau, and Vishikin [6], and Amir and Benson [2, 3] and Amir, Benson, and Farach [4] addressed its two-dimensional version. Farach and Thorup [9] and Gąsieniec, *et al.* [11] addressed the LZ77 compression [18]. Amir, Benson, and Farach [5] addressed the LZW compression [16]. Karpinski, *et al.* [12] and Miyazaki, *et al.* [15] addressed the straight-line programs. However, it seems that most of these studies were undertaken mainly from the theoretical viewpoint. Concerning the practical aspect, Manber [14] pointed out at CPM'94 as follows.

> It is not clear, for example, whether in practice the compressed search in [5] will indeed be faster than a regular decompression followed by a fast search.

In 1998 we gave in [13] an affirmative answer to the above question: We presented an algorithm for finding multiple patterns in LZW compressed text, which is a variant of the Amir-Benson-Farach algorithm [5], and showed that in practice the algorithm is faster than a decompression followed by a simple search.

Namely, it was proved that pattern matching in compressed text is not only of theoretical interest but also of practical interest. We believe that fast pattern matching in compressed text is of great importance since there is a remarkable explosion of machine readable text files, which are often stored in compressed forms.

On the other hand, the Shift-And approach [1, 7, 17] to the classical pattern matching is widely known to be efficient in many practical applications. This method is simple, but very fast when a pattern length is not greater than the word length of typical computers, say 32. In this paper, we apply this method to the problem of pattern matching in LZW compressed text and then give a new algorithm that solves it. Let m, n, r be the pattern length, the length of compressed text, and the number of occurrences of the pattern in the original text, respectively. The algorithm, after an $O(m + |\Sigma|)$ time and $O(|\Sigma|)$ space preprocessing of a pattern, scans a compressed text in $O(n + r)$ time using $O(n+m)$ space and reports all occurrences of the pattern in the original text. The $O(r)$ time is devoted only to reporting the pattern occurrences. Experimental results on the Brown corpus show that the proposed algorithm is approximately 1.5 times faster than a decompression followed by a search using the Shift-And method. Moreover, the algorithm can be extended to (1) the generalized pattern matching, to (2) the pattern matching with k mismatches, and to (3) the multiple pattern matching.

We assume, throughout this paper, that $m \leq 32$ and that the arithmetic operations, the bitwise logical operations, and the logarithm operation on integers can be performed in constant time.

The organization of this paper is as follows: We briefly sketch the LZW compression method, and the Shift-And pattern matching algorithm. We present our algorithm and discuss the complexity in Section 3. In Section 4, we show the experimental results in comparison with both an LZW decompression followed by a search using the Shift-And method and the previous algorithm presented in [13]. In Section 5 we shall discuss the extensions of the algorithm to the generalized pattern matching, to the pattern matching with k mismatches, and to the multiple pattern matching.

2 Preliminaries

We first define some notation. Let Σ, usually called an *alphabet*, be a finite set of characters, and Σ^* be a set of strings over Σ. We denote the length of $u \in \Sigma^*$ by $|u|$. We call especially the string whose length is 0 *null string*, and denote it by ε. We denote by $u[i]$ the ith character of a string u, and by $u[i:j]$ the string $u[i]u[i+1]...u[j]$, $1 \leq i \leq j \leq |u|$. For a set A of integers and an integer k, let $A \oplus k = \{i + k \mid i \in A\}$ and $k \ominus A = \{k - i \mid i \in A\}$.

In the following subsections we briefly sketch the LZW compression method and the Shift-And pattern matching algorithm.

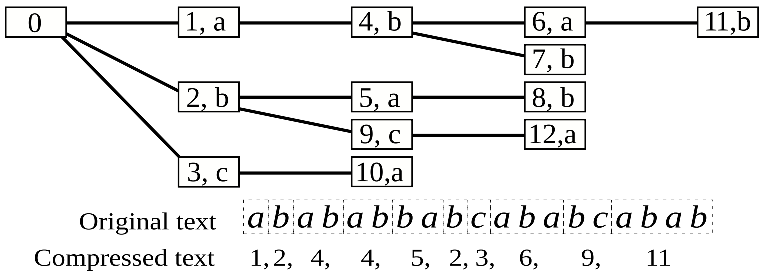

Fig. 1. Dictionary trie.

