A Boyer–Moore Type Algorithm for Compressed Pattern Matching

Yusuke Shibata, Tetsuya Matsumoto, Masayuki Takeda,
Ayumi Shinohara, and Setsuo Arikawa

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

Abstract. We apply the Boyer–Moore technique to compressed pattern matching for text string described in terms of collage system, which is a formal framework that captures various dictionary-based compression methods. For a subclass of collage systems that contain no truncation, our new algorithm runs in $O(\|\mathcal{D}\| + n \cdot m + m^2 + r)$ time using $O(\|\mathcal{D}\| + m^2)$ space, where $\|\mathcal{D}\|$ is the size of dictionary $\mathcal{D}$, n is the compressed text length, m is the pattern length, and r is the number of pattern occurrences. For a general collage system, the time complexity is $O(\mathit{height}(\mathcal{D}) \cdot (\|\mathcal{D}\|+n)+n \cdot m+m^2+r)$, where $\mathit{height}(\mathcal{D})$ is the maximum dependency of tokens in $\mathcal{D}$. We showed that the algorithm specialized for the so-called byte pair encoding (BPE) is very fast in practice. In fact it runs about $1.2 \sim 3.0$ times faster than the exact match routine of the software package `agrep`, known as the fastest pattern matching tool.

1 Introduction

The problem of compressed pattern matching is to find pattern occurrences in compressed text without decompression. The goal is to search in compressed files faster than a regular decompression followed by an ordinary search (**Goal 1**). This problem has been extensively studied for various compression methods by several researchers in the last decade. For recent developments, see the survey [18].

This paper, however, focuses on another aspect of compressed pattern matching. We intend to reduce the time taken to search through a text file by reducing the size of it in a special way. That is, we regard text compression as a means of speeding up pattern matching rather than of saving storage or communication costs. The research goal to this direction is to search in compressed files faster than an ordinary search in the original files (**Goal 2**). If the goal is achieved, files that are usually not compressed because they are often read, can now be compressed for a speed-up. Let t_d, t_s, and t_c be the time for a decompression, the time for searching in uncompressed files, and the time for searching in compressed files, respectively. Goal 1 aims for $t_d + t_s > t_c$ while Goal 2 for $t_s > t_c$. Thus, Goal 2 is more difficult to achieve than Goal 1.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 181–194, 2000.

Let n and N denote the compressed text length and the original text length, respectively. Theoretically, the best compression has $n = \sqrt{N}$ for the Lempel-Ziv-Welch (LZW) encoding [22], and $n = \log N$ for LZ77 [25]. Thus an $O(n)$ time algorithm for searching directly in compressed text is considered to be better than an $O(N)$ time algorithm for searching in the original text. However, in practice n is linearly proportional to N for real text files. For this reason, an elaborate $O(n)$ time algorithm for searching in compressed text is often slower than a simple $O(N)$ time algorithm running on the original text, namely, does not achieve Goal 2. For example, it is reported in [12,11,16] that the proposed algorithms of compressed pattern matching for LZW achieved Goal 1, but did not achieve Goal 2. In order to achieve Goal 2, we shall re-estimate existing compression methods, choose a suitable one, and then develop an efficient compressed pattern matching algorithm for it. It should be emphasized that we are not particular about the traditional criteria, i.e., the compression ratio and the compression/decompression time.

