Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Theoretical Computer Science 298 (2003) 253–272

Theoretical
Computer Science

www.elsevier.com/locate/tcs

Collage system: a unifying framework for compressed pattern matching

Takuya Kida[a,*,1], Tetsuya Matsumoto[a], Yusuke Shibata[a], Masayuki Takeda[a,b], Ayumi Shinohara[a], Setsuo Arikawa[a]

[a] *Department of Informatics, Kyushu University, 33 Fukuoka 812-8581, Japan*
[b] *PRESTO, Japan Science and Technology Corporation, Japan*

Abstract

We introduce a general framework which is suitable to capture the essence of *compressed pattern matching* according to various dictionary-based compressions. It is a formal system to represent a string by a pair of dictionary D and sequence S of phrases in D. The basic operations are concatenation, truncation, and repetition. We also 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), RE-PAIR, SEQUITUR, and the static dictionary-based method. The proposed algorithm runs in $O((\|D\| + |S|) \cdot height(D) + m^2 + r)$ time with $O(\|D\| + m^2)$ space, where $\|D\|$ is the size of D, $|S|$ is the number of tokens in S, $height(D)$ is the maximum dependency of tokens in D, m is the pattern length, and r is the number of pattern occurrences. For a subclass of the framework that contains no truncation, the time complexity is $O(\|D\| + |S| + m^2 + r)$.

Keywords: String matching; Compressed pattern matching; Data compression; Collage system

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

* Corresponding author.
E-mail addresses: kida@i.kyushu-u.ac.jp (T. Kida), t-matsu@i.kyushu-u.ac.jp (T. Matsumoto), yusuke@i.kyushu-u.ac.jp (Y. Shibata), takeda@i.kyushu-u.ac.jp (M. Takeda), ayumi@i.kyushu-u.ac.jp (A. Shinohara), arikawa@i.kyushu-u.ac.jp (S. Arikawa).

[1] Research Fellow of the Japan Society for the Promotion of Science (JSPS). Partly supported by Grant-in-Aid for JSPS research fellows (12000410).

0304-3975/03/$ - see front matter
PII: S0304-3975(02)00426-7

classical or advanced pattern matching algorithms have been proposed (see [6,7]). 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 [30]).

Recently, the *compressed pattern matching* problem has attracted special concern where the goal is to find a pattern in a compressed text without decompressing it. The problem was first defined by Amir and Benson [1], and various compressed pattern matching algorithms have been proposed depending on underlying compression methods: Eilam-Tzoreff and Vishkin [10] addressed the run-length compression, and Amir et al. [5], and Amir and Benson [1,2] and Amir et al. [3] addressed its two-dimensional version. Farach and Thorup [11] and Gąsieniec et al. [12] addressed the LZ77 compression [40]. Amir et al. [4] addressed the LZW compression [36].

However, it seems that most of these studies were undertaken mainly from the theoretical viewpoint, and less attention was paid to practical aspect of the problem. For example, the algorithm in [11] achieved an $O(n \log^2 N/n + m)$ time complexity, where n is the compressed text length, N is the original text length, and m is the pattern length, but the constant factor hidden behind the O-notation is relatively large. In fact, an experiment showed that the algorithm spent enormous time and was slower than a decompression followed by a simple search. On the other hand, one of the algorithms proposed by Amir et al. [4] runs in $O(n + m^2)$ time over an Lempel–Ziv–Welch (LZW) compressed text of length n, and the experimental results in [19] showed that it is about twice faster than a decompression followed by a search with *Agrep* [37]. The basic idea of the algorithm is to simulate the move of the Knuth–Morris–Pratt (KMP) automaton [7] on the compressed text directly. In [19,20] we have extended it in order to find all occurrences of *multiple* patterns simultaneously, by simulating the move of the Aho–Corasick pattern matching machine [7]. 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 pattern matching machine. In [18,20], we also presented another implementation of the algorithm utilizing *bit-parallelism*, and reported some experiments.

Navarro and Raffinot [29] 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.

For other compression methods, we developed compressed pattern matching algorithms for compressed text using anti-dictionaries [34], and for compressed text using byte-pair encoding [32]. Especially, the latter was showed to be even faster than pattern matching in uncompressed texts. Miyazaki et al. [27] presented an efficient realization of pattern matching machine for searching directly in a Huffman encoded text. Moura et al. [8,9] addressed a new compression scheme that uses a word-based Huffman encoding with a byte-oriented code.

In this paper, we introduce a *collage system*, that is a formal system to represent a string by a pair of dictionary D and sequence S of phrases in 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), 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 $\|D\|$ grows $O(n \log n)$ and truncation operations are required. We remark that a straight-line program [17] is a collage system containing concatenation only, and a composition system introduced in [13] is also a collage system which allows concatenation and truncation.

It should be stated that Kieffer et al. [38,22,21] proposed a compression scheme called *grammar transform*. Their idea is to build first a context-free grammar G that produces the original string w uniquely, and then encode G. The compression algorithms RE-PAIR [25] and SEQUITUR [31] fit into this scheme. We remark that the grammar transform corresponds to a subclass of collage systems called *regular*.

We develop a compressed pattern matching algorithm for collage systems which contain no truncation, whose running time is $O(\|D\| + |S| + m^2 + r)$ using $O(\|D\| + m^2)$ space, where $\|D\|$ denotes the size of the dictionary D and $|S|$ is the length of the sequence S. For the case of LZW compression, it matches the bound $O(n + m^2 + r)$ in [20]. For general collage systems, which contain truncation, we show a compressed pattern matching algorithm which runs in $O((\|D\| + |S|) \cdot \mathit{height}(D) + m^2 + r)$ time with $O(\|D\| + m^2)$ space, where $\mathit{height}(D)$ denotes the maximum dependency of the operations in D. These results show that the truncation slows down the compressed pattern matching to the factor $\mathit{height}(D)$.

2. Preliminaries

Let Σ be a finite set of characters, called an *alphabet*. A finite sequence of characters is called a *string*. Let Σ^* be the set of strings over Σ. 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. Let $\mathit{Prefix}(u)$ be the set of prefixes of a string u. We also define the sets *Suffix* and *Factor* in a similar way. For strings $u, v \in \Sigma^*$, let

$$\mathit{lpf}_v(u) = \text{the longest prefix of } u \text{ that is also in } \mathit{Factor}(v),$$

$$\mathit{lsf}_v(u) = \text{the longest suffix of } u \text{ that is also in } \mathit{Factor}(v),$$

$$\mathit{lps}_v(u) = \text{the longest prefix of } u \text{ that is also in } \mathit{Suffix}(v),$$

$$\mathit{lsp}_v(u) = \text{the longest suffix of } u \text{ that is also in } \mathit{Prefix}(v)$$

(see Figs. 1 and 2). From above definitions, we have the following fact.

