More Speed and More Compression: Accelerating Pattern Matching by Text Compression

Matsumoto, Tetsuya
Department of Informatics, Kyushu University

Hagio, Kazuhito
Department of Informatics, Kyushu University

Takeda, Masayuki
Department of Informatics, Kyushu University

出版情報：DOI Technical Report. 232, 2007-11. Department of Informatics, Kyushu University
バージョン：
権利関係：

More Speed and More Compression: Accelerating Pattern Matching by Text Compression

Tetsuya Matsumoto, Kazuhito Hagio, and Masayuki Takeda

Department of Informatics, Kyushu University, Fukuoka 819-0395, Japan
{tetsuya.matsumoto, kazuhito.hagio, takeda }@i.kyushu-u.ac.jp

Abstract. This paper addresses the problem of speeding up string matching by text compression, and presents a compressed pattern matching (CPM) algorithm which finds a pattern within a text given as a collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ such that variable sequence $\mathcal{S}$ is encoded by byte-oriented Huffman coding. The compression ratio is high compared with existing CPM algorithms addressing the problem, and the search time reduction ratio compared to the Knuth-Morris-Pratt algorithm over uncompressed text is nearly the same as the compression ratio.

1 Introduction

The *compressed pattern matching* (CPM in short) problem is to find occurrences of a pattern in a compressed text *without decompressing it*. One goal of this problem is to perform a faster search in a compressed text in comparison with decompression followed by an ordinary search (**Goal 1**).

A more ambitious goal is to perform a faster search in a compressed text in comparison with an ordinary search in the original text (**Goal 2**). We note that in the setting of Goal 1, users compress their text data using their favored compression methods for the traditional purpose, i.e., saving disk storage or data transmission cost. On the contrary, the aim of compression is not only to reduce disk storage requirement but also to speed up string searching in the setting of Goal 2. Let t_d, t_u, and t_c be the times for decompression, for searching in uncompressed text, and for searching in compressed text, respectively. Goal 2 aims for $t_u > t_c$ while Goal 1 does for $t_d + t_u > t_c$. Goal 2 is thus more difficult to achieve than Goal 1, and most Goal 2-oriented researches involve in designing a new compression scheme appropriate for it, rather than using existing scheme with high compression ratio.

We introduced in [3] a useful CPM-oriented abstraction of compression formats, named *collage systems*, where a text is represented as a pair of dictionary $\mathcal{D}$ and sequence $\mathcal{S}$ of variables in $\mathcal{D}$. Algorithms designed for collage systems can be implemented for many different compression formats. We designed in the same paper a general Knuth-Morris-Pratt (KMP) type algorithm on collage systems, and in [14] a general Boyer-Moore (BM) type algorithm on collage systems. Then we specialized in [13, 14] the algorithms for the byte-pair encoding (BPE) [2]. The obtained algorithms are, respectively, faster than their original

algorithms over uncompressed text. BPE compression thus accelerates string matching, but its compression ratio is poor. In this paper we try to improve both the compression ratio and the search time reduction ratio.

Main contributions. We consider a general compression scheme which transforms texts to truncation-free collage systems and encodes them using byte-oriented Huffman coding. The dictionary size can be tuned by a parameter n, the number of internal nodes of the Huffman tree. The compression ratio can then be much better than those of the compression schemes used by existing Goal 2-oriented CPM algorithms. Then we develop a new KMP type CPM algorithm which runs $O(|\mathcal{S}| + occ)$ time after $O(|\mathcal{D}| \cdot |P|)$ time and space preprocessing, where P is a pattern and occ is the number of pattern occurrences. We note that the bound $O(|\mathcal{D}| \cdot |P|)$ improves the preprocessing time and space bound $O(|\mathcal{D}| \cdot |P| + |P|^2)$ of straightforward application of [3]. The proposed algorithm runs practically fast and the search time reduction compared to uncompressed KMP matching is almost the same as the compression ratio.

Related work. There are two lines in CPM studies addressing Goal 2. One is to put restriction on compression scheme so that every pattern occurrence can be identified simply as a substring of encoded text that is identical to encoded pattern. The advantage is that any favored pattern matching algorithm can be used to search encoded text for encoded pattern. The works of Manber [7] and Rautio et al. [12] are along this line. The drawback is that the restriction sacrifices the compression ratio. The work of Moura et al. [9] uses a word-based Huffman encoding with a byte-oriented code, and shows a high compression ratio. However it is limited to word-based search. The other line is to develop CPM algorithms for coping with compression scheme in which some occurrences of encoded pattern can be false matches, and/or pattern possibly occurs in several different forms within encoded text. The work of Miyazaki et al. [8], our previous works [13, 14] for BPE, and the present paper are along this line. While all of the works [7, 12, 8, 13, 14] mentioned here achieve Goal 2, the compression ratios are poor: BPE is the best among them.