2.1 LZW Compression

The LZW compression 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 the *dictionary.* The dictionary initially consists of the characters in Σ. The compression procedure repeatedly finds the longest match in 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.

Figure 1 shows the dictionary trie for the text *abababbabcababcabab*, 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. It can be reconstructed from the compressed text. In the decompression, the original text is obtained with the aid of the recovered 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 a compressed text is summarized in Figure 2.

2.2 The Shift-And Pattern Matching Algorithm

The Shift-And pattern matching algorithm was proposed by Abrahamson [1], Baeza-Yates and Gonnet [7], and Wu and Manber [17]. In the following, we present the algorithm according to the notation in [1].

Let $\mathcal{P} = \mathcal{P}[1 : m]$ 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

$$R_k = \big\{1 \le i \le m \;\big|\; i \le k \text{ and } \mathcal{P}[1 : i] = \mathcal{T}[k - i + 1 : k]\big\}, \qquad (1)$$

```
Input.      An LZW compressed text $u_1 u_2 \dots u_n$.
Output.     Dictionary $D$ represented in the form of 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.

and for any $a \in \Sigma$, let

$$M(a) = \big\{1 \leq i \leq m \;\big|\; \mathcal{P}[i] = a\big\}. \tag{2}$$

Definition 1. *Define the function* $f : 2^{\{1,2,\dots,m\}} \times \Sigma \rightarrow 2^{\{1,2,\dots,m\}}$ *by*

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

where $S \subseteq \{1, \cdots, m\}$ *and* $a \in \Sigma$.

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

1. $R_0 = \emptyset$,
2. $R_{k+1} = f(R_k, \mathcal{T}[k+1]) \qquad (k \geq 0)$.

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

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

1. $R_0 = 0$,
2. $R_{k+1} = ((R_k \ll 1) + 1) \;\&\; M(\mathcal{T}[k+1]) \qquad (k \geq 0)$,

where '$\ll$' and '&' denote the bit-shift operation and the bitwise logical product, respectively. We can get a pattern occurrence if $R_k \& 2^{m-1} \neq 0$. For example, the values of R_k for $k = 0, 1, \dots$ are shown in Figure 3, where $\mathcal{T} = ababab babcababc$ and $\mathcal{P} = ababc$.

The time complexity of this algorithm is $O(mN)$. However, the bitwise logical product, the bit-shift, and the arithmetic operations on 32 bit integers can be performed at high speed, and thus be considered to be done in $O(1)$ time. Then we can regard the time complexity as $O(N)$ if m is at most 32 (in fact such a case occurs very often).

$$
\begin{array}{llcccccccccccccccc}
\textbf{original text:} & & & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{c} \\
 & a & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
 & b & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
R_k\text{:} & a & 0 \rightarrow & 0 \rightarrow & 0 \rightarrow & 1 \rightarrow & 0 \rightarrow & 1 \rightarrow & 0 \rightarrow & 0 \rightarrow & 0 \rightarrow & 0 \rightarrow & 0 \rightarrow & 0 \rightarrow & 0 \rightarrow & 1 \rightarrow & 0 \rightarrow & 0 \\
 & b & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \underline{1} \\
 & & & & & & & & & & & & & & & & & \triangle
\end{array}
$$

Fig. 3. Behavior of the Shift-And algorithm.

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

3 Proposed Algorithm

We want to design a new pattern matching algorithm that runs on an LZW compressed text and simulates the behavior of the Shift-And algorithm on the original text. Assume that the text is parsed as $u_1 u_2 \ldots u_n$. 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 not to the original text length N but to the compressed text length n.

Definition 2. *Let $\widehat{f}$ be the function f extended to $2^{\{1,\cdots,m\}} \times \Sigma^*$ by*

$$\widehat{f}(S, \varepsilon) = S \qquad \text{and} \qquad \widehat{f}(S, ua) = f(\widehat{f}(S, u), a),$$

where $S \subseteq \{1, \cdots, m\}$, $u \in \Sigma^*$ and $a \in \Sigma$.

Lemma 1. *Suppose that the text is $\mathcal{T} = xuy$ with $x, u, y \in \Sigma^*$ and $u \neq \varepsilon$. Then,*

$$R_{|xu|} = \widehat{f}(R_{|x|}, u).$$

Proof. It follows directly from the definition of $\widehat{f}$. ■

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

Definition 3. *For any $u \in \Sigma^*$, let $\widehat{M}(u) = \widehat{f}(\{1, \ldots, m\}, u)$.*

Lemma 2. *For any $S \subseteq \{1, \ldots, m\}$ and any $u \in \Sigma^*$,*

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

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,

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

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,

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

■