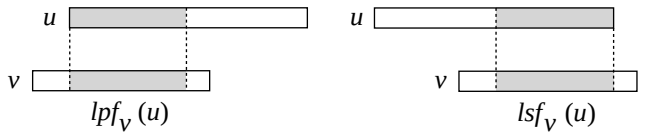

Fig. 1. $lpf_v(u)$ and $lsf_v(u)$.

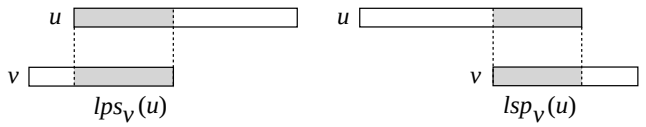

Fig. 2. $lps_v(u)$ and $lsp_v(u)$.

Fact 1. *For strings $u, v \in \Sigma^*$, we have $lps_v(u) = lps_v(lpf_v(u))$ and $lsp_v(u) = lsp_v(lsf_v(u))$.*

We denote the length of a string u by $|u|$ and the cardinality of a set V by $|V|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$.

The ith symbol of a string u is denoted by $u[i]$ for $1 \leqslant i \leqslant |u|$, and the factor of a string u that begins at position i and ends at position j is denoted by $u[i:j]$ for $1 \leqslant i \leqslant j \leqslant |u|$. Denote by ${}^{[i]}u$ (resp. $u^{[i]}$) the string obtained by removing the length i prefix (resp. suffix) from u for $0 \leqslant i \leqslant |u|$. That is, ${}^{[i]}u = u[i+1:|u|]$ (resp. $u^{[i]} = u[1:|u|-i]$). The concatenation of i copies of the same string u is denoted by u^i. The reversed string of a string u is denoted by u^R.

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, we denote the set of occurrences of x in y by $Occ(x, y)$. That is, $Occ(x, y) = \{i \mid |x| \leqslant i \leqslant |y|, x = y[i - |x| + 1 : i]\}$. Also denote by $Occ^{\star}(x, u \bullet v)$ the set of occurrences of x within the concatenation of two strings u and v which covers the boundary between u and v. That is, $Occ^{\star}(x, u \bullet v) = \{i \mid i \in Occ(x, uv), |u| < i < |u| + |x|\}$.

A *period* of a string u is an integer p, $0 < p \leqslant |u|$, such that $x[i] = x[i + p]$ for all $i \in \{1, \ldots, |x| - p\}$. The next lemma provides an important property on periods of a string.

Lemma 1 (Periodicity Lemma (see Crochemore and Rytter [7])). *Let p and q be two periods of a string x. If $p + q - \gcd(p, q) \leqslant |x|$, then $\gcd(p, q)$ is also a period of x.*

The next lemma follows from the periodicity lemma.

Lemma 2. *Let x and y be strings. If $Occ(x, y)$ 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.*

Proof. The proof is essentially the same as that of Lemma 3.1 in [28]. Let i, j, and k be consecutive elements arbitrarily chosen from $Occ(x, y)$ in the increasing order. We will

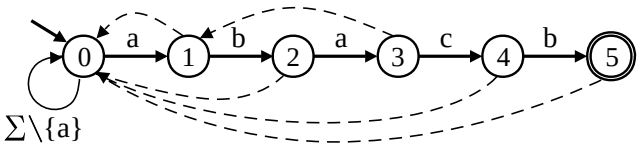

Fig. 3. KMP automaton for $\pi = abacb$. The circles denote the states, and the thick circle the final state. The solid and the broken arrows represent the goto and the failure functions, respectively.

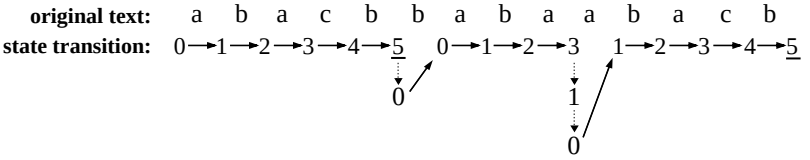

Fig. 4. Move of KMP automaton. The solid and the broken arrows represent the state transitions with the goto and the failure functions, respectively. The underlined number indicates that the pattern occurs.

show that $j - i = k - j$, which implies that $Occ(x, y)$ forms an arithmetic progression. Since $i, k \in Occ(x, y)$ and the difference of the maximum and the minimum elements in $Occ(x, y)$ is at most $|x|$, we have $k - i \leqslant |x|$. Let p_0 be the smallest period of x, and let $p_1 = j - i$ and $p_2 = k - j$. Since both p_1 and p_2 are periods of x, and p_0 is the smallest period of x, we have $p_0 \leqslant p_1$ and $p_0 \leqslant p_2$. Thus $p_1 + p_0 \leqslant p_1 + p_2 = (j - i) + (k - j) = k - i \leqslant |x|$. By the periodicity lemma, the greatest common divisor d of p_1 and p_0 is also a period of x. Since p_0 is the smallest period, we have $d = p_0$, which implies that $p_1 = \ell \cdot p_0$ for some $\ell \geqslant 1$. Suppose $\ell \geqslant 2$. Then $j = i + \ell \cdot p_0 > i + p_0 > i$. Since both i and j are in $Occ(x, y)$, and p_0 is a period of x, we have $i + p_0 \in Occ(x, y)$. This contradicts the assumption that i and j are consecutive elements in $Occ(x, y)$. Therefore $\ell = 1$, that is $p_1 = p_0$. In the same way, we can see that $p_2 = p_0$. The proof is complete. □

2.1. Knuth–Morris–Pratt algorithm

The Knuth–Morris–Pratt (KMP) algorithm [23] is a classical linear time pattern matching algorithm, which uses an automaton (KMP automaton) built from a given pattern. The KMP automaton for a pattern π consists of two functions:

goto function $g : Q \times \Sigma \to Q \cup \{fail\}$, and

failure function $f : Q \setminus \{0\} \to Q$,

where $Q = \{0, 1, \ldots, |\pi|\}$ is the set of states, and *fail* is a special value not in Q. The goto function g takes $j \in Q$ and $a \in \Sigma$ as input and returns $j+1$ if $\pi[j+1] = a$, otherwise, returns *fail*. (The case $j = 0$ is an exception. Let $g(0, a) = 0$ for every $a \in \Sigma$ with $\pi[1] \neq a$.) The failure function f takes $j \in Q \setminus \{0\}$ as input and returns the maximum integer k such that $\pi[1 : k] = \pi[j - k + 1 : j]$. Fig. 3 shows the KMP automaton for $\pi = abacb$ with $\Sigma = \{a, b, c\}$. The move of the KMP automaton of Fig. 3 on the text *abacbbaababacbb* is shown in Fig. 4. If the state reaches to 5, it implies that the pattern