2 Preliminaries

Let Σ be a finite alphabet. An element of Σ^* is called *string*. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string $s = xyz$, respectively. A prefix, factor, and suffix of a string s is said to be *proper* if it is not s. The length of a string s is denoted by $|s|$. The ith symbol of a string s is denoted by $s[i]$ for $1 \leq i \leq |s|$, and the factor of s that begins at position i and ends at position j is denoted by $s[i..j]$ for $1 \leq i \leq j \leq |s|$. Denote by ${}^{[i]}s$ (resp. $s^{[i]}$) the string obtained from s by removing the length i prefix (resp. suffix) for $0 \leq i \leq |s|$. The concatenation of i copies of the same string s is denoted by s^i. For a set A of integers and an integer k, let $A \oplus k = \{i + k \mid i \in A\}$.

For strings x, y, and z, let $Occ_x(y) = \left\{|v| \;\middle|\; \exists u, \exists v : y = uxv\right\}$ and $Occ^\star_x(y, z) = \left\{d \;\middle|\; d \in Occ_x(yz) \wedge |x| + d > |z| > d\right\}$. Note that $Occ^\star_x(y, z)$ is the set of occurrences of x within yz which cover the boundary between y and z.

A *period* of a string s is an integer p, $0 < p \leq |s|$, such that $s[i] = s[i+p]$ for all $i \in \{1, \ldots, |s| - p\}$. The next lemma follows from the periodicity lemma [1].

Lemma 1 ([3]). *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 whose step is the smallest period of x.*

Corollary 1. *$Occ^{\star}_x(y, z)$ forms an arithmetic progression for any strings x, y, z.*

3 Collage systems and compressed pattern matching

3.1 Collage systems

A *collage system* is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ defined as follows. $\mathcal{D}$ is a sequence of assignments $X_1 = expr_1;\ X_2 = expr_2;\ \cdots\ ; X_n = expr_n$, where, for each $k = 1, \ldots, n$, 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 a positive integer j,	(j *length prefix truncation*)
$X_i^{[j]}$	for $i < k$ and a positive integer j,	(j *length suffix truncation*)
$(X_i)^j$	for $i < k$ and a positive integer j.	(j *times repetition*)

The *size* of $\mathcal{D}$ is the number n of assignments and denoted by $|\mathcal{D}|$. The *height* of $\mathcal{D}$ is the height of the dependence tree of $\mathcal{D}$ and denoted by $h(\mathcal{D})$. $\mathcal{S}$ is a sequence $X_{i_1} \cdots X_{i_\ell}$ of variables defined in $\mathcal{D}$. The *length* of $\mathcal{S}$ is the number ℓ of variables and denoted by $|\mathcal{S}|$. The variables X_k represent the strings $\overline{X_k}$ obtained by evaluating their expressions. A collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ represents the string obtained by concatenating the strings $\overline{X_{i_1}}, \ldots, \overline{X_{i_\ell}}$ represented by variables $X_{i_1}, \ldots, X_{i_\ell}$ of $\mathcal{S}$.

A collage system is said to be *truncation-free* if $\mathcal{D}$ contains no truncation operation, and *regular* if $\mathcal{D}$ contains neither repetition nor truncation operation. For example, the collage systems for the run-length encoding is truncation-free, and those for RE-PAIR [6], SEQUITUR [11], BPE [2], and the grammar-transform based compression [5] are regular. In the Lempel-Ziv family, the collage systems for LZ78/LZW are regular, while those for LZ77/LZSS are not truncation-free.

3.2 CPM algorithm over collage systems

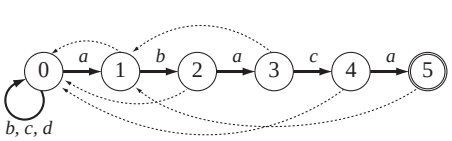
The algorithm of [3] has two stages: First it preprocesses $\mathcal{D}$ and P, and second it processes the variables of $\mathcal{S}$. In the second stage, it simulates the KMP automaton on uncompressed text, by using two functions *Jump* and *Output*, both take as input a state q and a variable X. The former is used to substitute just one state transition for the consecutive state transitions of the KMP automaton for the string $\overline{X}$ for each variable X of $\mathcal{S}$, and the latter is used to report all pattern occurrences found during the state transitions. Thus the two functions form a Mealy-type finite-state transducer. Let us call it *compressed pattern matching*

machine (CPMM in short). Formally, let δ be the state-transition function of the KMP automaton. The definition of the functions is as follows.

$$Jump(q, X) = \delta(q, \overline{X})$$

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

An example of CPMM and its move are displayed in Fig. 1 and 2, respectively.



