A Unifying Framework for Compressed Pattern Matching

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

Abstract

We introduce a general framework which is suitable to capture an essence of compressed pattern matching according to various dictionary based compressions, and propose a compressed pattern matching algorithm for the framework. The goal is to find all occurrences of a pattern in a text without decompression, which is one of the most active topics in string matching. Our framework includes such compression methods as Lempel-Ziv family, (LZ77, LZSS, LZ78, LZW), byte-pair encoding, and the static dictionary based method. Technically, our pattern matching algorithm extends that for LZW compressed text presented by Amir, Benson and Farach.

1 Introduction

Pattern matching is one of the most fundamental operations in string processing. The problem is to find all occurrences of a given pattern in a given text. A lot of classical or advanced pattern matching algorithms have been proposed (see [3, 2]). Data compression is another most important research topic, whose aim is to reduce its space usage. Considerable amount of compression methods have been proposed (see [15]).

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. Various compressed pattern matching algorithms have been proposed depending on underlying compression methods. Among them, we focus on the following works. Amir et al.[1] introduced an elegant compressed pattern matching algorithm for Lempel-Ziv-Welch (LZW) compression which runs in $O(n + m^2)$ time, where n is the length of the compressed text and m is the length of the pattern. (They considered finding only the first occurrence of the pattern). The basic idea is to simulate the move of the Knuth-Morris-Pratt (KMP) automaton [3] on the compressed text directly. In [11] we have extended it in order to find *all* occurrences of *multiple* patterns simultaneously, by simulating the move of the Aho-Corasick automaton [3]. The running time is $O(n + m^2 + r)$, where m is the total length of the patterns and r is the number of pattern occurrences. We implemented a simple version of the algorithm and observed that it is approximately twice faster than a decompression followed by a search using the Aho-Corasick automaton. We took another implementation of the algorithm utilizing *bit-parallelism*, and reported some experiments [10]. Independently, Navarro and Raffinot [14] developed a more general technique for string matching on a text given as a sequence of blocks, which abstracts both LZ77 and LZ78 compressions, and gave bit-parallel implementations. The running time of these algorithms based on the bit-parallelism for LZW is $O(nm/w + m + r)$, where w is the length in bits of the machine word. If the pattern is short ($m < w$), these algorithms are efficient in practice. Moura et al. [4, 5] proposed practical algorithms. They presented a new compression scheme which uses a word-based Huffman encoding with a byte-oriented code. In recent papers, we developed compressed pattern matching algorithms for compressed text using anti-dictionaries [17], and for compressed text using byte-pair encoding [16]. Especially, the latter was showed to be even faster than pattern matching in uncompressed texts.

In this paper, we introduce a *collage system*, that is a formal system to represent a string by a pair of dictionary $\mathcal{D}$ and sequence $\mathcal{S}$ of phrases in $\mathcal{D}$. The basic operations are concatenation, truncation, and repetition. Collage systems give us a unifying framework of various dictionary-based compression method, such as Lempel-Ziv family (LZ77, LZSS, LZ78, LZW), byte-pair encoding, and the static dictionary based method. Most of these compressed text can be transformed in linear time into a corresponding collage system which contains no truncation. Exceptions are LZ77 and LZSS, where the size grows $O(n \log n)$ and truncation operations are required. We remark that a straight-line program [9, 13] is a collage system containing concatenation

only, and a composition system introduced in [7] is also a collage system which allows concatenation and truncation.

We develop a compressed pattern matching algorithm for collage systems which contain no truncation, whose running time is $O(\|\mathcal{D}\| + |\mathcal{S}| + m^2 + r)$ using $O(\|\mathcal{D}\| + m^2)$ space, where $\|\mathcal{D}\|$ denotes the size of the dictionary $\mathcal{D}$ and $|\mathcal{S}|$ is the length of the sequence $\mathcal{S}$. For the case of LZW compression, it matches the bound $O(n + m^2 + r)$ in [11]. For general collage systems, which contain truncation, we show a compressed pattern matching algorithm which runs in $O((\|\mathcal{D}\| + |\mathcal{S}|) \cdot \mathit{height}(\mathcal{D}) + m^2 + r)$ time with $O(\|\mathcal{D}\| + m^2)$ space, where $\mathit{height}(\mathcal{D})$ denotes the maximum dependency of the operations in $\mathcal{D}$. These results show that the truncation slows down the compressed pattern matching to the factor $\mathit{height}(\mathcal{D})$.

2 Preliminaries

Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string $u = xyz$, respectively. A prefix, factor, and suffix of a string u is said to be *proper* if it is not u. The length of a string u is denoted by $|u|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. The ith symbol of a string u is denoted by $u[i]$ for $1 \leq i \leq |u|$, and the factor of a string u that begins at position i and ends at position j is denoted by $u[i : j]$ for $1 \leq i \leq j \leq |u|$. The reversed string of a string u is denoted by u^R.