Lemma 3. *The function which takes as input $u \in D$ and returns in $O(1)$ time the m-bit representation of the set $\widehat{M}(u)$, can be realized in $O(|D| + m)$ time using $O(|D|)$ 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 $\widehat{M}(u) = f(\widehat{M}(u'), a) = \big((\widehat{M}(u') \oplus 1) \cup \{1\}\big) \cap M(a)$. Since the table $M(a)$ is computed in $O(|\Sigma| + m)$ time using $O(|\Sigma|)$ space and $\Sigma \subseteq D$, the total time and space complexities are $O(|D| + m)$ and $O(|D|)$, respectively. ■

Now we have the following theorem from Lemmas 1, 2, and 3.

Theorem 1. *The function which takes as input $(S, u) \in 2^{\{1, \cdots, m\}} \times D$ and returns in $O(1)$ time the m-bit representation of the set $\widehat{f}(S, u)$, can be realized in $O(|D| + m)$ time using $O(|D|)$ space.*

Since $|D| = O(n)$, we can perform in $O(n + m)$ time the computation of R_{k_i} for $i = 1, \ldots, n$ by executing the bit-shift and the bitwise logical operations. However, we have to examine whether $m \in R_j$ for every $j = 1, 2, \ldots, N$. For a complete simulation of the move of the Shift-And algorithm, we need a mechanism for enumerating the set $\mathit{Output}(R_{k_i}, u_{i+1})$ defined as follows.

Definition 4. *For $S \subseteq \{1, \ldots, m\}$ and $u \in D$, let*

$$\mathit{Output}(S, u) = \big\{1 \le i \le |u| \;\big|\; m \in \widehat{f}(S, u[1:i])\big\}.$$

To realize the procedure enumerating the set *Output*, we define the following sets.

Definition 5. *For any $u \in D$, let*

$$U(u) = \big\{1 \le i \le |u| \;\big|\; i < m \textit{ and } m \in \widehat{M}(u[1:i])\big\}, \textit{ and}$$
$$V(u) = \big\{1 \le i \le |u| \;\big|\; i \ge m \textit{ and } m \in \widehat{M}(u[1:i])\big\}.$$

Then, we have the following lemma.

Lemma 4. *For any $S \subseteq \{1, \dots, m\}$ and any $u \in \Sigma^*$,*

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

Proof. By Lemma 2 and Definitions 4 and 5, we obtain:

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

■

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

Lemma 5. *The function which takes as input $u \in D$ and returns in $O(1)$ time the m-bit representation of $U(u)$, can be realized in $O(|D|+m)$ time using $O(|D|)$ 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 3. ■

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 the set, we repeatedly use the logarithm operation to find the leftmost bit of the integer that is one. Assuming that the logarithm operation can be performed in constant time, this enumeration takes only linear time proportional to the set size.

Next, we consider $V(u)$. Since the set $V(u)$ cannot be represented as an m-bit integer, we shall represent it as a linked list as shown in the proof of the next lemma.

Lemma 6. *The procedure which takes as input $u \in D$ and enumerates the set $V(u)$, can be realized in $O(|D| + m)$ time using $O(|D|)$ space, so that it runs in linear time with respect to $|V(u)|$.*

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

$$V(u) = V(u') \cup \{|u| \mid m \leq |u| \text{ and } m \in \widehat{M}(u)\}.$$

We use the function $Prev(u)$ that returns the node of the dictionary trie D that represents the longest proper prefix v of u such that $|v| \in V(u)$. Then, we have

$$V(u) = V(Prev(u)) \cup \{|u| \mid m \leq |u| \text{ and } m \in \widehat{M}(u)\}.$$

The function $Prev(u)$ can be realized to answer in $O(1)$ time, using $O(|D|)$ time and space. Therefore it is sufficient to store in every node u of the dictionary trie D the value $Prev(u)$ and the boolean value $in\_V(u)$ indicating whether $|u| \in V(u)$. The proof is now complete. ■

From Lemmas 4, 5, and 6, we have the following theorem.

Theorem 2. *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 linear time with respect to $|Output(S, u)|$.*

Now we can simulate the behavior of the Shift-And algorithm on an uncompressed text completely. The algorithm is summarized as in Figure 4. The behavior of the new algorithm is illustrated in Figure 5.

Theorem 3. *The algorithm of Figure 4 runs in $O(|\Sigma| + m + n + r)$ time using $O(|\Sigma| + m + n)$ space, where r is the number of pattern occurrences.*

4 Experimental Results

In order to estimate the performance of the proposed algorithm, we carried out some experiments on the following four methods.

Method 1. A decompression followed by the Shift-And algorithm.
Method 2. Our previous algorithm presented in [13].
Method 3. The new algorithm proposed in this paper.
Method 4. Searching the uncompressed text, using the Shift-And algorithm.

In our experiments we used the Brown corpus as the text to be searched. The uncompressed size is about 6.8Mb and the compressed size is about 3.4Mb. The experiments were performed in the following two different situations.

Situation 1. Workstation (SPARCstation 20) with remote disk storage. The file transfer ratio is 0.96 Mbyte/sec.
Situation 2. Workstation (SPARCstation 20) with local disk storage. The file transfer ratio is 3.27 Mbyte/sec.