$\delta(q, s)$

	a	b	c	d
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	2	4	0
4	5	0	0	0
5	1	2	0	0

$\mathcal{D}$

$X_1 = a$
$X_2 = b$
$X_3 = c$
$X_4 = d$
$X_5 = X_2X_1$
$X_6 = X_3X_1$
$X_7 = X_6X_1$
$X_8 = X_1X_5$
$X_9 = X_5X_6$
$X_{10} = X_6X_9$

$Jump(q, X_i)/Output(q, X_i)$

state	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
0	1/∅	0/∅	0/∅	0/∅	1/∅	1/∅	1/∅	3/∅	1/∅	5/{0}
1	1/∅	2/∅	0/∅	0/∅	3/∅	1/∅	1/∅	3/∅	5/{0}	5/{0}
2	3/∅	0/∅	0/∅	0/∅	1/∅	1/∅	1/∅	3/∅	1/∅	5/{0}
3	1/∅	2/∅	4/∅	0/∅	3/∅	5/{0}	1/{1}	3/∅	5/{0}	5/{4, 0}
4	5/{0}	0/∅	0/∅	0/∅	1/∅	1/∅	1/∅	3/{2}	1/∅	5/{0}
5	1/∅	2/∅	0/∅	0/∅	3/∅	1/∅	1/∅	3/∅	5/{0}	5/{0}

Fig. 1. KMP automaton for $P = abaca$ is shown on the upper-left, where the state-transition function δ is represented by the goto and the failure functions (depicted by the solid and the broken arrows, respectively), and the deterministic version of δ is displayed on the upper-right. The functions *Jump* and *Output* built from the KMP automaton for the dictionary $\mathcal{D}$ shown on the lower-left, are shown on the lower-right.

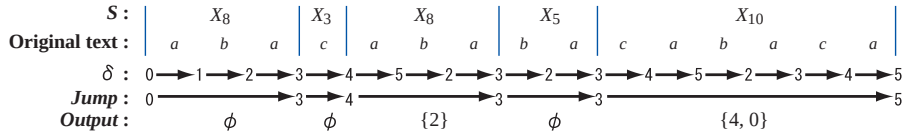


Fig. 2. Move of CPMM of Fig. 1 against $\mathcal{S} = X_8X_3X_8X_5X_{10}$ is demonstrated.

Lemma 2 ([3]). *The function Jump can be built in $O(|\mathcal{D}| \cdot h(\mathcal{D}) + |P|^2)$ time using $O(|\mathcal{D}| + |P|^2)$ space, so that it responds in constant time. The factor $h(\mathcal{D})$ disappears if $\mathcal{D}$ is truncation-free.*

Lemma 3 ([3]). *The function Output can be built in $O(|\mathcal{D}| \cdot h(\mathcal{D}) + |P|^2)$ time using $O(|\mathcal{D}| + |P|^2)$ space, so that it responds in $O(h(\mathcal{D}) + \ell)$ time, where ℓ is the answer size. The factor $h(\mathcal{D})$ disappears if $\mathcal{D}$ is truncation-free.*

Theorem 1 ([3]). *The CPM problem can be solved in $O\big((|\mathcal{D}| + |\mathcal{S}|) \cdot h(\mathcal{D}) + |P|^2 + occ\big)$ time using $O(|\mathcal{D}| + |P|^2)$ space, where occ is the number of pattern occurrences. The factor $h(\mathcal{D})$ is dropped for truncation-free collage systems.*

3.3 Practical aspect

Theorem 1 suggests that compression schemes which describe texts in terms of truncation-free collage systems might be suitable for CPM. For example, it implies that LZW (truncation-free) is suitable compared with LZ77 (not truncation-free). This coincides with the observation by Navarro and Raffinot [10] that CPM for LZW achieves Goal 1, whereas CPM for LZ77 does not so.

From practical CPM viewpoints, we have to pay attention to the way collage systems $\langle \mathcal{D}, \mathcal{S} \rangle$ are encoded. Desirable conditions of compression scheme other than the truncation-freeness can be summarized as follows [14].

- $\mathcal{D}$ is encoded separately from $\mathcal{S}$.
- $\mathcal{D}$ is so small to fit the two-dimensional array implementation of *Jump*.
- The variables of $\mathcal{S}$ can be decoded without bit-level computation: A byte-oriented code is preferable.

LZW satisfies none of these conditions, and it was observed in [4] that CPM for LZW is too slow to achieve Goal 2 though it achieves Goal 1. On the other hand, BPE [2] satisfies all of them. It transforms a text into a regular collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ such that $|\mathcal{D}| \leq 256$ and each variable of $\mathcal{S}$ is encoded in one byte. It is shown in [13] that BPE reduces the searching time at nearly the same ratio as the compression ratio, compared with KMP algorithm over original text.

The compression ratio of BPE is, however, poor. In the next section, we raise the restriction on $|\mathcal{D}|$ and encode $\mathcal{S}$ with byte-oriented Huffman coding, in order to improve the compression ratio and keep satisfying the above conditions.

4 New Method

4.1 Basic idea

We encode the variables of $\mathcal{S}$ by byte-oriented Huffman coding, that is, we use a prefix code Φ that maps the variables of $\mathcal{D}$ to strings over the byte alphabet $\Gamma = \{\texttt{00}, \ldots, \texttt{FF}\}$. Although we are interested only in the case that both the source alphabet Σ and the coding alphabet Γ are of size 256, we illustrate our method using an example with $\Sigma = \{a, b, c, d\}$ and $\Gamma = \{A, B, C, D\}$ for the sake of simplicity in explanation.

Fig. 3 shows a prefix code Φ that maps variables $X_1, \ldots, X_{10}$ to strings over $\Gamma = \{A, B, C, D\}$ and its code tree. Using this code, the variable sequence $\mathcal{S} = X_8X_3X_8X_5X_{10}$ of Fig. 2 is encoded as $AD\ D\ AD\ AB\ AAC$. Input to our problem is thus a string over Γ representing $\mathcal{S}$.