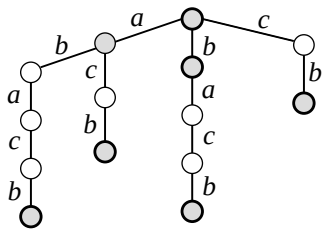

Fig. 5. Suffix trie for $\pi = abacb$. The explicit nodes are shaded, and the nodes which represent suffixes of π are indicated by the thick circles.

occurs in the text. Note that the states of the KMP automaton for π have a one-to-one correspondence with the prefixes of π. For example, the initial state 0 corresponds to the empty string ε and the state 4 corresponds to the string *abac* in Fig. 3.

To eliminate the failure function, we define the state transition function $\delta : Q \times \Sigma \to Q$ by

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

and extend δ to the domain $Q \times \Sigma^*$ in the standard manner, namely,

$$\delta(j,\varepsilon) = j, \quad \delta(j,ua) = \delta(\delta(j,u),a),$$

where $j \in Q$, $u \in \Sigma^*$ and $a \in \Sigma$. The following lemma characterizes the function δ.

Lemma 3. *For any $j \in Q$ and $u \in \Sigma^*$, we have $\delta(j,u) = |lsp_\pi(\pi[1:j] \cdot u)|$.*

2.2. Suffix trie

A *suffix trie* for a pattern π, denoted by ST_π, is the trie representing the set of suffixes of π. Fig. 5 shows ST_π for $\pi = abacb$. A node of ST_π is said to be *explicit* if and only if either it represents a suffix of π or its out-degree is more than one. The nodes that are not explicit are said to be *implicit*. Note that ST_π can be built in $O(m^2)$ time and space, where $m = |\pi|$, and that the number of explicit nodes in ST_π is $O(m)$, whereas the number of all nodes is $O(m^2)$ (see [7]).

Note that each node of ST_π corresponds to a string in $Factor(\pi)$. Hereafter, we identify a string $x \in Factor(\pi)$ with the node representing x if no confusion occurs. For example, 'to compute $lpf_\pi(x)$' means 'to compute the node of ST_π representing the string $lpf_\pi(x)$'.

For a string $x \in Factor(\pi)$, denote by $\overleftarrow{x}$ the longest string $y \in Factor(\pi)$ such that $Occ(y,\pi) = Occ(x,\pi)$. Also denote by $\overrightarrow{x}$ the longest string $y \in Factor(\pi)$ such that $Occ(y,\pi) \ominus |y| = Occ(x,\pi) \ominus |x|$. Let α and β be the strings such that $\overleftarrow{x} = \alpha x$ and $\overrightarrow{x} = x\beta$. Intuitively, $\overleftarrow{x} = \alpha x$ (resp. $\overrightarrow{x} = x\beta$) means that every occurrence of x in π is preceded by α (resp. followed by β) and the string α (resp. β) is as long as possible.

Although $\overleftarrow{x}$ and $\overrightarrow{x}$ depend on a pattern π, we omit it for convenience. For $\pi = abacb$, we have $\overleftarrow{\varepsilon} = \overrightarrow{\varepsilon} = \varepsilon$, $\overrightarrow{a} = a$, $\overleftarrow{ba} = aba$, $\overrightarrow{ba} = bacb$. Note that, for any $x \in Factor(\pi)$, the node of ST_π representing $\overrightarrow{x}$ is explicit. Moreover, the node of ST_π representing $\overrightarrow{x}$ is the nearest explicit descendant of the node representing x. Similarly, the node of ST_{π^R} representing $(\overleftarrow{x})^R$ is explicit. The table that stores $\overrightarrow{x}$ (resp. $\overleftarrow{x}$) for all $x \in Factor(\pi)$ can be built in $O(m^2)$ time and space, by traversing over ST_π (resp. ST_{π^R}).

We can merge two tries ST_π and ST_{π^R} into a data structure [14]. In the data structure, the node representing x and the node representing x^R are exactly the same, and the reverses of the edges of ST_π are identical to the suffix links of ST_{π^R}, and vice versa.

3. Collage system and text compressions

Dictionary-based 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 $\langle D, S\rangle$ defined as follows: D is a sequence of assignments $X_1 = expr_1; X_2 = expr_2; \cdots; X_n = expr_n$, where each X_k is a token 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 token represents a string obtained by evaluating the expression as it implies. The strings represented by tokens are called *phrases*. Denote by $X.u$ the phrase represented by a token X. The *size* of D is the number n of assignments and denoted by $\|D\|$. Also denote by $F(D)$ the set of tokens defined in D. That is, $\|D\| = |F(D)| = n$.

The syntax tree of a token X in D, denoted by $T(X)$, is defined inductively as follows. The root node of $T(X)$ is labeled by X and has:

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

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

On the other hand, $S = X_{i_1}, X_{i_2}, \dots, X_{i_k}$ is a sequence of tokens defined in D. We denote by $|S|$ the number k of tokens in S. The collage system represents a string

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

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. In this case, the definition of the size of D should be changed from the number of assignments to the total length of right-hand sides of assignments. Of course, it is easy to rewrite such an abbreviated assignment as a sequence of assignments within the formalism. For example, the abbreviated assignment $X = Y_1 Y_2 (Y_3)^3 ({}^{[2]}Y_4)$ can be translated into the sequence of assignments $X_1 = (Y_3)^3; X_2 = {}^{[2]}Y_4; X_3 = Y_1 Y_2; X_4 = X_1 X_2; X = X_3 X_4$.

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

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

where w_k is a string in Σ^+ with $|w_k| > 1$. 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 S. On the other hand, the strings $w_1, w_2, \ldots, w_s$ in D are considered to be frequent in many texts in common. It is often stored independently of the compressed texts.