```
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
    /* We represent the set $V(u)$ by the functions $Prev(u)$ and $in\_V(u)$.
       See the proof of Lemma 6. */

    /* Preprocessing */
    Construct the table $M$ from $\mathcal{P}$;
    $D := \emptyset$;   $U'(\varepsilon) := \emptyset$;   $in\_V(\varepsilon) := false$;   $Prev(\varepsilon) := \varepsilon$;
    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 $p \in \big(R \cap U'(u_\ell)\big) \cup V(u_\ell)$ do
            report a pattern occurrence at position $k + p - m + 1$;
        $R := \big((R \oplus |u_\ell|) \cup \{1, 2, \ldots, |u_\ell|\}\big) \cap \widehat{M}(u_\ell)$;
        $k := k + |u_\ell|$
    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 $m \in \widehat{M}(u \cdot a)$ then
            $in\_V(u \cdot a) := true$
        else
            $in\_V(u \cdot a) := false$;
        if $in\_V(u) = true$ then
            $Prev(u \cdot a) := u$
        else
            $Prev(u \cdot a) := Prev(u)$
    end
end;
```

Fig. 4. Pattern matching algorithm in LZW compressed text

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

Fig. 5. Behavior of the algorithm.

Table 1. CPU time and elapsed time.

method	CPU time (sec)	elapsed time (sec) Situation 1	Situation 2
Method 1	7.52	8.16	7.62
Method 2	6.57	7.31	6.83
Method 3	5.15	6.05	5.41
Method 4	3.09	9.36	3.25

The searching times, measured in both the CPU time and the elapsed time, are shown in Table 1, where we included the preprocessing time.

Although the time complexities of our algorithms are linear with respect to the compressed text size n not to the original size N, the LZW compression of typical English texts normally gives $n = N/2$ and thus the constant factor is crucial. It is observed from Table 1 that, in the CPU time comparison, our algorithms (Methods 2 and 3) are slower than the uncompressed case (Method 4) whereas they are faster than a decompression followed by a search (Method 1). It is also observed that the new algorithm (Method 3) is about 1.3 times faster than the previous one (Method 2).

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. In fact, even a decompression followed by a simple search (Method 1) was faster than the uncompressed search (Method 4) in Situation 1. It should be noted that, in this situation, the previous algorithm (Method 2) and the new algorithm (Method 3) are faster than the uncompressed case (Method 4), and especially the latter is approximately 1.5 times faster than the uncompressed case.

On the contrary, in the situations that the data transfer is ralatively fast, the CPU time becomes a dominant factor. It is observed that, like in the CPU time comparison, Methods 2 and 3 are slower than Method 4 while they are faster than Method 1 in the elapsed time comparison in Situation 2.

Thus we conclude that, for the LZW compression, the compressed search is indeed faster than a decompression followed by a fast search, and that the Shift-And approach is effective in the LZW compressed pattern matching. When the data transfer is slow, e.g. network environments, the compressed search can be faster than the uncompressed search.

5 Extensions

In this section, we mention how to extend our algorithm.

5.1 Generalized Pattern Matching

The generalized pattern matching problem [1] is a pattern matching problem in which a pattern element is a set of characters. For instance, $(\mathtt{b}+\mathtt{c}+\mathtt{h}+\mathtt{l})\mathtt{ook}$ is a pattern that matches the strings `book`, `cook`, `hook`, and `look`. Formally, let $\Delta = \{X \subseteq \Sigma \mid X \neq \emptyset\}$ and $\mathcal{P} = X_1...X_m \quad (X_i \in \Delta)$. Then we want to find all integers i such that $\mathcal{T}[i : i+m-1] \in \mathcal{P}$.