One naive solution to the problem would be to use the code tree as a decoder and to apply the CPM technique discussed in the previous section to the decoded variables. However, the processing is not very fast in practice.

The method we propose in this paper is to embed the decoder (code tree) into the CPMM. That is, we replace every transition by X_i from state s to state t of CPMM with a consecutive transitions from s to t that spells out the codeword of X_i. The decoder-embedded CPMM (DCPMM for short) is thus a Mealy-type

finite-state transducer with state-transition function Δ and output function Λ. In the running example, the transition $\mathit{Jump}(3, X_{10}) = 5$ of CPMM is replaced with the transitions $\Delta(3, A) = 9$, $\Delta(9, A) = 15$, and $\Delta(15, C) = 5$, where 9 and 15 are newly introduced states. All but last transitions have no outputs, that is, $\Lambda(3, A) = \Lambda(9, A) = \emptyset$, and $\Lambda(15, C) = \mathit{Output}(3, X_{10})$. Fig. 4 displays DCPMM.

4.2 Definition

A formal definition follows. Let Γ be a coding alphabet and Φ be a prefix code that maps the variables in $\mathcal{D}$ to strings over Γ. The *code tree* of Φ, denoted by $\mathcal{T}_\Phi$, is a trie representing the set of codewords of Φ such that (1) the leaves are one-to-one associated with variables in $\mathcal{D}$, and (2) the path from the root to the leaf associated with X_i spells out the codeword $\Phi(X_i)$.

Let I be the set of internal nodes of the code tree $\mathcal{T}_\Phi$. Let $\tilde{Q} = Q \times I$, which is the set of states of DCPMM. The state transition function Δ and the output function Λ of DCPMM are defined on $\tilde{Q} \times \Gamma$. Let $((j, t), \gamma)$ be any element in $\tilde{Q} \times \Gamma$. Let t' be the child of t such that the edge (t, t') is labeled $\gamma \in \Gamma$ in $\mathcal{T}_\Phi$. The definitions of Δ and Λ are as follows.

- If t' is a leaf of $\mathcal{T}_\Phi$ associated with variable X_i, then $\Delta((j, t), \gamma) = \mathit{Jump}(j, X_i)$ and $\Lambda((j, t), \gamma) = \mathit{Output}(j, X_i)$.
- If t' is an internal node of $\mathcal{T}_\Phi$, then $\Delta((j, t), \gamma) = (j, t')$ and $\Lambda((j, t), \gamma) = \emptyset$.

In the sequel we assume code trees are *full*, that is, every internal node has exactly $|\Gamma|$ children. Then a code tree with n internal nodes has exactly $(|\Gamma| - 1)n + 1$ leaves, and hence $|\mathcal{D}| = (|\Gamma| - 1)n + 1$. A two-dimensional table J storing the values of *Jump* is of size $|Q| \times |\mathcal{D}| = (|P| + 1)(255n + 1)$ and a two-dimensional table storing Δ is of size $|\tilde{Q}| \times |\Gamma| = (|P| + 1) \cdot 256n$. The fraction is $\frac{256n}{255n+1} = \frac{256}{255}(1 - \frac{1}{255n+1}) \leq \frac{256}{255}$. Thus, the size of the table storing Δ is almost the same as the table J.

4.3 Storing dictionary and code tree

A code tree with n internal nodes can be represented by the bit-string obtained during a preorder traversal over it such that every internal node is represented

Code Φ

variable	codeword	variable	codeword
X_1	B	X_6	AC
X_2	C	X_7	AAA
X_3	D	X_8	AD
X_4	AAD	X_9	AAB
X_5	AB	X_{10}	AAC

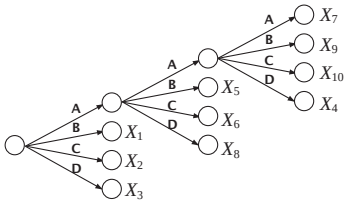


Fig. 3. A prefix code Φ that maps the variables $X_1, \ldots, X_{10}$ to strings over $\Gamma = \{A, B, C, D\}$ is shown on the left, and its code tree is displayed on the right.