Let u be a string in Σ^*, and let i be a non-negative integer. Denote by ${}^{[i]}u$ (resp. $u^{[i]}$) the string obtained by removing the length i prefix (resp. suffix) from u. For a set A of integers and an integer k, let $A \oplus k = \{i + k \mid i \in A\}$ and $A \ominus k = \{i - k \mid i \in A\}$.

For strings x and y, denote by $\mathit{Occ}(x, y)$ the set of occurrences of x in y. That is, $\mathit{Occ}(x, y) = \big\{|x| \leq i \leq |y| \;\big|\; x = y[i - |x| + 1 : i]\big\}$. The next lemma follows from the periodicity lemma.

Lemma 1 *If $\mathit{Occ}(x, y)$ has more than two elements and the difference of the maximum and the minimum elements is at most $|x|$, then it forms an arithmetic progression, in which the step is the smallest period of x.*

3 Collage system and text compressions

Text compression methods can be viewed as mechanisms to factorize a text into a series of *phrases* $T = u_1 u_2 \dots u_n$ and to store a sequence of 'representations' of phrases u_i. The set of phrases is called *dictionary*. In this section we introduce a collage system as a general framework for dictionary-based text compressions, and show that most of such compression methods can be directly translated into collage systems.

A *collage system* is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ defined as follows: $\mathcal{D}$ is a sequence of assignments $X_1 = \mathit{expr}_1;\ X_2 = \mathit{expr}_2;\ \cdots;\ X_n = \mathit{expr}_n$, where each X_k is a variable and expr_k is any of the form

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

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

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

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

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

We now translate various compression methods into corresponding collage systems. For notational convenience, we allow abbreviations by composing multiple assignments into one in the sequel.

LZW compression. [19] $\mathcal{S} = X_{i_1}, X_{i_2}, \dots, X_{i_n}$ and $\mathcal{D}$ is as follows:

$$
\begin{aligned}
&X_1 = a_1; \quad X_2 = a_2; \quad \cdots; \quad X_q = a_q;\\
&X_{q+1} = X_{i_1} b_{i_2}; \quad X_{q+2} = X_{i_2} b_{i_3}; \quad \cdots;\\
&X_{q+n-1} = X_{i_{n-1}} b_{i_n},
\end{aligned}
$$

where the alphabet is $\Sigma = \{a_1, \dots, a_q\}$, and b_j denotes the first symbol of the string X_j. $\mathcal{S}$ is encoded as a sequence of integers $i_1, i_2, \dots, i_n$ in which an integer i_j is represented

in $\lceil \log_2(q+j) \rceil$ bits, while $\mathcal{D}$ is not encoded since it can be obtained from $\mathcal{S}$.

LZ78 compression. [20] $\mathcal{S} = X_1, X_2, \ldots, X_n$, and $\mathcal{D}$ is as follows:

$$X_0 = \varepsilon;\ X_1 = X_{i_1} b_1;\ X_2 = X_{i_2} b_2;\ \cdots;\ X_n = X_{i_n} b_n,$$

where b_j is a symbol in Σ. While no need to encode $\mathcal{S}$, the dictionary $\mathcal{D}$ is encoded as a sequence in which integer i_k and character b_k appear alternately. Note that LZW is a simplification of LZ78.

We will turn our attention to LZ77 and its variant. Although we have no direct representations for LZ77, we can convert in $O(n \log n)$ time a compressed text of size n encoded by LZ77 into a collage system with $\|\mathcal{D}\| = O(n \log n)$ [7]. Below we give a translation of the LZSS compression method which is a simplified variant of LZ77. The differences between LZSS and LZ77 are essentially the same as those between LZW and LZ78.

LZSS compression. [18] $\mathcal{S} = X_{q+1}, X_{q+2}, \ldots, X_{q+n}$, and $\mathcal{D}$ is as follows:

$$\begin{aligned}
&X_1 = a_1; \quad X_2 = a_2; \quad \cdots; X_q = a_q;\\
&X_{q+1} = \left(\left({}^{[i_1]}X_{\ell(1)} X_{\ell(1)+1} \cdots X_{r(1)} \right)^{m_1} \right)^{[j_1]} b_1;\\
&\qquad \vdots\\
&X_{q+n} = \left(\left({}^{[i_n]}X_{\ell(n)} X_{\ell(n)+1} \cdots X_{r(n)} \right)^{m_n} \right)^{[j_n]} b_n;
\end{aligned}$$

where $0 \le i_k, j_k, m_k$ and $b_k \in \Sigma$.

Byte pair encoding. [6] $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$, and $\mathcal{D}$ is as follows:

$$\begin{aligned}
&X_1 = a_1; \quad X_2 = a_2; \quad \cdots; \quad X_q = a_q;\\
&X_{q+1} = X_{\ell(1)} X_{r(1)}; \quad X_{q+2} = X_{\ell(2)} X_{r(2)}; \cdots;\\
&X_{q+s} = X_{\ell(s)} X_{r(s)},
\end{aligned}$$

where $a_1, \cdots, a_q$ are the characters which appear in the text, and $q+s \le 256$. $\mathcal{S}$ is encoded as a byte sequence. It is a desirable property from the practical viewpoint of compressed pattern matching. $\mathcal{D}$ is encoded in a simple way.