As an effective tool for such a re-estimation, we introduced in [10] a *collage system*, that is a formal system to describe a string by a pair of dictionary $\mathcal{D}$ and sequence $\mathcal{S}$ of tokens defined in $\mathcal{D}$. The basic operations are concatenation, truncation, and repetition. Collage systems give us a unifying framework of various dictionary-based compression methods. We developed in [10] a general compressed pattern matching algorithm for the framework, which basically simulates the move of the Knuth-Morris-Pratt (KMP) automaton [13] on original texts. For a collage system which contains no truncation, the algorithm runs in $O(n + r)$ time after an $O(\|\mathcal{D}\| + m^2)$ time and space preprocessing, where $\|\mathcal{D}\|$ denotes the size of dictionary $\mathcal{D}$, m is the pattern length, and r is the number of pattern occurrences. For the case of LZW, it matches the same bound given in [12]. For a general collage system, which contains truncation, it runs in $O(\mathit{height}(\mathcal{D}) \cdot n + r)$ time after an $O(\mathit{height}(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time preprocessing using $O(\|\mathcal{D}\| + m^2)$ space, where $\mathit{height}(\mathcal{D})$ denotes the maximum dependency of the operations in $\mathcal{D}$. These results show that the truncation slows down the compressed pattern matching to the factor $\mathit{height}(\mathcal{D})$. It coincides with the observation by Navarro and Raffinot [16] that LZ77 is not suitable for compressed pattern matching compared with LZ78/LZW compression.

In a recent work [19], we focused on the compression method called the byte pair encoding (BPE) [9]. It describes a text as a collage system with concatenation only, in which the size of $\mathcal{D}$ is restricted to at most 256 so as to encode each token of $\mathcal{S}$ into one byte. Decompression is fast and requires small work space. Moreover, partial decompression is possible. This is a big advantage of BPE compared with the Lempel-Ziv family. Despite such advantages, BPE was seldom used until now. The reason is mainly for the following two disadvantages: the compression is terribly slow and the compression ratio is not as good as those of Compress and Gzip. However, we have shown that BPE is suitable for speeding up pattern matching. The algorithm proposed in [19] runs in $O(n+r)$ time after an $O(\|\mathcal{D}\| \cdot m)$ time and space preprocessing, and it is indeed faster than such $O(N)$ time algorithms as the KMP algorithm and the Shift-Or algorithm [24,4].

Moreover, it can be extended to deal with multiple patterns. The searching time is reduced at almost the same rate as the compression ratio.

However, there are sublinear time algorithms for the usual (not compressed) pattern matching problem, such as the Boyer–Moore (BM) algorithm [5], which skip many characters of text and run faster than the $O(N)$ time algorithms on the average, although the worst-case running time is $O(mN)$. Our algorithm presented in [19] defeats the exact match routine of `agrep` [23] for highly compressible texts such as genomic data. But it is not better than `agrep` when searching for a long pattern in texts that are not highly compressible by BPE. Then a question arises: *Does text compression speed up such a sublinear time algorithm?*

In this paper, we give an affirmative answer to this question. We present a general, BM type algorithm for texts described in terms of collage system. The algorithm runs on the sequence $\mathcal{S}$, with skipping some tokens. To our best knowledge, this is the first attempt to develop such an algorithm in compressed text.[1] The token-wise processing has two advantages compared with the usual character-wise processing. One is quick detection of a mismatch at each stage of the algorithm, and the other is larger shift depending upon one token (not upon one character) to align the phrase with its occurrence within the pattern. For a general collage system, the algorithm runs in $O((\mathit{height}(\mathcal{D}) + m) \cdot n + r)$ time, after an $O(\mathit{height}(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time preprocessing with $O(\|\mathcal{D}\| + m^2)$ space. In the case of no truncation, it runs in $O(mn + r)$ time, after an $O(\|\mathcal{D}\| + m^2)$ time and space preprocessing. However, we cannot shift the pattern without knowing the total length of phrases corresponding to skipped tokens. This slows down the algorithm in practice. To do without such information, we assume that the skipped phrases are all of length C, the maximum phrase length in $\mathcal{D}$, and divide the shift value by C. The value of C is crucial in this approach. For the BPE compression, we observed that putting a restriction on C makes no great sacrifice of compression ratio even for $C = 3, 4$. Experimental results show that the proposed algorithm searching in BPE compressed files is about $1.2 \sim 3.0$ times faster than `agrep` on the original files.