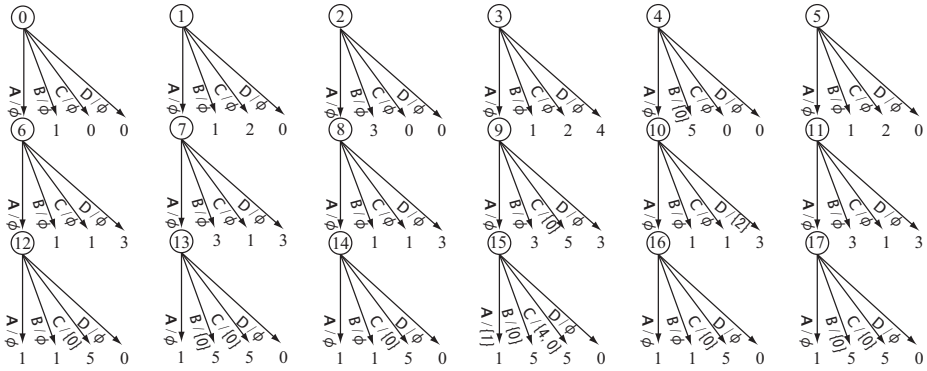


Fig. 4. DCPMM is displayed, where the numbers in circles represent the states, and the numbers not in circles imply the states with the same numbers. The path consisting of the edges $(3, 9)$, $(9, 15)$, and $(15, 5)$ which are labeled $A/\emptyset$, $A/\emptyset$, and $C/\{4, 0\}$, respectively, implies that $Jump(3, X_{10}) = 5$ and $Output(3, X_{10}) = \{4, 0\}$. The number of states of DCPMM is 18, which is 3 times larger than the original CPMM of Fig. 1 as the number of internal nodes of the code tree of Fig. 3 is 3.

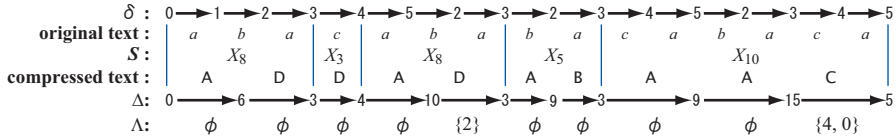


Fig. 5. Move of DCPMM for input $ADDADABAAC$ is demonstrated.

by '0' and every leaf is represented by '1' followed by a $\lceil \log_2 |\mathcal{D}| \rceil$-bit integer indicating the corresponding variable, and thus stored in $|\mathcal{D}|(\lceil \log_2 |\mathcal{D}| \rceil + 1) + n$ bits. It suffices to store the right-hand-sides (RHSs) of the assignments for storing $\mathcal{D}$. Since the RHSs of primitive assignments can be omitted, we have only to store the RHSs of concatenation and repetition assignments. A dictionary $\mathcal{D}$ can be stored as $(|\mathcal{D}| - |\Sigma|)$-pairs of $\lceil \log_2 |\mathcal{D}| \rceil$-bits integers, where we assume that $k \leq |\mathcal{D}|$ for every repetition assignment $X = Y^k$. Table 1 shows space requirement for dictionary and code tree for $|\Sigma| = |\Gamma| = 256$.

Table 1. Space requirement for dictionary and code tree are given, where $|\Sigma| = |\Gamma| = 256$. We note that the total size 299414 bits for $n = 30$ is approximately 37 Kbytes, which is much smaller than text size we are expecting.

	$n = 2$	$n = 5$	$n = 10$	$n = 15$	$n = 20$	$n = 25$	$n = 30$
$\lvert\mathcal{D}\rvert = 255n + 1$	511	1276	2551	3826	5101	6376	7651
dictionary (in bits)	4590	22440	55080	85680	125970	159120	192270
code tree (in bits)	5112	15317	33173	49753	71434	89289	107144
total size (in bits)	9702	37757	88253	135433	197404	248409	299414

5 Algorithm

Given a collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ and a code Φ, our algorithm first builds CPMM consisting of *Jump* and *Output*, and then convert it into DCPMM according to Φ. The conversion is rather straightforward: If we have built a table J of size $|Q| \times |\mathcal{D}|$ storing *Jump*, we can construct a table of size $|\tilde{Q}| \times |\Gamma|$ storing Δ in $O(|\tilde{Q}| \cdot |\Gamma|)$ time and space. Similarly, if we have built a data structure storing *Output*, then we can construct a data structure for Λ in $O(|\tilde{Q}| \cdot |\Gamma|)$ time and space. Thus, we shall concentrate on construction of the table J for *Jump* and the data structure for *Output* below.

5.1 Direct construction of two-dimensional array storing *Jump*

Consider to build a two-dimensional array J that stores the values of *Jump*, namely, $J[q, X] = \mathit{Jump}(q, X)$ for any $q \in Q = \{0, \ldots, |P|\}$ and any variable X defined in $\mathcal{D}$. A straightforward application of the algorithm of [3] requires $O(|\mathcal{D}| \cdot |P| + |P|^2)$ time and space, where the $O(|P|^2)$ factor is needed for solving a subproblem named "factor concatenation problem" [3]. Since we use the two-dimensional array J, we can avoid this problem and take a more direct way. Below we reduce the time and space complexity to $O(|\mathcal{D}| \cdot |P|)$.

It is easy to build J for regular collage systems: first we set $J[q, X] = \delta(q, a)$ for every primitive assignment $X = a$ and for any $q \in Q$, and then set $J[q, X] = J[J[q, Y], Z]$ for every concatenation assignment $X = YZ$ and for any $q \in Q$, assuming the entries $J[p, Y]$ and $J[p, Z]$ are already filled for any $p \in Q$.

Now, we consider the case where repetition assignments $X = Y^k$ $(k \geq 3)$ occur in $\mathcal{D}$. We can fill the entries $J[q, X]$ for all $q \in Q$ in the following fashion.