Static dictionary based methods. $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$, and $\mathcal{D}$ is as follows:

$$\begin{aligned}
&X_1 = a_1; \quad X_2 = a_2; \quad \cdots; \quad X_q = a_q;\\
&X_{q+1} = w_1; \quad X_{q+2} = w_2; \quad \cdots; \quad X_{q+s} = w_s,
\end{aligned}$$

where w_k is a string in Σ^+ with $|w_k| > 1$. $\mathcal{S}$ is encoded in various ways, such as the Huffman coding. Note that, when $s = 0$ the compression methods of this type are called *character-based compression* and the compression ratio depends only on how to encode $\mathcal{S}$. On the other hand, the strings $w_1, w_2, \ldots, w_s$ in $\mathcal{D}$ are considered to be frequent in many texts in common. It is often stored independently of the compressed texts.

We emphasis that the truncation operation is only used in LZSS (and LZ77) in the above, and that the repetition operation is used in order to express the *self-reference* in LZSS (and LZ77). By using the repetition operation, we can express the run-length encoding in an obvious way.

4 Main result

Amir, Benson, and Farach presented in [1] a series of algorithms with various time and space complexities for LZW compressed text. From the viewpoint of speeding up of the pattern matching, the most attractive one is the $O(n + m^2)$ time and space algorithm, where n is the compressed text length and m is the pattern length. It essentially simulates the move of the KMP automaton. The simulation utilizes the fact that in the LZW compression a phrase newly added to dictionary is restricted to a concatenation of an existing phrase and a single character. The main contribution of this paper is a generalization of their idea to the collage systems, in which concatenation of two phrases, k times repetition of a phrase, and prefix and suffix truncations of a phrase are allowed.

One possible approach is to use the bit-parallelism, as in the recent work by Navarro and Raffinot [14], which deals with compressed pattern matching for the Lempel-Ziv family. Although this approach is in fact efficient when $m < w$, where w is the machine word length in bits, we take in this paper another approach in order to deal with general case. Moreover, our approach can be extended to multiple pattern matching, if we allow only the concatenation operations.

Consider how to simulate the move of the KMP automaton for a pattern P running on the uncompressed text T. Let $\delta_{\mathrm{KMP}} : \{0, 1, \ldots, m\} \times \Sigma \to \{0, 1, \ldots, m\}$ be the state transition function of the KMP automaton for $P = P[1 : m]$. We extend δ_{KMP} to the domain $\{0, 1, \ldots, m\} \times \Sigma^*$ in the standard manner. That is,

$$\delta_{\mathrm{KMP}}(j, \varepsilon) = j, \ \ \delta_{\mathrm{KMP}}(j, ua) = \delta_{\mathrm{KMP}}(\delta_{\mathrm{KMP}}(j, u), a),$$

where $j \in \{0, 1, \ldots, m\}$, $u \in \Sigma^*$ and $a \in \Sigma$. Let D be the set of phrases defined by $\mathcal{D}$. Define the function *Jump* : $\{0, 1, \ldots, m\} \times D \to \{0, 1, \ldots, m\}$ by

$$\mathit{Jump}(j, u) = \delta_{\mathrm{KMP}}(j, u).$$

We also define the set $Output(j, u)$ for any pair $\langle j, u \rangle$ in $\{0, 1, \ldots, m\} \times D$ by

$$\begin{aligned}
&\mathit{Output}(j, u)\\
&= \left\{ 1 \le i \le |u| \;\middle|\; P \text{ is a suffix of string } P[1 : j] \cdot u[1 : i] \right\}.
\end{aligned}$$

Input. A collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ and a pattern $P = P[1:m]$.
Output. All positions at which P occurs in T.

```
/* Preprocessing */
Perform the processing required for Jump and Output (See Section 5);
/* Text scanning */
let S = X_{i_1} X_{i_2} ... X_{i_n}.
ℓ := 0;
state := 0;
for k := 1 to n do begin
    for each p ∈ Output(state, X_{i_k}) do
        Report a pattern occurrence that ends at position ℓ + p ;
    state = Jump(state, X_{i_k});
    ℓ := ℓ + |X_{i_k}|
end
```

Figure 1. Pattern matching algorithm.

Our algorithm, given a pattern P and an encoding of a collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ representing a text T, processes the sequence $\mathcal{S}$ variable by variable (i.e. phrase by phrase) to report all occurrences of P within T. Thus we need to realize

- the function *Jump*(j, X), and
- the procedure which enumerates the set *Output*(j, X),

both take as input any pair of an integer $j \in \{0, 1, \ldots, m\}$ and a variable X defined in $\mathcal{D}$. An overview of the algorithm based on the function and the procedure is shown in Fig. 1.