LZW compression [36]: $S = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$ and D is as follows:

$$X_1 = a_1; \quad X_2 = a_2; \quad \cdots; \quad X_q = a_q;$$
$$X_{q+1} = X_{i_1} X_{\sigma(i_2)}; \quad X_{q+2} = X_{i_2} X_{\sigma(i_3)}; \quad \cdots;$$
$$X_{q+n-1} = X_{i_{n-1}} X_{\sigma(i_n)},$$

where the alphabet is $\Sigma = \{a_1, \ldots, a_q\}$, $1 \leqslant i_1 \leqslant q$, and $\sigma(\ell)$ denotes the integer k, $1 \leqslant k \leqslant q$, such that a_k is the first symbol of the phrase $X_\ell.u$. S is encoded as a sequence of integers $i_1, i_2, \ldots, i_n$ in which an integer i_j is represented in $\lceil \log_2(q+j) \rceil$ bits, while D is not encoded since it can be obtained from S.

LZ78 compression [39]: $S = X_1, X_2, \ldots, X_n$, and D is as follows:

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

where b_j is a symbol in Σ. While no need to encode S, the dictionary 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 $\|D\| = O(n \log n)$ [13]. 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 [35]: $S = X_{q+1}, X_{q+2}, \ldots, X_{q+n}$, and D is as follows:

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

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

We emphasize 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 et al. [4] presented 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 [29], 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.

Consider how to simulate the move of the KMP automaton for a pattern π running on the original text. For a collage system $\langle D, S\rangle$ and π, define the function $Jump : Q \times F(D) \to Q$ by

$$Jump(j, X) = \delta(j, X.u).$$

Input. A pattern π and a collage system $\langle D, S\rangle$, where $S = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$.
Output. All positions at which π occurs in $X_{i_1}.uX_{i_2}.u \cdots X_{i_n}.u$.

```
/* Preprocessing */
Perform the preprocessing required for Jump and Output
(The complexity of this part depends on π and D. See Section 5);
/* Text scanning */
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
```

Fig. 6. Pattern matching algorithm.

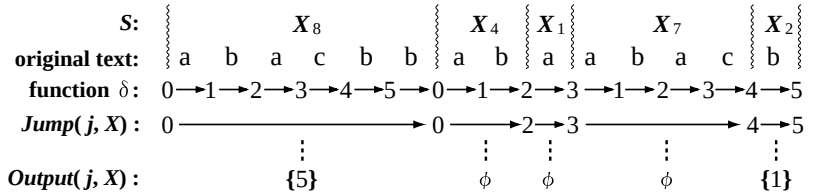

Fig. 7. Move of our algorithm.

We also define the set $Output(j,X)$ for any pair $\langle j,X\rangle$ in $Q \times F(D)$ by

$$Output(j,X) = \left\{ |v| \;\middle|\; \begin{array}{l} v \text{ is a non-empty prefix of } X.u \text{ such that} \\ \delta(j,v) \text{ is the final state.} \end{array} \right\}.$$

Our algorithm, given a pattern π and an encoding of a collage system $\langle D, S\rangle$ representing a text T, processes the sequence S token by token (i.e. phrase by phrase) to report all occurrences of π 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 a pair of an integer $j \in Q$ and a token X defined in D. An overview of the algorithm based on the function and the procedure is shown in Fig. 6. For example, Fig. 7 shows that the move of our algorithm on S for a pattern $\pi = abacb$, where D is $X_1 = a$; $X_2 = b$; $X_3 = c$; $X_4 = X_1 \cdot X_2$; $X_5 = X_1 \cdot X_3$; $X_6 = (X_2)^2$; $X_7 = X_4 \cdot X_5$; $X_8 = X_7 \cdot X_6$, and $S = X_8, X_4, X_1, X_7, X_2$.

For static dictionary-based methods, D is followed by S in the encoding, or D is given independently of S. Thus we can process D as a preprocessing. For adaptive dictionary-based methods like the Lempel–Ziv family, D is not given explicitly in the encoding of $\langle D, S\rangle$, and will be rebuilt incrementally in the token-by-token

processing of S. From the theoretical viewpoint, we can process D being incrementally reconstructed from S in the first step, and then process S again in the second step, without increasing the time complexity. In practice, we merge these two 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* $O(\|D\| \cdot height(D) + m^2)$ *time using* $O(\|D\| + m^2)$ *space, so that it answers in* $O(1)$ *time. If D contains no truncation, the time complexity becomes* $O(\|D\| + m^2)$.

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

In our framework, we can consider that the compressed text length n is $\|D\| + |S|$, 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 [20] 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. The relation between the contents of this section and the algorithm presented in Section 4 is as follows.

- *Preprocessing of the pattern*: From the beginning up to Lemma 7, and Lemmas 9 and 10.
- *Preprocessing of the dictionary D*: Lemma 8 and computation of the short-cut pointers in the proof of Lemma 14, which is based on Lemmas 11 and 13.
- *Text scanning*: The proof of Theorem 1, Lemma 12, and parts of Lemma 14.

5.1. Realization of Jump

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

Instance: Two factors x and y each represented as a node of ST_π.
Question: Is the string xy in $Factor(\pi)$? If 'Yes' then return the node of ST_π representing the string xy (see Fig. 8). Otherwise return *nil*.

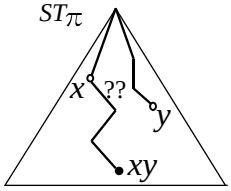

Fig. 8. Factor concatenation problem.

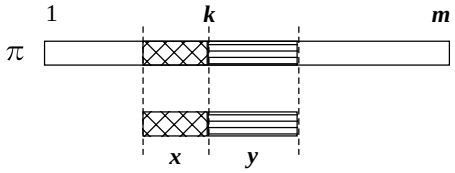

Fig. 9. *Boundary*(x, y).

A naive solution to this problem is to store all the answers in a two-dimensional table of size $|Factor(\pi)|^2 = O(m^4)$, where m is the length of the pattern π. 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_π. 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 π, let *Boundary*(x, y) denote the smallest integer k with $2 \leqslant k \leqslant m$ such that $x = \pi[k-|x| : k-1]$ and $y = \pi[k : k+|y|-1]$ (see Fig. 9). If no such integer, let *Boundary*$(x, y) = nil$. Using this function we get a position of an occurrence of xy in π, and then we can obtain the value the node of ST_π representing xy using an $O(m^2)$ size table that stores the values the node of ST_π representing $\pi[i : j]$ for all pairs of integers i and j such that $0 \leqslant i \leqslant j \leqslant m$. Thus, we focus on the realization of the function *Boundary*(x, y).