(1) Initialize $J[q, X]$ by *nil* for all $q \in Q$.
(2) Compute the value $\delta(0, \overline{X})$, and set $J[0, X]$ to it.
(3) For each $q \in Q$ such that $\overline{X}$ occurs at position q of P, set $J[q, X] = q + |\overline{X}|$.
(4) For each $q \in Q$ in increasing order, set $J[q, X] = J[f(q), X]$, if $J[q, X] = \mathit{nil}$, where f is the failure function of the KMP automaton, i.e., $f(q)$ is the length of longest prefix of P that is also a proper suffix of $P[1..q]$.

For the computation of $\delta(0, \overline{X})$ in (2), we set $p := 0$, repeat $p := J[p, Y]$ exactly ℓ-times, and then set $J[0, X] := p$, where ℓ is k, if $k|\overline{Y}| \leq |P|$; and ℓ is the smallest integer such that $|P| \leq \ell|\overline{Y}|$, otherwise. This takes only $O(|P|)$ time since $\ell \leq |P|+1$. For (3), let $\mathit{rep}(q)$ denote the largest integer $\ell \geq 0$ such that the string $\overline{Y}^\ell$ occurs at position q of P. We assume $\mathit{rep}(q) = 0$ for q not in Q. Then the string $\overline{X}$ occurs at position q if and only if $\mathit{rep}(q) \geq k$. We note that $\mathit{rep}(q)$ is at most one larger than $\mathit{rep}(q + |\overline{Y}|)$, and $\mathit{rep}(q) = \mathit{rep}(q + |\overline{Y}|) + 1$ if and only if $\overline{Y}$ occurs at position q (equivalently $J[q, Y] = q + |\overline{Y}|$). This observation tells us that the entries $J[q, X]$ for all $q \in Q$ can be filled in $O(|P|)$ time.

Throughout the construction of the array J, the extra space we need is a one-dimensional array of size $|\mathcal{D}|$ that stores $|\overline{X}|$ for all variables X in $\mathcal{D}$.

Lemma 4. *A two-dimensional array J which stores the function Jump can be built in $O(|P| \cdot |\mathcal{D}|)$ time and space, if $\mathcal{D}$ is truncation-free.*

5.2 Construction of *Output*

Consider to build a data structure which takes $q \in Q$ and variable X in $\mathcal{D}$ as input and returns the set $Output(q, X)$ in linear time w.r.t. its size. A straightforward application of the algorithm of [3] again requires $O(|\mathcal{D}| \cdot |P| + |P|^2)$ time and space. Below we reduce this to $O(|\mathcal{D}| \cdot |P|)$.

We have

$$Output(q, X) = Occ^{\star}_P(P[1..q], \overline{X}) \cup Occ_P(\overline{X}).$$

By Corollary 1, $Occ^{\star}_P(P[1..q], \overline{X})$ forms an arithmetic progression for any q and any X and therefore it can be represented as a triple of integers. Hence the sets $Occ^{\star}_P(P[1..q], \overline{X})$ for all possible combinations of q and X can be stored in $O(|P| \cdot |\mathcal{D}|)$ space. Computation of $Occ^{\star}_P(P[1..q], \overline{X})$ is as follows.

When $X = a$, $Occ^{\star}_P(P[1..q], \overline{X})$ is $\{0\}$ if $|P| > 1$, $q = |P| - 1$, and $P[|P|] = a$; and $\emptyset$ otherwise.

When $X = YZ$, $Occ^{\star}_P(P[1..q], \overline{X})$ is the union of $Occ^{\star}_P(P[1..q], \overline{Y}) \oplus |\overline{Z}|$ and $Occ^{\star}_P(P[1..r], \overline{Z}) - Occ^{\star}_P(\overline{Y}, \overline{Z})$, where $r = J[q, Y]$. We note that $Occ^{\star}_P(P[1..q], \overline{X}) \oplus |\overline{Z}|$ is an arithmetic progression and the union is obtained as an extension of the progression.

When $X = Y^k$, we have two cases to consider.

Case 1: $|P| \leq |\overline{Y}|$. We have $Occ^{\star}_P(P[1..q], \overline{X}) = Occ^{\star}_P(P[1..q], \overline{Y})$.

Case 2: $|\overline{Y}| < |P|$. In general $Occ^{\star}_P(P[1..q], \overline{X})$ is the union of $\{|\overline{X}| + q - |P| \mid P \text{ is a prefix of } P[1..q]\overline{X}\}$ and $Occ^{\star}_P(P[1..f(q)], \overline{X})$, where f is the failure function of the KMP automaton, and hence we have only to determine the integers q such that $P[q+1..|P|]$ is a prefix of $\overline{X} = \overline{Y}^k$ in $O(|P|)$ time using $O(|P|)$ space. We note that $P[q+1..|P|]$ is a prefix of $\overline{X}$ if and only if: (1) $|\overline{Y}|$ is a period of $P[q+1..|P|]$; and (2) $P[q+1..q+|\overline{Y}|] = \overline{Y}$. Let w be the longest suffix of P such that $|\overline{Y}|$ is a period of w. The length of w can be computed in $O(|P|)$ time and space. The condition (1) holds if and only if $|w| \geq |P| - q$. The condition (2) holds if and only if $J[q, Y] = q + |\overline{Y}|$. Thus, we can enumerate the integers q satisfying the conditions in $O(|P|)$ time and space.