In some cases, such as static dictionary based methods, $\mathcal{D}$ is followed by $\mathcal{S}$ in the encoding, or $\mathcal{D}$ is given independently of $\mathcal{S}$. Thus we can process $\mathcal{D}$ as a preprocessing. In other cases, such as adaptive dictionary based methods like the Lempel-Ziv family, $\mathcal{D}$ is not given explicitly in the encoding of $\langle \mathcal{D}, \mathcal{S} \rangle$, and will be rebuilt incrementally in the variable-by-variable processing of $\mathcal{S}$. From the theoretical viewpoint, we can process $\mathcal{D}$ being incrementally reconstructed from $\mathcal{S}$ in first step, and then process $\mathcal{S}$ again in second step, without changing the time complexity. In practice, we merge these steps into one.

We have the following theorems which will be proved in the next section.

Theorem 1 *The function Jump(j, X) can be realized in $O(\|\mathcal{D}\| \cdot$ height$(\mathcal{D}) + m^2)$ time using $O(\|\mathcal{D}\| + m^2)$ space, so that it answers in $O(1)$ time. If $\mathcal{D}$ contains no truncation, the time complexity becomes $O(\|\mathcal{D}\| + m^2)$.*

Theorem 2 *The procedure to enumerate the set Output (j, X) can be realized in $O(\|\mathcal{D}\| \cdot$ height$(\mathcal{D}) + m^2)$ time using $O(\|\mathcal{D}\| + m^2)$ space, so that it runs in $O($height$(X) + \ell)$ time, where ℓ is the size of the set Output(j, X). If $\mathcal{D}$ contains no truncation, it can be realized in $O(\|\mathcal{D}\| + m^2)$ time and space, so that it runs in $O(\ell)$ time.*

Thus we have the following result.

Theorem 3 *The problem of compressed pattern matching can be solved in $O\big((\|\mathcal{D}\| + |\mathcal{S}|) \cdot$ height$(\mathcal{D}) + m^2 + r\big)$ time using $O(\|\mathcal{D}\| + m^2)$ space. If $\mathcal{D}$ contains no truncation, it can be solved in $O(\|\mathcal{D}\| + |\mathcal{S}| + m^2 + r)$ time.*

In our framework, we can consider that the compressed text length n is $\|\mathcal{D}\| + |\mathcal{S}|$, therefore the time and the space complexities in the case of no truncation become $O(n + m^2 + r)$ and $O(n + m^2)$, which match the bounds for the algorithm [11] for the LZW compression.

5 Algorithm in detail

This section discusses the realizations of the function *Jump* and the procedure that enumerates the set *Output* in order to prove Theorems 1 and 2.

5.1 Realization of *Jump*

For an integer j with $0 \leq j \leq m$ and for a factor u of P, let us denote by $N_1(j, u)$ the largest integer k with $1 \leq k \leq j$ such that $P[j-k+1:j] \cdot u$ is a prefix of P. Let $N_1(j, u) = nil$, if no such integer exists. Then, for any j with $0 \leq j \leq m$ and any string u,

$$\textit{Jump}(j, u) = \begin{cases} N_1(j, u) + |u|, & \text{if } u \text{ is a factor of } P \text{ and} \\ & N_1(j, u) \neq nil; \\ \textit{Jump}(0, u), & \text{otherwise.} \end{cases}$$

We assume that the second argument u of N_1 is given as a node of the suffix trie [3] ST_P for P. Let *Factor*(v) denote the set of factors of a string v. Amir *et al*. [1] showed the following fact.

Lemma 2 (Amir et al. 1996) *The function which takes as input $(j, u) \in \{0, \ldots, m\} \times$ Factor(P) and returns the value $N_1(j, u)$ in $O(1)$ time, can be realized in $O(m^2)$ time and space.*

Let *Node*$_{ST_P}(u)$ denote the node of ST_P representing a factor u of P. For a string $u \notin$ *Factor*(P), let *Node*$_{ST_P}(u) = nil$. We need the following tables both of size $\|\mathcal{D}\|$.

- The table storing the values *Node*$_{ST_P}(X)$ for the variables X in $\mathcal{D}$.
- The table storing the values *Jump*$(0, X)$ for the variables X in $\mathcal{D}$.

For a string $u \in \Sigma^*$, let

lpf(u) = the longest prefix of the string u that is also a factor of P, and

lsf(u) = the longest suffix of the string u that is also a factor of P.

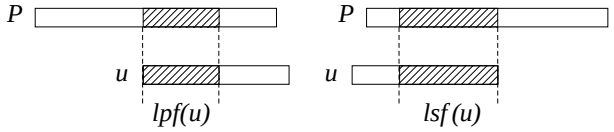

Figure 2. *lpf*(u) **and** *lsf*(u) **for pattern** P**.**

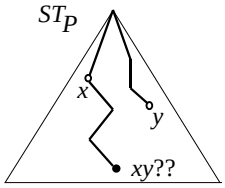

Figure 3. Factor concatenation problem.

As will be shown in the proof of Theorem 1, the values *Node*$_{ST_P}(X)$ and *Jump*$(0, X)$ are obtained from the values *lpf*(X) and *lsf*(X). Thus we concentrate ourselves on how to realize the tables of size $\|\mathcal{D}\|$ which store the values *lpf*(X) and *lsf*(X) for the variables $X \in \mathcal{D}$, respectively.