Lemma 4. *The function Boundary*(x, y) *can be realized in* $O(m^2)$ *time and space so that it answers in* $O(1)$ *time.*

Proof. From the definition of *Boundary*, it holds that, for any $x, y \in Factor(\pi)$,

$$Boundary(x, y) = Boundary(\overleftarrow{x}, \overrightarrow{y}).$$

Recall that the node representing $\overrightarrow{y}$ is an explicit node of ST_π, and the node representing $(\overleftarrow{x})^R$ is an explicit node of ST_{π^R}. Since the number of explicit nodes is $O(m)$, the number of possible pairs of $\overleftarrow{x}$ and $\overrightarrow{y}$ is $O(m^2)$. Thus the function *Boundary*$(\overleftarrow{x}, \overrightarrow{y})$ can be stored in an $O(m^2)$ size table. In order to get the value *Boundary*(x, y) for $x, y \in Factor(\pi)$, we refer to the table after computing $\overleftarrow{x}$ and $\overrightarrow{y}$ from x and y,

respectively. As mentioned in Section 2.2, we can compute $\overleftarrow{x}$ (resp. $\overrightarrow{x}$) in O(1) time for all $x \in Factor(\pi)$ after O(m^2) time and space preprocessing. Let $\overleftarrow{Fac}(\pi) = \{\overleftarrow{x} \mid x \in Factor(\pi)\}$ and $\overrightarrow{Fac}(\pi) = \{\overrightarrow{x} \mid x \in Factor(\pi)\}$. We can compute such table in the following manner.

(1) Let $Boundary(x, y) := nil$ for any pair of $x \in \overleftarrow{Fac}(\pi)$ and $y \in \overrightarrow{Fac}(\pi)$.
(2) For each $k = 2, 3, \ldots, m$, and for each suffix x of $\pi[1 : k-1]$ that is also in $\overleftarrow{Fac}(\pi)$, perform the following task:
For each prefix y of $\pi[k : m]$ that is also in $\overrightarrow{Fac}(\pi)$ 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). Consider the number of references to the table *Boundary*. Each entry of the table is changed only once. Namely, the statement $Boundary(x, y) := k$ is executed O(m^2) times totally. Therefore, the algorithm runs in O(m^2) time. □

Thus we have the following lemma.

Lemma 5. *Given a pattern π of length m, we can build in* O(m^2) *time and space a data structure that solves the factor concatenation problem in* O(1) *time.*

Also the next lemma holds.

Lemma 6. *The function that takes as input $x, y \in Factor(\pi)$ and returns $lpf_\pi(xy)$ in* O(1) *time, can be computed in* O(m^2) *time and space. This also holds for $lsf_\pi(xy)$.*

Proof. We show below only the computation of $lpf_\pi(xy)$ because $lsf_\pi(xy)$ can be computed in a symmetric way. For $u \in \overleftarrow{Fac}(\pi)$ and $v \in \overrightarrow{Fac}(\pi)$, we build the table $Lpf(u, v)$ that stores the string $w \in Prefix(v)$ such that $lpf_\pi(uv) = uw$. The table can be built in O(m^2) time by using $Boundary(x, y)$. Let $w = Lpf(\overleftarrow{x}, \overrightarrow{y})$ for $x, y \in Factor(\pi)$. Then, we have $lpf_\pi(xy) = xw$ if $|w| < |y|$, $lpf_\pi(xy) = xy$, otherwise. The proof is complete. □

Lemma 7. *We can compute in* O(m^2) *time and space the table that stores $lps_\pi(x)$ for any $x \in Factor(\pi)$. This also holds for $lsp_\pi(x)$.*

Proof. The table can be computed in O(m^2) time and space by a depth-first traversal of ST_π assuming the nodes representing suffixes are marked. For $lsp_\pi(x)$ it can be proved in a symmetric way by using ST_{π^R}. □

Lemma 8. *For any collage system $\langle D, S\rangle$ and any pattern π, the function that takes as input a token $X \in F(D)$ and returns $lpf_\pi(X.u)$ in* O(1) *time, can be computed in* O($\|D\| \cdot height(D) + m^2$) *time using* O($\|D\| + m^2$) *space. If D contains no truncation, it can be computed in* O($\|D\| + m^2$) *time and space. This also holds for $lsf_\pi(X.u)$.*

Proof. We show below only the computation of $lpf_\pi(X.u)$, because $lsf_\pi(X.u)$ can be computed in a symmetric way.

Case 1: $X = a$. It is not hard to see that $lpf_\pi(X.u) = a$ if and only if a appears in π.

Case 2: $X = YZ$. If $|lpf_\pi(Y.u)| < |Y.u|$, $lpf_\pi(X.u) = lpf_\pi(Y.u)$. Otherwise, $lpf_\pi(X.u) = lpf_\pi(Y.u \cdot lpf_\pi(Z.u))$. Then, we need $lpf_\pi(xy)$ for any pair of x and y in $Factor(\pi)$. From Lemma 6, it can be obtained in O(1) time after $O(m^2)$ time and space preprocessing.

Case 3: $X = Y^k$. It is trivial for $k \leqslant 2$. Suppose $k > 2$. We can obtain $lpf_\pi(Y.uY.u)$ in constant time (see Case 2). If $|lpf_\pi(Y.uY.u)| < |Y.uY.u|$, then $lpf_\pi(X.u) = lpf_\pi(Y.uY.u)$. If $|lpf_\pi(Y.uY.u)| = |Y.uY.u|$, we have to get the longest continuation of the period $Y.u$ to the right among the all occurrences of $Y.uY.u$ in π. The smallest periods of all factors of π can be computed in $O(m^2)$ time and space. We store the smallest periods into the nodes of ST_π, and build a data structure by which we can obtain, for every factor u of π, the longest factor v of π 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 $lpf_\pi({}^{[k]}Y.u)$. 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 $k \leqslant |Y_1.u|$ or not, where $k' = k - |Y_1.u|$. 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 $X = ({}^{[k']}Y_1)(Y_1)^j$, where $j = i - \lfloor k/|Y_1.u| \rfloor - 1$ and $k' = k - |Y_1.u|\lfloor k/|Y_1.u| \rfloor$. 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 $Y = Y_1^{[i]}$, we can compute the value $Q(Y,k)$ from the values $Q(Y_1,k)$ and i, since $X.u = {}^{[k]}((Y_1.u)^{[i]}) = ({}^{[k]}(Y_1.u))^{[i]}$.

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

Since recursive call of the function $Q(X,k)$ continues at most $height(X)$ times, the value $lpf_\pi(X.u)$ is computed in $O(height(X))$ time. □

Now we are ready to prove Theorem 1.