Lemma 5. *A two-dimensional table which stores the sets $Occ^{\star}_P(P[1..q], \overline{X})$ can be built in $O(|P| \cdot |\mathcal{D}|)$ time and space, if $\mathcal{D}$ is truncation-free.*

Now, we shift our attention to $Occ_P(\overline{X})$. For regular collage systems we have:

$$Occ_P(\overline{X}) = \begin{cases} \{0 \mid P = a\}, & \text{if } X = a; \\ Occ_P(\overline{Y}) \oplus |\overline{Z}| \cup Occ_P(\overline{Z}) \cup Occ^{\star}_P(\overline{Y}, \overline{Z}), & \text{if } X = YZ. \end{cases}$$

For any variable X, we can enumerate $Occ_P(\overline{X})$ in time linear in its size, by traversing the dependence tree rooted X with short-cut pointers as shown in [3].

Next we consider the case where repetition assignments $X = Y^k$ $(k \geq 3)$ occur in $\mathcal{D}$. There are two cases to consider.

Case 1: $|P| \leq |\overline{Y}|$. We have $Occ_P(\overline{X}) = Occ_P(\overline{YY}) \oplus \{i \cdot |\overline{Y}| \mid i = 0, \ldots, k-2\}$. We have only to store $Occ^{\star}_P(\overline{Y}, \overline{Y})$.

Case 2: $|\overline{Y}| < |P|$. Let $y = \overline{Y}$ and let $w = y^{\ell}$ with sufficiently large integer ℓ. We note that $Occ_P(w) \neq \emptyset$ if and only if (i) $\overline{Y}$ is a factor of P and (ii) $|\overline{Y}|$ is

a period of P. We can determine whether (i) holds in $O(|P|)$ time by finding q such that $J[q, Y] = q + |\overline{Y}|$. We can also determine whether (ii) holds in $O(|P|)$ time by checking $P[i] = P[i + |\overline{Y}|]$ for all $i = 1, \ldots, |P| - |\overline{Y}|$. Suppose that both (i) and (ii) hold. Let p ($p < |y|$) be the smallest integer in $Occ_P(w)$. We have $p + |y| \in Occ_P(w)$ since $|y|$ is a period of P. Let d be the smallest period of P. We have two subcases to consider.

1. $|y|$ is not a multiple of d. We assume for a contradiction that there exists p' with $p < p' < p + |y|$ such that $p' \in Occ_P(w)$. Then, $p' - p$ and $p + |y| - p'$ are periods of P, and $(p' - p) + (p + |y| - p') = |y| < |P|$. Since d is the smallest period of P, we have $(p' - p) + d < |P|$ and $d + (p + |y| - p') < |P|$. By Periodicity Lemma, the periods $(p' - p)$ and $(p + |y| - p')$ must be multiples of d, and therefore their sum $|y|$ must be a multiple of d, which is a contradiction. We now have proved that $Occ_P(w) = \{p + i \cdot |y| \mid i = 0, 1, \ldots\} \cap \{0, 1, \ldots, |w| - |P|\}$.
2. $|y|$ is a multiple of d. We can prove that there is no p' with $p < p' < p + d$ such that $p' \in Occ_P(w)$. Hence we have $Occ_P(w) = \{p + i \cdot d \mid i = 0, 1, \ldots\} \cap \{0, 1, \ldots, |w| - |P|\}$.

To summarize, we have

$$Occ_P(\overline{X}) = \{p + i \cdot h \mid i = 0, 1, \ldots\} \cap \{0, 1, \ldots, k \cdot |\overline{Y}| - |P|\},$$

where $h = d$, if $|\overline{Y}|$ is a multiple of d; and $h = |\overline{Y}|$, otherwise. Let q ($0 \leq q \leq |P|$) be the smallest integer such that $J[q, Y] = q + |\overline{Y}|$. Then, $p = |\overline{Y}| + h - |P| + q$.

Lemma 6. *A data structure can be built in $O(|\mathcal{D}| \cdot |P|)$ time and space which enumerates $Occ_P(\overline{X})$ in time linear in its size, if $\mathcal{D}$ is truncation-free.*

Lemma 7. *The function Output can be built in $O(|\mathcal{D}| \cdot |P|)$ time and space so that it responds in time linear in the answer size, if $\mathcal{D}$ is truncation-free.*

Theorem 2. *DCPMM can be built in $O(|\mathcal{D}| \cdot |P|)$ time and space, so that the values of the functions Δ and Λ, respectively, can be returned in constant time and in time linear in the answer size, if $\mathcal{D}$ is truncation-free.*

6 Experimental results

We implemented the proposed CPM algorithm in C language and evaluated their performance by a series of computational experiments. All the experiments were carried out on a PC with a 2.66 GHz Intel Core 2 Duo processor and 8.0 GB RAM running Linux (kernel 2.6.18). The text files we used are as follows.