First, we consider the following problem which we will refer to as the *factor concatenation problem.*

Instance: Two factors x and y of P each represented as a node of ST_P.

Question: Is the string xy a factor of P? If 'Yes' then return the node of ST_P representing the string xy. Otherwise return nil.

A naive solution to this problem is to store the all answers in a two-dimensional table of size $|\mathit{Factor}(P)|^2 = O(m^4)$. This table size can be reduced to $O(m^3)$ by reducing the number of entries to the second argument y to $O(m)$. Namely, we consider only the factors y that are represented as explicit nodes of ST_P. It seems that the same idea can be applied to the first argument x to reduce the table size to $O(m^2)$. To do this, we will change the contents of the table as follows.

For any factors x and y of P, let *Boundary*(x, y) denote the smallest integer k with $2 \leq k \leq m$ such that $x = P[k - |x| : k - 1]$ and $y = P[k : k + |y| - 1]$. If no such integer, let *Boundary*$(x, y) = nil$. Using this function we get a position of an occurrence of xy in P, and then we can obtain the value *Node*$_{ST_P}(xy)$ using an $O(m^2)$ size table which stores the values *Node*$_{ST_P}(P[i : j])$ for all pairs of integers i and j such that $0 \leq i \leq j \leq m$. Thus we focus on the realization of the function *Boundary*(x, y).

Lemma 3 *The function Boundary*(x, y) *can be realized in* $O(m^2)$ *time and space so that it answers in* $O(1)$ *time.*

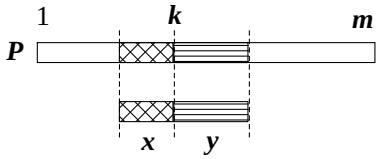

Figure 4. *Boundary*(x, y)**.**

Proof. The table storing the one-to-one correspondence between the nodes x of ST_P and the nodes x^R of ST_{P^R} can be built in $O(m^2)$ time and space. We can assume that the node representing y is an explicit node of ST_P, and the node representing x^R is an explicit node of ST_{P^R}. Therefore the function *Boundary*(x, y) can be stored in an $O(m^2)$ size table. We can compute such table in the following manner.

1. Let *Boundary*$(x, y) := nil$ for all explicit nodes x^R and y.

2. For each $k = 2, 3, \ldots, m$, and for each suffix x of $P[1 : k-1]$ such that x^R is an explicit node of ST_{P^R}, perform the following task:
   For each prefix y of $P[k : m]$ that is an explicit node of ST_P in the descending order of length, execute the statement *Boundary*$(x, y) := k$ until we encounter a string y such that *Boundary*$(x, y) \neq nil$.

We can show that the time complexity of the computation is only $O(m^2)$ although it seems to be $O(m^3)$. ■

Thus we have the following lemma.

Lemma 4 *Given a pattern* P *of length* m*, a data structure which solves the factor concatenation problem in* $O(1)$ *time, can be built in* $O(m^2)$ *time and space.*

Then we can prove the following lemma.

Lemma 5 *The tables which store the values lsf*(X) *and lpf*(X) *for the variables* X *in* $\mathcal{D}$ *can be computed in* $O(\|\mathcal{D}\| \cdot \mathit{height}(\mathcal{D}) + m^2)$ *time using* $O(\|\mathcal{D}\| + m^2)$ *space. If* $\mathcal{D}$ *contains no truncation, the time complexity becomes* $O(\|\mathcal{D}\| + m^2)$.

Proof. We show how we compute the values *lsf*(X) and *lpf*(X) for all variables X which are defined either $X = a$, $X = YZ$, $X = Y^k$, $X = {}^{[k]}Y$, $X = Y^{[k]}$, assuming that the values *lsf*(Y), *lsf*(Z), *lpf*(Y), and *lpf*(Z) are already computed, where X, Y, Z are variables and k is a positive integer. We show only the computation of *lpf*(X) since *lsf*(X) can be computed a symmetric way.

Case 1: $X = a$. It is not hard to see that *lpf*$(X) = a$ if and only if a appears in P.

Case 2: $X = YZ$. Note that, if $|\mathit{lpf}(Y)| < |Y|$, $\mathit{lpf}(X) = \mathit{lpf}(Y)$, and otherwise, $\mathit{lpf}(X) = \mathit{lpf}(Y \cdot \mathit{lpf}(Z))$. We need the function which returns $\mathit{lpf}(xy)$ for any pair of factors x and y of P. Based on the table $\mathit{Boundary}(x, y)$, we can build an $O(m^2)$ size table which stores the values $\mathit{lpf}(xy)$ for all pairs of x and y such that x^R is an explicit node of ST_{P^R}, and y is an explicit node of ST_P, and the computation requires only $O(m^2)$ time.