Proof of Theorem 1. From Lemma 3 and Fact 1, we can get the value of *Jump* by

$$Jump(j,X) = |lsp_\pi(lsf_\pi(\pi[1:j] \cdot X.u))|.$$

From Lemma 7, we can get the value $lsp_\pi(x)$ in O(1) time for any $x \in Factor(\pi)$ after $O(m^2)$ time and space preprocessing. Thus, we can concentrate to compute $lsf_\pi(\pi[1:j] \cdot X.u)$ for any $j \in Q$ and $X \in F(D)$.

Case 1: $|lsf_\pi(X.u)| < |X.u|$. In this case, $lsf_\pi(\pi[1:j] \cdot X.u) = lsf_\pi(X.u)$. From Lemma 8, $lsf_\pi(X.u)$ can be obtained in O(1) time after $O(\|D\| \cdot height(D) + m^2)$ time and $O(\|D\| + m^2)$ space preprocessing.

Case 2: $|lsf_\pi(X.u)| = |X.u|$. Since $X.u$ is a factor of π, we can obtain $lsf_\pi(\pi[1:j] \cdot X.u)$ in O(1) time after $O(\|D\| \cdot height(D) + m^2)$ time and $O(\|D\| + m^2)$ space preprocessing, from Lemmas 6 and 8.

If D contains no truncation, the time complexity in both cases can be reduced to $O(\|D\| + m^2)$ by Lemma 8. The proof is complete. □

5.2. Realization of Output

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

$$Output(j,X) = Occ^{\star}(\pi, \pi[1:j] \bullet X.u) \ominus j \cup Occ(\pi, X.u),$$

First, we consider the subset $Occ^{\star}(\pi, \pi[1:j] \bullet X.u)$.

Lemma 9. *Let x and y be strings. If $Occ^{\star}(\pi, x \bullet y)$ has more than two elements, it forms an arithmetic progression, where the step is the smallest period of π.*

Proof. It follows directly from Lemma 2. □

Lemma 10. *The table that stores $Occ^{\star}(\pi, x \bullet y)$ for all pairs of $x \in Prefix(\pi)$ and $y \in Suffix(\pi)$, 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 9 that $Occ^{\star}(\pi, x \bullet y)$ 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 $Occ^{\star}(\pi, x \bullet y)$ for $x \in Prefix(\pi)$ and $y \in Suffix(\pi)$ can be computed in $O(m^2)$ time as stated in [4]. The construction is as follows:

(1) Fill in with $|y|$ all table locations (x, y) where $|x| + |y| = m$.
(2) For each table column, fill in the rest of the entries, using the failure function of the KMP automaton.

By reversing the pattern π, the table of the maximum values is also computed in $O(m^2)$ time in a similar way. The smallest period of π is computed in $O(m)$ time. □

Lemma 11. *For a collage system $\langle D, S \rangle$ and a pattern π, the procedure that takes as input $X, Y \in F(D)$ and enumerates the set $Occ^{\star}(\pi, X.u \bullet Y.u)$ in $O(|Occ^{\star}(\pi, X.u \bullet Y.u)|)$ time, can be computed in $O(m^2)$ time and space, assuming that $lsf_{\pi}(X.u)$ and $lpf_{\pi}(Y.u)$ are already computed.*

Proof. It is obvious that $Occ^{\star}(\pi, X.u \bullet Y.u) = Occ^{\star}(\pi, lsp_{\pi}(X.u) \bullet lps_{\pi}(Y.u))$. Recall Fact 1, that is, $lsp_{\pi}(X.u) = lsp_{\pi}(lsf_{\pi}(X.u))$ and $lps_{\pi}(Y.u) = lps_{\pi}(lpf_{\pi}(Y.u))$. Then, this lemma follows from Lemmas 7 and 10. □

Lemma 12. *For a collage system $\langle D, S \rangle$ and a pattern π, the procedure that takes as input an integer j, $0 \leqslant j \leqslant m$, and $X \in F(D)$ and enumerates the set $Occ^{\star}(\pi, \pi[1:j] \bullet X.u)$ in $O(|Occ^{\star}(\pi, \pi[1:j] \bullet X.u)|)$ time, can be computed in $O(m^2)$ time and space, assuming that $lpf_{\pi}(X.u)$ is already computed.*

Proof. We can prove it in a similar way of the proof of Lemma 11. □

Next, we consider the subset $Occ(\pi, X.u)$. In what follows, we give the computation of a representation of the sets $Occ(\pi, X.u)$ for the tokens $X \in F(D)$.

Lemma 13. *For a collage system $\langle D, S\rangle$ and a pattern π, we can enumerate the set $Occ(\pi, X.u)$ for $X \in F(D)$ in $O(|Occ(\pi, X.u)|)$ time after $O(m^2)$ time and space preprocessing, assuming that $Occ(\pi, Y.u)$, $lpf_\pi(Y.u)$, and $lsf_\pi(Y.u)$ are already computed for all Y such that $T(Y)$ is the subtree of $T(X)$ in the syntax tree.*

Proof. We prove each form of the assignment below. □

Claim 2. *The lemma holds if $X = YZ$.*

Proof. We have $Occ(\pi, X.u) = Occ(\pi, Y.u) \cup Occ^\star(\pi, Y.u \bullet Z.u) \cup (Occ(\pi, Z.u) \oplus |Y.u|)$. Thus, we can enumerate the set $Occ^\star(\pi, Y.u \bullet Z.u)$ in $O(|Occ^\star(\pi, Y.u \bullet Z.u)|)$ time from Lemma 11. This proves the claim. □

Claim 3. *The lemma holds if $X = Y^k$ with $k > 1$.*

Proof. It is trivial for $k = 2$ from Claim 2. Suppose $k > 2$. Note that we can enumerate the set $Occ^\star(\pi, Y.u \bullet Y.u)$ in $O(|Occ^\star(\pi, Y.u \bullet Y.u)|)$ time by Lemma 11, and that $lps_\pi(Y.u) = lps_\pi(lpf_\pi(Y.u))$ from Fact 1. Now, we have three cases to consider.

Case 1: $|\pi| \leqslant |Y.u|$. Since π cannot cover more than two Y's, it is not hard to see that $Occ(\pi, X.u)$ can be enumerated in $O(|Occ(\pi, X.u)|)$ time using $Occ(\pi, Y.u)$, $Occ^\star(\pi, Y.u \bullet Y.u)$, $|Y.u|$, and k.

