Shift-And Approach to Pattern Matching in LZW Compressed Text

Kida, Takuya
Department of Informatics, Kyushu University

Takeda, Masayuki
Department of Informatics, Kyushu University

Shinohara, Ayumi
Department of Informatics, Kyushu University

Arikawa, Setsuo
Department of Informatics, Kyushu University

https://hdl.handle.net/2324/3023

出版情報：DOI Technical Report. 156, 1999-01. Department of Informatics, Kyushu University
バージョン：

DOI-TR-156

DOI Technical Report

Shift-And Approach to Pattern Matching in LZW Compressed Text

by

Takuya Kida, Masayuki Takeda, Ayumi Shinohara
and Setsuo Arikawa

January 1999

Department of Informatics
Kyushu University
Fukuoka 812-81, Japan

Email: kida@i.kyushu-u.ac.jp Phone: +81-92-642-2697

Shift-And Approach to Pattern Matching in LZW Compressed Text

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

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 the 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 the 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 the 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 notations. Let Σ, usually called an *alphabet*, be a finite set of characters, and Σ^* be the 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 $u[i]u[i+1]...u[j]$, $0 \leq i \leq j \leq |u|$.

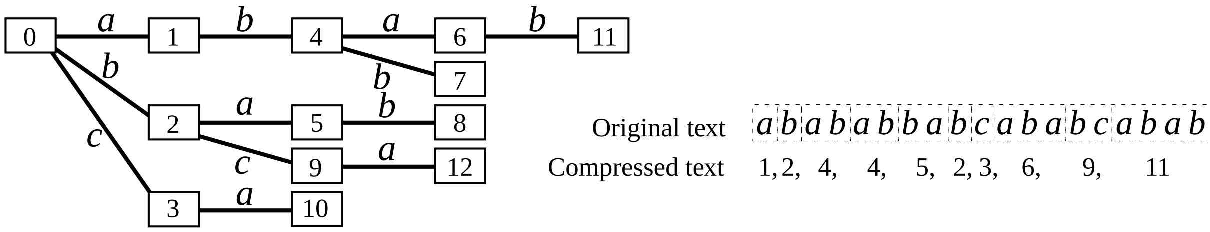

Figure 1: Dictionary trie.

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

2.1 LZW compression

The LZW compression is a very popular compression method, and is adopted as the `compress` command of UNIX, for instance. Formally, an LZW compressed text is a sequence of integers. Each integer j indicates the node numbered j of 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 for the text *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 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 notations 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 $k = 1, 2, \ldots, N$, let

$$R_k = \left\{1 \le i \le m \;\middle|\; i \le k \text{ and } \mathcal{P}[1:i] = \mathcal{T}[k-i+1:k]\right\}, \tag{1}$$

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 $u_{i+1} \le |D|$ **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.

Figure 2: Construction of dictionary trie.

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

$$M(a) = \left\{1 \le i \le m \;\middle|\; \mathcal{P}[i] = a\right\}. \tag{2}$$

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

$$f(S, a) = \big((S \oplus 1) \cup \{1\}\big) \cap M(a), \qquad (S \subseteq \{1, 2, \cdots, m\},\ a \in \Sigma)$$

where $\oplus$ *is defined as* $S \oplus p = \left\{i + p \;\middle|\; i \in S\right\}$ *in which* p *is an integer.*

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

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

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 $m \le 32$, 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. $R_{k+1} = ((R_k \ll 1) + 1)\ \&\ M(\mathcal{T}[k+1]) \qquad (k \ge 0)$,

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

original text:			a	b	a	b	a	b	b	a	b	c	a	b	a	b	c	a	b	a	b
	a	0	1	0	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0	1	0
	b	0	0	1	0	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0	1
R_k:	a	0 →	0 →	0 →	1 →	0 →	1 →	0 →	0 →	0 →	0 →	0 →	0 →	0 →	1 →	0 →	0 →	0 →	0 →	1 →	0
	b	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1
	c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	<u>1</u>	0	0	0	0
																	△				

Figure 3: Behavior of the Shift-And algorithm.
The symbol △ 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 behaviors of the Shift-And algorithm on the original text. Assume that the compressed text is $u_1u_2\dots u_n$ where u_i is a string in the dictionary trie represented as an integer. Let $k_i = |u_1u_2\dots u_i|$ for $i = 0, 1, \dots, n$. The idea is to compute only the values of R_{k_i} for $i = 1, 2, \dots, n$, 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 $\hat{f}$ be the function f extended to $2^{\{1,\dots,m\}} \times \Sigma^*$ by*

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

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, we have*

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

Proof. It follows directly from the definition of $\hat{f}$. □

Using the function $\hat{f}$ defined on $2^{\{1,\dots,m\}} \times D$, we compute the value of $R_{k_{i+1}} = \hat{f}(R_{k_i}, u_{i+1})$ from R_{k_i} and u_{i+1} for each $i = 0, 1, \dots, n-1$. 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 $\hat{M}$ defined as follows.