Case 3: $X = Y^k$. It is trivial for $k \leq 2$. Suppose $k > 2$. We can obtain $\mathit{lpf}(YY)$ in constant time. If $|\mathit{lpf}(YY)| < |YY|$, then $\mathit{lpf}(X) = \mathit{lpf}(YY)$. If $|\mathit{lpf}(YY)| = |YY|$, we have to get the longest continuation of the period Y to the right among the all occurrences of YY in P. The smallest periods of all factors of P can be computed in $O(m^2)$ time and space. We store the smallest periods into the nodes of ST_P, and build a data structure by which we can obtain, for every factor u of P, the longest factor v of P with the same period as u such that u is a prefix of v.

Case 4: $X = {}^{[k]}Y$. Let $Q(Y, k)$ be the function which returns the value $\mathit{lpf}({}^{[k]}Y)$. Consider the computation of $Q(Y, k)$. It is trivial for $Y = a$ ($a \in \Sigma \cup \{\varepsilon\}$). When $Y = Y_1Y_2$, we have $X = ({}^{[k]}Y_1) \cdot Y_2$ or $X = {}^{[k']}Y_2$ depending on whether $k \leq |Y_1|$ or not, where $k' = k - |Y_1|$. Therefore $Q(Y, k)$ is computed by a call of either $Q(Y_1, k)$ or $Q(Y_2, k')$. When $Y = (Y_1)^i$, we have $X = ({}^{[k']}Y_1)(Y_1)^j$, for some j. Thus $Q(Y, k)$ is computed by a call of $Q(Y_1, k')$. When $Y = {}^{[i]}Y_1$, it is trivial since $X = {}^{[i+k]}Y_1$. When $Y = Y_1^{[i]}$, since $X = {}^{[k]}(Y_1^{[i]}) = ({}^{[k]}Y_1)^{[i]}$, we can compute the value $Q(X, k)$ from the values $Q(Y_1, k)$ and i.

Case 5: $X = Y^{[k]}$. It is not hard to see that $\mathit{lpf}(X) = \mathit{lpf}(Y)$, if $|Y| - k > |\mathit{lpf}(X)|$, and $\mathit{lpf}(X) = (\mathit{lpf}(Y))^{[k-(|Y|-|\mathit{lpf}(Y)|)]}$, otherwise.

Since recursive call of the function $Q(X, k)$ continues at most $\mathit{height}(X)$ times, the value $\mathit{lpf}(X)$ is computed in $O(\mathit{height}(X))$ time. ■

For a string $u \in \Sigma^*$, let

$\mathit{lps}(u)$ = the longest prefix of the string u that is also a suffix of P, and

$\mathit{lsp}(u)$ = the longest suffix of the string u that is also a prefix of P.

Now we are ready to prove Theorem 1.

Proof of Theorem 1 Note that $\mathit{Node}_{ST_P}(X) = \mathit{lpf}(X)$, if $|X| = |\mathit{lpf}(X)|$, and nil, otherwise. Also note that $\mathit{Jump}(0, X) = \mathit{lsp}(X) = \mathit{lsp}(\mathit{lsf}(X))$. The table which stores the values $\mathit{lsp}(u)$ for all factors u of P can be computed in $O(m^2)$ time and space. The proof is complete. ■

5.2 Realization of *Output*

Recall the definition of the set $\mathit{Output}(j, u)$. According to whether a pattern occurrence covers the boundary between the strings $P[1 : j]$ and u, we can partition the set $\mathit{Output}(j, u)$ into two disjoint subsets as follows.

$$\mathit{Output}(j, u) = \mathit{Output}(j, \mathit{lpps}(u)) \cup \mathit{Output}(0, u),$$

where $\mathit{lpps}(u)$ denotes the longest prefix of the string u that is also a proper suffix of P. Since it holds that $\mathit{lpps}(X) = \mathit{lpps}(\mathit{lpf}(X))$, we can obtain $\mathit{lpps}(X)$ for $X \in \mathcal{D}$ in $O(1)$ time using a table which stores the values $\mathit{lpps}(u)$ for $u \in \mathit{Factor}(P)$. Such a table can be constructed in $O(m^2)$ time and space.

First, we consider the subset $\mathit{Output}(j, \mathit{lpps}(u))$. Let $\mathit{PrefSuff}(j, k) = \mathit{Occ}\big(P, P[1 : j] \cdot P[m-k+1 : m]\big) \ominus j$. It holds that $\mathit{Output}(j, \mathit{lpps}(u)) = \mathit{PrefSuff}(j, |\mathit{lpps}(u)|) \setminus \{0\}$, where we exclude the integer 0 which corresponds to the case of $j = m$. Then, it follows from Lemma 1 that the set $\mathit{PrefSuff}(j, k)$ has the following property.