Case 2: $|Y.u| < |\pi| < 2|Y.u|$. We compute two sets $Occ^\star(\pi, Y.u \bullet Y.u)$ and $Occ^\star(\pi, Y.u \bullet Y.uY.u) \setminus Occ^\star(\pi, Y.u \bullet Y.u)$. From Lemma 6, we can obtain $lpf_\pi(Y.uY.u)$ from $lpf_\pi(Y.u)$. Then, $Occ^\star(\pi, Y.u \bullet Y.uY.u)$ can be obtained from Lemma 11. Therefore, the set $Occ(\pi, X.u)$ can be enumerated in $O(|Occ(\pi, X.u)|)$ time using these sets, $|Y.u|$ and k.

Case 3: $2|Y.u| \leqslant |\pi|$. Note that π occurs within $(Y.u)^\ell$ for some $\ell > 0$ if and only if (1) $Y.u$ is a factor of π, and (2) $|Y.u|$ is a period of π. The first condition is satisfied when $|Y.u| = |lpf_\pi(Y.u)|$. The second condition is satisfied when $|Y.u|$ is a multiple of the smallest period t of π (recall Lemma 1). The set $Occ(\pi, X.u)$ forms an arithmetic progression, whose step is t. Thus, it can be enumerated in $O(|Occ(\pi, X.u)|)$ time. □

Claim 4. *The lemma holds if $X = Y^{[k]}$ (resp. $X = {}^{[k]}Y$) with $1 \leqslant k \leqslant |Y.u|$.*

Proof. This holds obviously since we have $Occ(\pi, X.u) = \{i \mid i \in Occ(\pi, Y.u), i \leqslant |Y.u| - k\}$ (resp. $Occ(\pi, X.u) = \{i - k \mid i \in Occ(\pi, Y.u), i > k\}$). □

Proof of the lemma. It is easy to prove if $X = a$. From this and above claims, the lemma holds for any form of the assignment of X. □

Although we need $lpf_\pi(X.u)$ and $lsf_\pi(X.u)$ for $X \in F(D)$, these are computed in $O(\|D\| \cdot height(D) + m^2)$ time using $O(\|D\| + m^2)$ space, from Lemma 8. The next lemma follows.

Lemma 14. *We can build in* $O(\|D\| \cdot height(D) + m^2)$ *time using* $O(\|D\| + m^2)$ *space a data structure by which the enumeration of the set* $Occ(\pi, X.u)$ *is performed in* $O(height(X) + \ell)$ *time, where* $\ell = |Occ(\pi, X.u)|$. *If* D *contains no truncation, it can be built in* $O(\|D\| + m^2)$ *time and space, and the enumeration requires only* $O(\ell)$ *time.*

Proof. Recall the syntax trees defined in Section 3. A node labeled X of the syntax tree is said to be *active* if (1) it does not have a child labeled Y such that either $Occ(\pi, X.u) = Occ(\pi, Y.u)$, or (2) it is a leaf node and $Occ(\pi, X.u) \neq \emptyset$. The equality testing of the sets is replaced by the equality testing of their cardinalities, since it holds that either $Occ(\pi, X.u) \supseteq Occ(\pi, Y.u) \oplus k$ for concatenation and repetition, or $Occ(\pi, X.u) \subseteq Occ(\pi, Y.u) \oplus k$ for truncation, where k is an appropriate offset.

From Lemma 13, it is not difficult to show that the table $Card(X)$ which stores the cardinalities of $Occ(\pi, X.u)$ for all tokens $X \in F(D)$, can be computed in $O(\|D\| \cdot height(D) + m^2)$ time using $O(\|D\| + m^2)$ space. If D contains no truncation, it can be computed in $O(\|D\| + m^2)$ time and space.

Next, using the table $Card$, we add, for each node v labeled 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 $Occ(\pi, X.u)$ for a token X. When there is no truncation, we have only to traverse the syntax tree $T(X)$ 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 violates the condition. Namely, for an inner node labeled Y in the syntax tree $T(X)$ such that Y is a k-suffix truncation, we enumerate $Occ(\pi, Y.u)$ in ascending order by utilizing the short-cut pointers unless its element exceeds $|Y.u| - k$. When we encounter a k-prefix truncation in the traversal of $T(X)$, a kind of binary search will navigate us in $O(height(X))$ time to the first position of the occurrence in its subtree. Namely, we do depth-first traversal of $T(X)$ by utilizing the short-cut pointers with calculating the offset of its subtrees, and find the leftmost and nearest descendant Y of X such that the offset of $T(Y)$ is equal or greater than k. Then we resume enumerating. □

We are now ready to prove Theorem 2.

Proof of Theorem 2. It follows from Lemmas 12 and 14. □

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 $O((\|D\| + |S|) \cdot height(D) + m^2 + r)$ time with $O(\|D\| + m^2)$ space. The factor

height(D) can be dropped if the collage system contains no truncation. It coincides with the observation by Navarro and Raffinot [29] 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 [29]. 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 [20]. That is, we simulate the move of the AC pattern matching machine instead of the KMP automaton, and use the generalized suffix trie [15]. For *Output*, we have also done if a collage system contains no repetitions. The rest is left for our future work.

For the approximate string matching problem, Kärkkäinen et al. [16] presented an algorithm which runs in $O(mkn + r)$ time on LZ78 and LZW, where k is the number of allowed errors and r is the number of the pattern occurrences. In [26], we proposed an approximate string matching algorithm on a *simple collage system*, which is a subclass of the collage system and covers the LZ78 and LZW compression methods. The algorithm runs in $O(k^2(\|D\| + |S|) + km)$ time using $O(k^2\|D\|)$ space.

In [33], we proposed the Boyer–Moore-type algorithm which runs on a collage system. The algorithm runs in $O(\|D\| + |S| \cdot m + m^2 + \ell)$ time using $O(\|D\| + m^2)$ space if D contains no truncation. We also showed that the algorithm specialized to the byte pair encoding is very fast in practice. In fact it runs about 1.2–3.0 times faster than the exact match routine of the software package `Agrep` [37], known as the fastest pattern matching tool. This means that text compression can accelerate pattern matching.