There are a few researches that aims Goal 2. The first attempt was made by Manber [14]. The compression scheme used is similar to but simpler than BPE, in which the maximum phrase length C is restricted to 2. The approach is to encode a given pattern and to apply any search routine in order to find the encoded pattern within compressed files. The problem in this approach is that the pattern may have more than one encoding. The solution given in [14] is to devise a way to restrict the number of possible encodings for any string with sacrifices in compression ratio. Thus the reductions in file size and searching time are only about 30%. Miyazaki et al. [15] presented an efficient realization of pattern matching machine for searching directly in a Huffman encoded text. The reduction in searching time is almost the same as that in file size. Moura et al. [8] proposed a compression scheme that uses a word-based Huffman encoding with

[1] However, Navarro et al. [17] in this conference gives a similar algorithm, which is restricted to the LZ78/LZW format.

a byte-oriented code, which allows a search twice faster than `agrep`. However, the compression method is not applicable to such texts as genomic sequence data since they cannot be segmented into words. Our previous and new algorithms can deal with such texts.

2 Preliminaries

Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string $u = xyz$, respectively. A prefix, factor, and suffix of a string u is said to be *proper* if it is not u. The length of a string u is denoted by $|u|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. The ith symbol of a string u is denoted by $u[i]$ for $1 \leq i \leq |u|$, and the factor of a string u that begins at position i and ends at position j is denoted by $u[i:j]$ for $1 \leq i \leq j \leq |u|$. For convenience, let $u[i:j] = \varepsilon$ for $j < i$. For a string u and a non-negative integer i, the string obtained by removing the length i prefix (resp. suffix) from u is denoted by ${}^{[i]}u$ (resp. $u^{[i]}$). That is, ${}^{[i]}u = u[i+1:|u|]$ and $u^{[i]} = u[1:|u|-i]$.

3 A Unifying Framework for Compressed Pattern Matching

In a dictionary-based compression, a text string is described by a pair of a *dictionary* and a sequence of *tokens*, each of which represents a phrase defined in the dictionary. Kida et al. [10] introduced a unifying framework, named *collage system*, which abstracts various dictionary-based methods, such as the Lempel-Ziv family and the static dictionary methods. In [10] they presented a general compressed pattern matching algorithm for the framework. Consequently, any compression method within the framework has a compressed pattern matching algorithm as an instance.

3.1 Collage System

A *collage system* is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ defined as follows: $\mathcal{D}$ is a sequence of assignments $X_1 = expr_1;\ X_2 = expr_2;\ \cdots;\ X_\ell = expr_\ell$, where each X_k is a token (or a variable) and $expr_k$ is any of the form

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

Each 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 $\mathcal{D}$ is the number n of assignments and

denoted by $\|\mathcal{D}\|$. Define the *height* of a token X to be the height of the syntax tree whose root is X. The *height* of $\mathcal{D}$ is defined by $height(\mathcal{D}) = \max\{height(X) \mid X \text{ in } \mathcal{D}\}$. It expresses the maximum dependency of the tokens in $\mathcal{D}$. On the other hand, $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$ is a sequence of tokens defined in $\mathcal{D}$. The collage system represents a string obtained by concatenating the phrases represented by $X_{i_1}, X_{i_2}, \ldots, X_{i_n}$.

3.2 Pattern Matching in Collage Systems

Our problem is defined as follows.

> Given a pattern $\pi = \pi[1 : m]$ and a collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ with $\mathcal{S} = \mathcal{S}[1 : n]$, find all locations at which π occurs within the original text $\mathcal{S}[1].u \cdot \mathcal{S}[2].u \cdots \mathcal{S}[n].u$.

Kida et al. [10] presented an algorithm solving the above problem. Figure 1 gives an overview of the algorithm, which processes $\mathcal{S}$ token-by-token. The algorithm

```
Input:   Pattern π and collage system consisting of D and S = S[1 : n].
Output:  All occurrences of π in the original text.
begin
/* Preprocessing for computing Jump_KMP and Output_KMP. */
    Preprocess the pattern π and the dictionary D;
/* Main routine */
    state := 0;   ℓ := 0;
    for i := 1 to n do begin
        for each d ∈ Output_KMP(state, S[i]) do
            Report a pattern occurrence that ends at position ℓ + d;
        state := Jump_KMP(state, S[i]);   ℓ := ℓ + |S[i].u|
    end
end.
```