Medline. A clinically-oriented subset of Medline, consisting of 348,566 references. The file size is 60.3 Mbytes.

Genbank. The file consisting only of accession numbers and nucleotide sequences taken from a data set in Genbank. The file size is 17.1 Mbytes.

To transform the texts into collage systems, we used the recursive-pairing [6] with upper-limit $255n + 1$ to $|\mathcal{D}|$ where n varied from 2 to 30. We next encoded the obtained collage systems with byte-oriented Huffman coding method. Then we ran the proposed CPM algorithm over these compressed files. The patterns searched for are substrings of the original texts.

Table 2 compares the compression ratios of our method and other compressors. Although the compression ratio of our method is poor compared to the standard compressors gzip and bzip2, it is higher than those of the Goal 2-oriented compressors.

Table 2. Compression ratios (in %) are compared, where gzip and bzip2 were executed with "-9" option (yielding "best" compression ratios).

	standard compressors			compressors for Goal 2				
						proposed method		
	compress	gzip	bzip2	BPE	[12]	$n = 10$	$n = 20$	$n = 30$
Medline	42.34	33.29	24.13	56.41	66.51	44.42	39.53	36.79
Genbank	26.80	21.98	22.71	31.37	51.54	29.21	29.41	29.74

Fig. 6 displays the compression ratios and the ratios $|\Phi(\mathcal{S})|/|T|$ (the compression ratios which ignores $\mathcal{D}$ and Φ) of our method with n varying from 2 to 30. The ratio $|\Phi(\mathcal{S})|/|T|$ monotonically decreases as n grows, whereas the change of compression ratio is not necessarily monotonic because the encoding size of $\mathcal{D}$ and Φ increases due to as n grows. The figure also shows the search time reduction ratios of our method in comparison with the KMP algorithm running over the original texts (the preprocessing times are negligible compared with the search times and excluded). Basically the search time reduction ratios are linear in $|\Phi(\mathcal{S})|/|T|$, although the DCPMM size grows linearly proportional to n, which possibly increases the L2-cache miss rate.

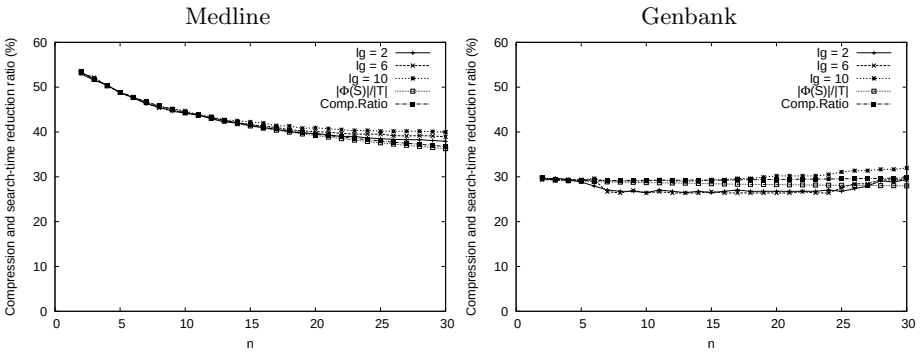


Fig. 6. Compression ratios and search-time reduction ratios are displayed of proposed method with n varying from 2 to 30, where pattern lengths are 2, 6, and 10.

7 Conclusion

The CPM algorithm presented in this paper takes as input truncation-free collage systems encoded using byte-oriented Huffman coding. Recursive-pairing used in our experiments produces regular collage systems, rather than truncation-free. To develop a new compression algorithm producing truncation-free collage systems of smaller size is one interesting future work.

Experimental results (omitted for lack of space) show that the method of Rautio et al. [12] is faster than ours in long pattern case, based on the Horspool variant of the BM algorithm, although the compression ratio is much worse than ours. To develop a BM-type algorithm for our scheme will be our future work.

References

1. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
2. P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.
3. T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage systems: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.
4. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. *J. of Discrete Algorithms*, 1(1):133–158, 2000.
5. J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. on Information Theory*, 46(3):737–754, 2000.
6. N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference '99 (DCC'99)*, pages 296–305. IEEE Press, 1999.
7. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. on Information Systems*, 15(2):124–136, 1997.
8. M. Miyazaki, S. Fukamachi, M. Takeda, and T. Shinohara. Speeding up the pattern matching machine for compressed texts. *Trans. of Information Processing Society of Japan*, 39(9):2638–2648, 1998.
9. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. on Information Systems*, 18(2):113–139, 2000.
10. G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, pages 14–36. Springer-Verlag, 1999.
11. C. G. Nevill-Manning, I. H. Witten, and D. L. Maulsby. Compression by induction of hierarchical grammars. In *Proc. Data Compression Conference '94 (DCC'94)*, pages 244–253. IEEE Press, 1994.
12. J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching*, LNCS 2373, pages 42–52. Springer-Verlag, 2002.
13. Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. 4th Italian Conference on Algorithms and Complexity (CIAC'00)*, LNCS 1767, pages 306–315. Springer-Verlag, 2000.
14. Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, LNCS 1848, pages 181–194. Springer-Verlag, 2000.