Lemma 6 *If $\mathit{PrefSuff}(j, k)$ has more than two elements, it forms an arithmetic progression, where the step is the smallest period of P.*

Lemma 7 *The table $\mathit{PrefSuff}(j, k)$ for all pairs $(k, \ell) \in \{0, \ldots, m\} \times \{0, \ldots, m\}$ can be computed in $O(m^2)$ time and space. Each entry of the table occupies only $O(1)$ space.*

Proof. It follows from Lemma 6 that $\mathit{PrefSuff}(j, k)$ can be stored in $O(1)$ space as a pair of the minimum and the maximum values in it. The table storing the minimum values of $\mathit{PrefSuff}(j, k)$ for all (k, ℓ) can be computed in $O(m^2)$ time as stated in [1]. (Table N_2 defined in [1] satisfies $\min(\mathit{PrefSuff}(j, k)) = m - N_2(k, \ell)$.) By reversing the pattern P, the table of the maximum values is also computed in $O(m^2)$ time. The smallest period of P is computed in $O(m)$ time. ■

From the above, we have the following lemma.

Lemma 8 *The procedure which enumerates the set $\mathit{Output}(j, \mathit{lpps}(u))$ for $j \in \{0, \ldots, m\}$ and $u \in D$ can be realized in $O(m^2)$ time and space, and it can run in $O(\ell)$ time, where $\ell = |\mathit{Output}(j, \mathit{lpps}(u))|$.*

Next, we consider the subset $\mathit{Output}(0, u)$. Then, it holds that $\mathit{Output}(0, u) = \mathit{Occ}\ (P, u)$. In what follows, we give the computation of a representation of the sets $\mathit{Occ}(P, X)$ for the variables X in $\mathcal{D}$.

Denote by $\mathit{Occ}^\star(P, u \bullet v)$ the set of occurrences of P within the concatenation of two strings u and v which covers the boundary between u and v. That is, $\mathit{Occ}^\star(P, u \bullet v) =$

$\{s \in \mathit{Occ}(P, uv) \mid |u| \leq s \leq |u| + |P|\}$. Since $\mathit{lsp}(u) = \mathit{lsp}(\mathit{lsf}(u))$ and $\mathit{lps}(u) = \mathit{lps}(\mathit{lpf}(u))$, we can obtain in $O(1)$ time the values $\mathit{lsp}(X)$ and $\mathit{lps}(X)$ for $X \in \mathcal{D}$ using the table which stores the values $\mathit{lsp}(u)$ and $\mathit{lps}(u)$ for $u \in \mathit{Factor}(P)$, respectively. Then we have the next two lemmas.

Lemma 9 *For $X = YZ$, the set $\mathit{Occ}(P, X)$ can be computed in $O(1)$ time using the table PrefSuff, assuming that the sets $\mathit{Occ}(P, Y)$ and $\mathit{Occ}(P, Z)$ and the values $|\mathit{lsp}(Y)|$ and $|\mathit{lps}(Z)|$ are already computed.*

Proof. It follows from the facts $\mathit{Occ}^\star(P, Y \bullet Z) = \mathit{PrefSuff}\Big(|\mathit{lsp}(Y)|, |\mathit{lps}(Z)|\Big)$ and $\mathit{Occ}(P, X) = \mathit{Occ}(P, Y) \cup ((\mathit{Occ}^\star(P, Y \bullet Z) \cup \mathit{Occ}(P, Z)) \oplus |Y|)$. ■

Lemma 10 *For $X = Y^k$ with $k > 1$, the set $\mathit{Occ}(P, X)$ can be computed in $O(1)$ time using the table PrefSuff, assuming that the set $\mathit{Occ}(P, Y)$ and the value $|\mathit{lsp}(Y)|$ and $|\mathit{lps}(Y)|$ are already computed.*

Proof. We have three cases to consider.

- Case 1: $|P| \leq |Y|$. Since P cannot cover more than two Y's, $\mathit{Occ}(P, X)$ is represented by a four tuple of a pointer to $\mathit{Occ}(P, Y)$, $\mathit{Occ}^\star(P, Y \bullet Y)$, $|Y|$, and k.
- Case 2: $|Y| < |P| < 2|Y|$. We build two sets $\mathit{Occ}^\star(P, Y \bullet Y)$ and $\mathit{Occ}^\star(P, Y \bullet YY) \backslash \mathit{Occ}^\star(P, Y \bullet Y)$. These sets are computed only in $O(1)$ time and space. $\mathit{Occ}(P, X)$ is represented by these sets, $|Y|$ and k.
- Case 3: $2|Y| \leq |P|$. Note that P occurs within Y^ℓ for some $\ell > 0$ if and only if (1) Y is a factor of P, and (2) $|Y|$ is a period of P. The first item is true if $\mathit{Node}_{ST_P}(Y) \neq nil$. The second item is true if $|Y|$ is a multiple of the smallest period t of P (recall the periodicity lemma). The set $\mathit{Occ}(P, X)$ forms an arithmetic progression, whose step is t. ■

Lemma 11 *We can build in $O(\|\mathcal{D}\| \cdot \mathit{height}(\mathcal{D}) + m^2)$ time using $O(\|\mathcal{D}\| + m^2)$ space a data structure by which the enumeration of the set $\mathit{Occ}(P, X)$ is performed in $O(\mathit{height}(X) + \ell)$ time, where $\ell = |\mathit{Occ}(P, X)|$. If $\mathcal{D}$ contains no truncation, it can be built in $O(\|\mathcal{D}\| + m^2)$ time and space, and the enumeration requires only $O(\ell)$ time.*