Fig. 1. General algorithm for searching in a collage system.

simulates the move of the KMP automaton running on the original text, by using two functions $Jump_{\mathrm{KMP}}$ and $Output_{\mathrm{KMP}}$, both take as input a state and a token. The former is used to substitute just one state transition for the consecutive state transitions of the KMP automaton caused by each of the phrases, and the latter is used to report all pattern occurrences found during the state transitions. Thus the definitions of the two functions are as follows.

$$Jump_{\mathrm{KMP}}(j, t) = \delta(j, t.u),$$

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

where δ is the state transition function of the KMP automaton.

Theorem 1 (Kida et al. [10]). *The algorithm of Fig. 1 runs in $O(height(\mathcal{D}) \cdot n+r)$ time after an $O(height(\mathcal{D}) \cdot \|\mathcal{D}\|+m^2)$ time preprocessing using $O(\|\mathcal{D}\|+m^2)$ space, where r is the number of pattern occurrences. The factor $height(\mathcal{D})$ can be dropped if $\mathcal{D}$ contains no truncation.*

This idea is a generalization of the algorithm due to Amir et al. [3], which is restricted to LZW compressed texts. Shibata et al. [20] applied a similar technique to the case of the compression using anti-dictionaries [6]. An extension of [3] to multiple pattern searching was presented by Kida et al. [12], which is based on the Aho-Corasick (AC) pattern matching algorithm [1]. The technique of [12] was then generalized to multiple pattern searching in collage systems which contain concatenation only [10]. Bit-parallel realization of [3] was independently proposed in [11,16] and proved to be fast in practice for a short pattern ($m \leq 32$).

3.3 Practical Aspects

Theorem 1 suggests that a compression method which describes a text as a collage system with no truncation might be suitable for the compressed pattern matching. For example, the collage systems for LZW contain no truncation but those for LZ77 have truncation. It implies that LZW is suitable compared with LZ77. This coincides with the observation by Navarro and Raffinot [16] that the compressed pattern matching for LZ77 achieves none of Goal 1 and Goal 2, but that for LZW achieves Goal 1. However, it was observed that the compressed pattern matching for LZW is too slow to achieve Goal 2. We have two reasons. One is that in LZW the dictionary $\mathcal{D}$ is not encoded explicitly: it will be incrementally re-built from $\mathcal{S}$. The preprocessing of $\mathcal{D}$ is therefore merged into the main routine (see Fig. 1 again). The other reason is as follows. Although $Jump_{\mathrm{KMP}}$ can be realized using only $O(\|\mathcal{D}\| + m^2)$ space so that it answers in constant time, the constant factor is relatively large. The two-dimensional array realization of $Jump_{\mathrm{KMP}}$ would improve this, but it requires $O(\|\mathcal{D}\| \cdot m)$ space, which is unrealistic because $\|\mathcal{D}\|$ is linear in n in the case of LZW.

From the above observations the desirable properties for compressed pattern matching can be summarized as follows.

- The dictionary $\mathcal{D}$ contains no truncation.
- The dictionary $\mathcal{D}$ is encoded separately from the sequence $\mathcal{S}$.
- The size of $\mathcal{D}$ is small enough.
- The tokens of $\mathcal{S}$ are encoded using a fixed length code.

The BPE compression [9] is the one which satisfies all of the properties. The collage systems for BPE have concatenation only, and $\|\mathcal{D}\|$ is restricted to at most 256 so as to encode each token of $\mathcal{S}$ into one byte. By using the two-dimensional array implementation, we presented in [19] an algorithm for searching in BPE compressed files, which runs in $O(n+r)$ time after an $O(\|D\| \cdot m)$ time and space preprocessing. This algorithm defeats the BM algorithm for highly compressible text files such as biological data. However, it is not better

```
T[0] := $; /* $ is a character that never occurs in pattern */
i := m;
while i ≤ N do begin
    state := 0;    ℓ := 0;
    while g(state, T[i − ℓ]) is defined do begin
        state := g(state, T[i − ℓ]);        ℓ := ℓ + 1
    end;
    if state = m then report a pattern occurrence;
    i := i + σ(state, T[i − ℓ])
end
```

Fig. 2. BM algorithm on uncompressed text.

than the BM algorithm in the case of searching for a long pattern in text files that are not highly compressible by BPE. For this reason, we try to devise a BM type algorithm for searching in BPE compressed files.