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

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

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

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,

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

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,

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

□

Lemma 3 *Function which takes as input* $u \in D$ *and returns in* $O(1)$ *time the* m*-bit representation of the set* $\hat{M}(u)$, *can be realized in* $O(|D|+m)$ *time using* $O(|D|)$ *space.*

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

Now we have the following theorem.

Theorem 1 *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* $\hat{f}(S, u)$, *can be realized in* $O(m + |D|)$ *time using* $O(|D|)$ *space.*

Since $O(|D|) = O(n)$, we can perform in $O(n)$ 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 elements of the set $\mathit{Output}(R_{k_i}, u_{i+1})$ defined as below.

Definition 4 *For any* x, u, y *in* Σ^* *such that* $\mathcal{T} = xuy$ *and* $u \neq \varepsilon$, *let*

$$\mathit{Output}(R_{|x|}, u) = \big\{1 \leq j \leq |u| \;\big|\; m \in R_{|x|+j}\big\}.$$

To realize the function *Output*, we define the following sets.

Definition 5 *For any $u \in \Sigma^*$, let*

$$\begin{aligned} U(u) &= \big\{1 \le i \le |u| \bigm| i < m \text{ and } u[1:i] = \mathcal{P}[m-i+1:m]\big\}, \\ A(u) &= \big\{1 \le i \le |u| \bigm| m \le i \text{ and } u[i-m+1:i] = \mathcal{P}\big\}. \end{aligned}$$

For a set S of integers, let $m \ominus S = \big\{m - i \bigm| i \in S\big\}$. Then, we have the following lemma.

Lemma 4 *Suppose that the text is $\mathcal{T} = xuy$ with $x, u, y \in \Sigma^*$ and $u \neq \varepsilon$. Then, we have*

$$Output(R_{|x|}, u) = \big((m \ominus R_{|x|}) \cap U(u)\big) \cup A(u).$$

Proof. If $i \in U(u)$ and $m - i \in R_{|x|}$, then there is a pattern occurrence at position $|x| + i - m + 1$ of $\mathcal{T}$. If $i \in A(u)$, then there is a pattern occurrence at position $|x| + i - m + 1$ of $\mathcal{T}$. Based on these observations, we have:

$$\begin{aligned} Output(R_{|x|}, u) &= \{1 \le i \le |u| \mid 0 \le |x| + i - m \text{ and } \mathcal{T}[|x| + i - m + 1 : |x| + i] = \mathcal{P}\} \\ &= \{1 \le i \le |u| \mid (i \in U(u) \text{ and } m - i \in R_{|x|}) \text{ or } i \in A(u)\} \\ &= \big((m \ominus R_{|x|}) \cap U(u)\big) \cup A(u). \end{aligned}$$

□

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

Lemma 5 *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 $O(m + |D|)$ time using $O(|D|)$ space.*

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

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

Then, we can prove the lemma in a similar way to the proof of Lemma 3. □

To eliminate the cost of performing the operation $\ominus$, we store the set $U'(u) = m \ominus U(u)$ instead of $U(u)$. Then, we can obtain the value of $R_{|x|} \cap U'(u)$ by one execution of the bitwise logical product operation. To enumerate the elements of the set represented as an m-bit integer, 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 $A(u)$. Since $A(u) \subseteq \{1, 2, \ldots, |u|\}$ and $|u|$ may be greater than m, it cannot be represented as an m-bit integer. The set $A(u)$ is represented as a linked list as shown in the proof of the next lemma.

Lemma 6 *Function which takes as input $u \in D$ and returns in $O(1)$ time a linear size representation of $A(u)$, can be realized in $O(m + |D|)$ time using $O(|D|)$ space.*

Proof. By the definitions of $A(u)$ and $\hat{M}(u)$, for any $u = u'a$ with $u' \in \Sigma^*$ and $a \in \Sigma$,

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

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

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

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

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

Theorem 2 *The function Output can be realized to enumerate the elements of the answer in linear time proportional to the number of the elements in $O(m + n)$ time using $O(n)$ space.*

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

4 Experimental results

Compressed pattern matching has two important goals from the practical viewpoints.

Goal 1. A faster search in compressed text in comparison with a decompression followed by a simple search.

Goal 2. A faster search in compressed text in comparison with a simple search in uncompressed text, namely, to speed up pattern matching by text compression.

Though it seems that many studies on the problem aimed Goal 1, Goal 2 is also worthwhile to considering since most of the time required for pattern matching is consumed in data transmission of text from local or remote disk devices [14]. To see whether the proposed algorithm achieves these goals, we implemented the following four methods in the C++ language on Sun SPARCstation 20.