Proof. Recall the syntax trees defined in Section 3. A node labeled by X of a syntax tree is said to be *active* if (1) it has a child labeled by Y such that either $\mathit{Occ}(P, X) \neq \mathit{Occ}(P, Y)$, or (2) it is a leaf node and $\mathit{Occ}(P, X) \neq \emptyset$. The equality testing of the sets is replaced by the equality testing of their cardinalities, since it holds that either $\mathit{Occ}(P, X) \supseteq \mathit{Occ}(P, Y) \oplus k$ for concatenation and repetition, or $\mathit{Occ}(P, X) \subseteq \mathit{Occ}(P, Y) \oplus k$ for truncation, where k is an appropriate offset.

It is not difficult to show that the table $\mathit{Card}(X)$ which stores the cardinalities of $\mathit{Occ}(P, X)$ for all variables X in $\mathcal{D}$, can be computed in $O(\|\mathcal{D}\| \cdot \mathit{height}(\mathcal{D}) + m^2)$ time using $O(\|\mathcal{D}\| + m^2)$ space. If $\mathcal{D}$ contains no truncation, it can be computed in $O(\|\mathcal{D}\| + m^2)$ time and space.

Next, using the table *Card*, we add, for each node v labeled by X, pointers as short-cut from it into the nearest active descendants. If v has two children, we add two pointers. By using these pointers, we can skip the inactive nodes in traversing the syntax trees so that the enumeration is completed in linear time proportional to the number of elements. To report the exact positions of pattern occurrences, we also associate the 'offset' information.

We now briefly describe how to enumerate the set $\mathit{Occ}(P, X)$ for a variable X. When there is no truncation, we have only to traverse the syntax tree $\mathcal{T}(X)$ utilizing the short-cut pointers, and output the position of occurrences. The time complexity is obviously linear to the number of occurrences in this case. When we encounter a suffix truncation, we monitor the enumeration in its descendants and terminate the process if it exceeds the condition. If we encounter a prefix truncation, a kind of binary search will navigate us in $O(\mathit{height}(X))$ time to the first position of the occurrence in its subtree. Then we continue the enumeration. ■

Proof of Theorem 2 It follows from Lemmas 8 and 11. ■

6 Concluding remarks

We introduced a collage system which is an abstraction of various dictionary-based compression methods. We developed a general compressed matching algorithm which runs in $O((\|\mathcal{D}\| + |\mathcal{S}|) \cdot \mathit{height}(\mathcal{D}) + m^2 + r)$ time with $O(\|\mathcal{D}\| + m^2)$ space. The factor $\mathit{height}(\mathcal{D})$ can be dropped if the collage system contains no truncation. It coincides with the observation by Navarro and Raffinot [14] that LZ77 compression is not suitable for compressed pattern matching compared with LZ78 compression. Recall that LZ77 requires truncation in our collage system while LZ78 does not. They proposed a new hybrid compression method of LZ77 and LZ78, whose intention is to achieve both effective compression and efficient compressed pattern matching [14]. We can represent their compression method by a collage system with no truncation.

For dealing with multiple patterns, we need to modify the function *Jump* and the procedure for enumerating *Output*. We have verified that *Jump* can be generalized to treat multiple patterns. Although we omit the detail, the idea is almost the same as [11]. That is, we simulate the move of the AC automaton instead of the KMP automaton, and use the generalized suffix trie [8]. For *Output*, we have also done if a collage system contains neither repetitions nor truncations. The rest is left for our future work.

Kosaraju [12] showed a faster pattern matching algorithm for LZW compression, which runs in $O(n + m\sqrt{m}\log m)$ time. It is a challenging problem to achieve this bound in our general framework.

References

[1] 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.

[2] A. Apostolico and Z. Galil. *Pattern Matching Algorithm*. Oxford University Press, New York, 1997.

[3] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.

[4] 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.

[5] 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.

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

[7] L. Gasieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 5th Scandinavian Workshop on Algorithm Theory*, pages 392–403, 1996.

[8] 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.

[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] 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. Springer-Verlag, 1999. to appear.

[11] 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. Data Compression Conference '98*, pages 103–112. IEEE Computer Society, 1998.

[12] S. Kosaraju. Pattern matching in compressed texts. In *Proc. Foundation of Software Technology and Theoretical Computer Science*, pages 349–362. Springer-Verlag, 1995.

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

[14] 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. Springer-Verlag, 1999. to appear.

[15] M. Nelson. *The data compression book*. M&T Publishing, Inc., Redwood City, Calif., 1992.

[16] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte pair encoding: a text compression scheme that accelerates pattern matching. Technical Report DOI-TR-CS-161, Department of Informatics, Kyushu University, April 1999.

[17] 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. Springer-Verlag, 1999. to appear.

[18] J. Storer and T. Szymanski. Data compression via textual substitution. *J. Assoc. Comput. Mach.*, 29(4):928–951, Oct 1982.

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

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