4 BM Type Algorithm for Compressed Pattern Matching

We first briefly sketches the BM algorithm, and show a general, BM type algorithm for searching in collage systems. Then, we discuss searching in BPE compressed files from the practical viewpoints.

4.1 BM Algorithm on Uncompressed Text

The BM algorithm performs the character comparisons in the right-to-left direction, and slides the pattern to the right using the so-called shift function when a mismatch occurs. The algorithm for searching in text $T[1 : N]$ is shown in Fig. 2. Note that the function g is the state transition function of the (partial) automaton that accepts the reversed pattern, in which state j represents the length j suffix of the pattern ($0 \leq j \leq m$).

Although there are many variations of the shift function, they are basically designed to shift the pattern to the right so as to align a text substring with its rightmost occurrence within the pattern. Let

$$rightmost\_occ(w) = \min \left\{ \ell > 0 \;\middle|\; \begin{array}{l} \pi[m-\ell-|w|+1 : m-\ell] = w, \text{ or} \\ \pi[1 : m-\ell] \text{ is a suffix of } w \end{array} \right\}.$$

The following definition, given by Uratani and Takeda [21] (for multiple pattern case), is the one which utilizes all information gathered in one execution of the inner-while-loop in the algorithm of Fig. 2.

$$\sigma(j, a) = rightmost\_occ(a \cdot \pi[m-j+1 : m]).$$

The two-dimensional array realization of this function requires $O(|\Sigma| \cdot m)$ memory, but it becomes realistic due to recent progress in computer technology. Moreover, the array can be shared with the goto function g. This saves not only memory requirement but also the number of table references.

4.2 BM Type Algorithm for Collage System

Now, we show a BM type algorithm for searching in collage systems. Figure 3 gives an overview of our algorithm. For each iteration of the while-loop, we report

```
/*Preprocessing for computing Jump_BM(j,t), Output_BM(j,t), and Occ(t) */
    Preprocess the pattern π and the dictionary D;
/* Main routine */
    focus := an appropriate value;
    while focus ≤ n do begin
Step 1: Report all pattern occurrences that are contained in the phrase S[focus].u
          by using Occ(t);
Step 2: Find all pattern occurrences that end within the phrase S[focus].u
          by using Jump_BM(j,t) and Output_BM(j,t);
Step 3: Compute a possible shift Δ based on information gathered in Step 2;
        focus := focus + Δ
    end
```

Fig. 3. Overview of BM type compressed pattern matching algorithm.

in Step 1 all the pattern occurrences that are contained in the phrase represented by the token we focus on, determine in Step 2 the pattern occurrences that end within the phrase, and then shift our focus to the right by Δ obtained in Step 3. Let us call the token we focus on the *focused token*, and the phrase it represents the *focused phrase*. For step 1, we shall compute during the preprocessing, for every token t, the set $Occ(t)$ of all pattern occurrences contained in the phrase $t.u$. The time and space complexities of this computation are as follows.

Lemma 1 (Kida et al. 1999). *We can build in $O(height(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time using $O(\|\mathcal{D}\| + m^2)$ space a data structure by which the enumeration of the set $Occ(t)$ is performed in $O(height(t) + \ell)$ time, where $\ell = |Occ(t)|$. If $\mathcal{D}$ contains no truncation, it can be built in $O(\|\mathcal{D}\| + m^2)$ time and space, and the enumeration requires only $O(\ell)$ time.*

In the following we discuss how to realize Step 2 and Step 3.