It is not difficult to extend our algorithm to the problem. We have only to modify some equations: For example, we modify Equations (1) and (2) in Section 2.2 as follows.

$$R_k = \big\{1 \le i \le m \;\big|\; \mathcal{P}[1:i] \ni \mathcal{T}[k-i+1:k]\big\}, \tag{1$'$}$$

$$M(a) = \big\{1 \le i \le m \;\big|\; \mathcal{P}[i] \ni a\big\}. \tag{2$'$}$$

5.2 Pattern Matching with k Mismatches

This problem is a pattern matching problem in which we allow up to k characters of the pattern to mismatch with the corresponding text [10]. For example, if $k = 2$, the pattern `pattern` matches the strings `postern` and `cittern`, but does not match `eastern`. The idea stated in [7] to solve this problem is to count up the number of mismatches using $\lceil m \log_2 m \rceil$ bits instead of using one bit to see whether $\mathcal{P}[i] = \mathcal{T}[k]$. This technique can be used to adapt our algorithm for the problem.

5.3 Multiple Pattern Matching

Suppose we are looking for multiple patterns in a text. One solution is to keep one bit vector R per pattern and perform the Shift-And algorithm in parallel, but the time complexity is linearly proportional to the number of patterns. The solutions in [7] and in [17] are to coalesce all vectors, keeping all the information in only one vector. Such technique can be used to adapt our algorithm for the multiple pattern matching problem in LZW compressed text.

6 Conclusion

In this paper we addressed the problem of searching in LZW compressed text directly, and presented a new algorithm. We implemented the algorithm, and showed that it is approximately 1.5 times faster than a decompression followed by a search using the Shift-And algorithm. Moreover we showed that our algorithm has several extensions, and is therefore useful in many practical applications. Some future directions of this study will be extensions to the pattern matching with k differences, and to the regular expression matching, and will be to develop a compression method which enables us to scan compressed texts faster.

References

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

[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. Optimal two-dimensional compressed matching. In *Proc. 21st International Colloquium on Automata, Languages and Programming*, 1994.

[5] 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.

[6] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms*, 13(1):2–32, 1992.

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

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

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

[10] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4:33–72, 1988.

[11] 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.

[12] 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.

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

[14] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc. 5th Annu. Symp. Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 113–124. Springer-Verlag, 1994.

[15] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. 8th Annu. Symp. Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1997.

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

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

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