Method 1. A decompression followed by the Shift-And algorithm.

Input. An LZW compressed text $u_1u_2...u_n$ and a pattern $\mathcal{P}$.
Output. All positions at which $\mathcal{P}$ occurs.

```
begin
    /* Preprocessing */
    Construct the table M from P;
    D := ∅;   U'(ε) := ∅;   A(ε) := ∅;   Prev(ε) := ε;
    for each a ∈ Σ do call Update(ε, a);
    /* Text scanning phase */
    k := 0;   R := ∅;   u_0 = ε;
    for ℓ := 1 to n do begin
        call Update(u_{ℓ-1}, u_ℓ);
        for each p ∈ (R ∩ U'(u_ℓ)) ∪ A(u_ℓ) do
            report a pattern occurrence at position k + p − m + 1;
        R := ((R ⊕ |u_ℓ|) ∪ {1, 2, ..., |u_ℓ|}) ∩ M̂(u_ℓ);
        k := k + |u_ℓ|
    end
end.

procedure Update(u, v)
begin
    if v ≤ |D| then
        let a be the first character of v
    else
        let a be the first character of u;
    D := D ∪ {u · a};      /* New node u · a is added to D. */
    M̂(u · a) := ((M̂(u) ⊕ 1) ∪ 1) ∩ M(a);
    if |u · a| < m then
        if m ∈ M̂(u · a) then
            U'(u · a) := U'(u) ∪ {m − |u · a|}
        else
            U'(u · a) := U'(u)
    else begin
        U'(u · a) := ∅;
        if m ∈ M̂(u · a) then
            in_A(u · a) := true     /* in_A indicates |u| ∈ A(u) */
        else
            in_A(u · a) := false;
        if in_A(u) = true then
            Prev(u · a) := u
        else
            Prev(u · a) := Prev(u)
    end
end;
```

Figure 4: Pattern matching algorithm in LZW compressed text

original text:			a	b	ab	ab	ba	b	c	aba	bc	abab
compressed text:			1	2	4	4	5	2	3	6	9	11
	a	0	1	0	0	0	1	0	0	1	0	0
	b	0	0	1	1	1	0	1	0	0	0	1
R_k:	*a*	0	0	0	0	0	0	0	0	1	0	0
	b	0	0	0	1	1	0	0	0	0	0	1
	c	0	0	0	0	0	0	0	0	0	1	0
$Output(R_k, u_\ell)$:			$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{2\}$	$\emptyset$

Figure 5: Behaviors of algorithm.

Table 1: Time complexity.

method	preprocessing of pattern	running time		
		data transmission	decompression	text scanning
Method 1	$O(m + \|\Sigma\|)$	$O(N)$	$O(N)$	$O(N)$
Method 2	$O(m^2 + m\|\Sigma\|)$	$O(n)$	-	$O(n)$
Method 3	$O(m + \|\Sigma\|)$	$O(n)$	-	$O(n)$
Method 4	$O(m + \|\Sigma\|)$	$O(N)$	-	$O(N)$

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.

Table 1 illustrates the time complexities of the methods. Ignoring the preprocessing time, the running time of Methods 1 and 4 is $O(N)$ while that of Methods 2 and 3 is $O(n)$, where N is the original text length and n is the compressed text length. Although the best LZW compression gives $n = \sqrt{2N}$, 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 both the CPU time and the elapsed time comparisons.

In the experiment, 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. We measured the CPU time and the elapsed time for 30 patterns, and calculated the average of them. The results are shown in Table 2. We excluded the preprocessing time because we can ignore it when the pattern length m is sufficiently smaller than the text length N.

Comparison with Method 1 implies that both Methods 2 and 3 achieve Goal 1. Comparison with Method 4 also implies that both Methods 2 and 3 achieve Goal 2. That is, our previous and new algorithms achieve both Goals 1 and 2. Moreover, the new algorithm is faster than the previous one: in the CPU time comparison the new one is approximately 1.3 times faster than the previous one. We therefore conclude that 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

Table 2: CPU time and elapsed time.

method	elapsed time (s)	CPU time (s)
Method 1	6.554	3.373
Method 2	6.130	2.914
Method 3	5.400	2.241
Method 4	11.425	0.699

matching.

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 $\mathcal{P} = X_1 \ldots X_m \quad (X_i \in \Delta)$. 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.

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

$$M(a) = \left\{1 \leq i \leq m \;\middle|\; \mathcal{P}[i] \ni a\right\}. \tag{2'}$$

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 1 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. *Communications of the 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, *Proceedings 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. 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. 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. *Communications of the ACM*, 35(10):83–91, October 1992.

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