Figure 4 illustrates pattern occurrences that end within the focused phrase. A candidate for pattern occurrence is a non-empty prefix of the focused phrase that is also a proper suffix of the pattern. There may be more than one candidate to be checked. One naive method is to check all of them independently, but we take here another approach. We shall start with the longest one. For the case of uncompressed text, we can do it by using the partial automaton for the reversed pattern stated in Section 4.1. When a mismatch occurs, we change the state by using the failure function and try to proceed into the left direction. The process is repeated until the pattern does not have an overlap with the focused phrase. In order to perform such processing over compressed text, we use the two functions $Jump_{\mathrm{BM}}$ and $Output_{\mathrm{BM}}$ defined in the sequel.

Let $lpps(w)$ denote the longest prefix of a string w that is also a proper suffix of the pattern π. Extend the function g into the domain $\{0,\ldots,m\} \times \Sigma^*$ by $g(j, aw) = g(g(j, w), a)$, if $g(j, w)$ is defined and otherwise, $g(j, aw)$ is undefined, where $w \in \Sigma^*$ and $a \in \Sigma$. Let $f(j)$ be the largest integer k $(k < j)$ such that the length k suffix of the pattern is a prefix of the length j suffix of the pattern. Note that f is the same as the failure function of the KMP automaton. Define the functions $Jump_{\mathrm{BM}}$ and $Output_{\mathrm{BM}}$ by

$$Jump_{\mathrm{BM}}(j,t) = \begin{cases} g(j, t.u), & \text{if } j \neq 0; \\ g(j, lpps(t.u)), & \text{if } j = 0 \text{ and } lpps(t.u) \neq \varepsilon; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$Output_{\mathrm{BM}}(j,t) = \begin{cases} true, & \text{if } g(j,w) = m \text{ and } w \text{ is a proper suffix of } t.u; \\ false, & \text{otherwise.} \end{cases}$$

The procedure for Step 2 is shown in Fig. 5.

We now discuss how to compute the possible shift Δ of the focus. Let

$$Shift(j,t) = rightmost\_occ(t.u \cdot \pi[m-j+1:m]).$$

Assume that starting at the token $\mathcal{S}[focus]$, we encounter a mismatch against a token t in state j. Find the minimum integer $k > 0$ such that

$$Shift(0, \mathcal{S}[focus]) \leq \sum_{i=1}^{k} \Big| \mathcal{S}[focus+i].u \Big|, \quad \text{or} \tag{1}$$

$$Shift(j,t) \leq \sum_{i=0}^{k} \Big| \mathcal{S}[focus+i].u \Big| - \Big| lpps(\mathcal{S}[focus].u) \Big|. \tag{2}$$

Note that the shift due to Eq. (1) is possible independently of the result of the procedure of Fig. 5. When returning at the first if-then statement of the procedure in Fig. 5, we can shift the focus by the amount due to Eq. (1). Otherwise, we shift the focus by the amount due to both Eq. (1) and Eq. (2) for $j = state$ and $t = \mathcal{S}[focus - \ell]$ just after the execution of the while-loop at the first iteration of the repeat-until loop.

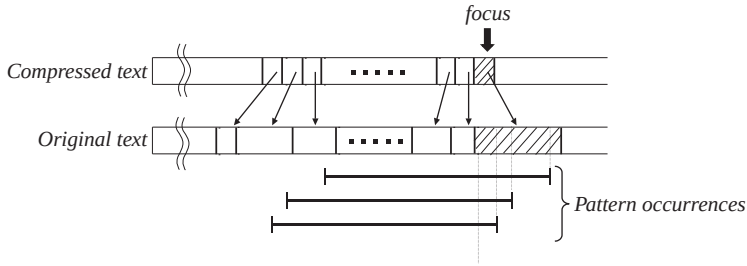

Fig. 4. Pattern occurrences.

.

```
procedure Find_pattern_occurrences(focus : integer);
begin
    if Jump_BM(0, S[focus]) is undefined then return;
    state := Jump_BM(0, S[focus]);    d := state;    ℓ := 1;
    repeat
        while Jump_BM(state, S[focus − ℓ]) is defined do begin
            state := Jump_BM(state, S[focus − ℓ]);    ℓ := ℓ + 1
        end;
        if Output_BM(state, S[focus − ℓ]) = true then report a pattern occurrence;
        d := d − (state − f(state));    state := f(state)
    until d ≤ 0
end;
```

Fig. 5. Finding pattern occurrences in Step 2.