Kosaraju [24] 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, Efficient two-dimensional compressed matching, in: Proc. Data Compression Conference, 1992, p. 279.
[2] A. Amir, G. Benson, Two-dimensional periodicity and its application, in: Proc. of the 3rd Ann. ACM-SIAM Symp. on Discrete Algorithms, 1992, pp. 440–452.
[3] A. Amir, G. Benson, M. Farach, Optimal two-dimensional compressed matching, in: Proc. 21st Internat. Colloq. on Automata, Languages and Programming, 1994, pp. 215–226.
[4] A. Amir, G. Benson, M. Farach, Let sleeping files lie: pattern matching in Z-compressed files, J. Comput. System Sci. 52 (1996) 299–307.
[5] A. Amir, G.M. Landau, U. Vishkin, Efficient pattern matching with scaling, J. Algorithms 13 (1) (1992) 2–32.
[6] A. Apostolico, Z. Galil, Pattern Matching Algorithm, Oxford University Press, New York, 1997.
[7] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, New York, 1994.
[8] E.S. de Moura, G. Navarro, N. Ziviani, R. Baeza-Yates, Direct pattern matching on compressed text, in: Proc. 5th Internat. Symp. on String Processing and Information Retrieval, IEEE Computer Society, Silver Spring, MD, 1998, pp. 90–95.

[9] E.S. de Moura, G. Navarro, N. Ziviani, R. Baeza-Yates, Fast searching on compressed texts allowing errors, in: Proc. 21st Ann. Internat. ACM SIGIR Conf. on Research and Development in Information Retrieval, ACM Press, New York, 1998, pp. 298–306.
[10] T. Eilam-Tzoreff, U. Vishkin, Matching patterns in strings subject to multi-linear transformations, Theoret. Comput. Sci. 60 (3) (1988) 231–254.
[11] M. Farach, M. Thorup, String-matching in Lempel–Ziv compressed strings, in: 27th ACM STOC, 1995, pp. 703–713.
[12] L. Gąsieniec, M. Karpinski, W. Plandowski, W. Rytter, Efficient algorithms for Lempel–Ziv encoding, in: Proc. 4th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science, Vol. 1097, Springer, Berlin, 1996, pp. 392–403.
[13] L. Gasieniec, M. Karpinski, W. Plandowski, W. Rytter, Efficient algorithms for Lempel–Ziv encoding, in: Proc. 5th Scandinavian Workshop on Algorithm Theory, 1996, pp. 392–403.
[14] R. Giegerich, S. Kurtz, From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree construction, Algorithmica 19 (3) (1997) 331–353.
[15] L.C.K. Hui, Color set size problem with application to string matching, in: Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 644, Springer, Berlin, 1992, pp. 230–243.
[16] J. Kärkkäinen, G. Navarro, E. Ukkonen, Approximate string matching over Ziv-Lempel compressed text, in: Proc. 11th Ann. Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 1848, Springer, Berlin, 2000, pp. 195–209.
[17] M. Karpinski, W. Rytter, A. Shinohara, An efficient pattern-matching algorithm for strings with short descriptions, Nordic J. Comput. 4 (1997) 172–186.
[18] T. Kida, M. Takeda, A. Shinohara, 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, Vol. 1645, Springer, Berlin, 1999, pp. 1–13.
[19] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, S. Arikawa, Multiple pattern matching in LZW compressed text, in: J.A. Storer, M. Cohn (Eds.), Proc. Data Compression Conf. '98, IEEE Computer Society, Silver Spring, MD, 1998, pp. 103–112.
[20] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, S. Arikawa, Multiple pattern matching in LZW compressed text, J. Discrete Algorithms 1 (1) (2000) 133–158 (previous version in DCC'98 and CPM'99).
[21] J.C. Kieffer, E. Yang, Grammar-based codes: a new class of universal lossless source codes, IEEE Trans. Inform. Theory 46 (3) (2000) 737–754.
[22] J.C. Kieffer, E. Yang, G.J. Nelson, P. Cosman, Universal lossless compression via multilevel pattern matching, IEEE Trans. Inform. Theory 46 (4) (2000) 1227–1245.
[23] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[24] S. Kosaraju, Pattern matching in compressed texts, in: Proc. Foundation of Software Technology and Theoretical Computer Science, Springer, Berlin, 1995, pp. 349–362.
[25] N.J. Larsson, A. Moffat, Offline dictionary-based compression, in: Proc. Data Compression Conf. '99, IEEE Computer Society, Silver Spring, MD, 1999, pp. 296–305.
[26] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, S. Arikawa, Bit-parallel approach to approximate string matching in compressed texts, in: Proc. 7th Internat. Symp. on String Processing and Information Retrieval, IEEE Computer Society, Silver Spring, MD, 2000, pp. 221–228.
[27] M. Miyazaki, A. Shinohara, M. Takeda, An improved pattern matching algorithm for strings in terms of straight-line programs, in: Proc. 8th Ann. Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 1264, Springer, Berlin, 1997, pp. 1–11.
[28] M. Miyazaki, A. Shinohara, M. Takeda, An improved pattern matching algorithm for strings in terms of straight-line programs, J. Discrete Algorithms 1 (1) (2000) 187–204 (previous version in CPM'97).
[29] G. Navarro, 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, Vol. 1645, Springer, Berlin, 1999, pp. 14–36.
[30] M. Nelson, The Data Compression Book, M&T Publishing, Inc., Redwood City, CA, 1992.
[31] C.G. Nevill-Manning, I.H. Witten, D.L. Maulsby, Compression by induction of hierarchical grammars, in: Proc. Data Compression Conf. '94, IEEE Press, New York, 1994, pp. 244–253.

[32] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, S. Arikawa, Speeding up pattern matching by text compression, in: Proc. 4th Italian Conf. on Algorithms and Complexity, Lecture Notes in Computer Science, Vol. 1767, Springer, Berlin, 2000, pp. 306–315.
[33] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, S. Arikawa, A Boyer–Moore type algorithm for compressed pattern matching, in: Proc. 11th Ann. Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 1848, Springer, Berlin, 2000, pp. 181–194.
[34] Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa, Pattern matching in text compressed by using antidictionaries, in: Proc. 10th Ann. Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 1645, Springer, Berlin, 1999, pp. 37–49.
[35] J. Storer, T. Szymanski, Data compression via textual substitution, J. Assoc. Comput. Mach. 29 (4) (1982) 928–951.
[36] T.A. Welch, A technique for high performance data compression, IEEE Comput. 17 (1984) 8–19.
[37] S. Wu, U. Manber, Agrep—a fast approximate pattern-matching tool, in: Usenix Winter 1992 Technical Conference, 1992, pp. 153–162.
[38] E. Yang, J.C. Kieffer, Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform, IEEE Trans. Inform. Theory 46 (3) (2000) 755–777.
[39] J. Ziv, A. Lempel, Compression of individual sequences via variable-length coding, IEEE Trans. Inform. Theory 24 (5) (1978) 530–536.
[40] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory IT-23 (3) (1997) 337–349.