Lemma 2. *The functions $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$, and Shift can be built in $O(height(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time and $O(\|\mathcal{D}\| + m^2)$ space, so that they answer in $O(1)$ time. The factor $height(\mathcal{D})$ can be dropped if $\mathcal{D}$ contains no truncation.*

Proof. We can prove the lemma by using techniques similar to those in [10], but the proof is omitted for lack of space. □

Theorem 2. *The algorithm of Fig. 3 runs in $O(height(\mathcal{D}) \cdot (\|\mathcal{D}\| + n) + n \cdot m + m^2 + r)$ time, using $O(\|\mathcal{D}\| + m^2)$ space. If $\mathcal{D}$ contains no truncation, the time complexity becomes $O(\|\mathcal{D}\| + n \cdot m + m^2 + r)$.*

4.3 Searching in BPE Compressed Files

We again take the two-dimensional array implementation for the functions $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$, and *Shift*. Since the collage systems for BPE contain no truncation, the construction of the functions seems to require $O(\|\mathcal{D}\| + m^2)$ time and space in addition to $O(\|\mathcal{D}\| \cdot m)$ time and space. However, we can build them in another way, and have the following result.

Theorem 3. *The tables storing $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$, and Shift can be built in $O(\|\mathcal{D}\| \cdot m)$ time and space, if $\mathcal{D}$ contains concatenation only.*

Proof. We can fill the entries of the tables in a bottom-up manner by using the directed acyclic word graph [7] for the reversed pattern. □

The computation of Δ stated in Section 4.2 requires knowing the lengths of the phrases represented by the skipped tokens. This slows down the searching speed. To do without such information, we assume they are all of length C, where C is the maximum phrase length, and let

$$\Delta(j, t) = \max\Big(\lceil \mathit{Shift}(0, t)/C \rceil, \lfloor \mathit{Shift}(j, t)/C \rfloor\Big).$$

The value of C is a crucial factor in such an approach, We estimated the change of compression ratios depending on C. The text files we used are:

Medline. A clinically-oriented subset of Medline, consisting of 348,566 references. The file size is 60.3 Mbyte and the entropy is 4.9647.

Genbank. The file consisting only of accession numbers and nucleotide sequences taken from a data set in Genbank. The file size is 17.1 Mbyte and the entropy is 2.6018.

Table 1 shows the compression ratios of these texts for BPE, together with those for the Huffman encoding, `gzip`, and `compress`, where the last two are well-known compression tools based on LZ77 and LZW, respectively. Remark that the change of compression ratios depending on C is non-monotonic. The reason for this is that the BPE compression routine we used builds a dictionary $\mathcal{D}$ in a greedy manner only from the first block of a text file. It is observed that we can restrict C with no great sacrifice of compression ratio. Thus we decided to use the BPE compressed file of Medline for $C = 3$, and that of Genbank for $C = 4$ in our experiment in the next section.

Table 1. Compression ratios (%).

	Huffman	BPE							`compress`	`gzip`
		$C = 3$	$C = 4$	$C = 5$	$C = 6$	$C = 7$	$C = 8$	unlimit.		
Medline	62.41	59.44	58.46	58.44	58.53	58.47	58.58	59.07	42.34	33.35
Genbank	33.37	36.93	32.84	32.63	32.63	32.34	32.28	32.50	26.80	23.15

5 Experimental Results

We estimated the performances of the following programs:

(A) *Decompression followed by ordinary search.*
We tested this approach with the KMP algorithm for the compression methods: `gzip`, `compress`, and BPE. We did not combine the decompression programs and the KMP search program using the Unix 'pipe' because it is slow. Instead, we embedded the KMP routine in the decompression programs, so that the KMP automaton processes the decoded characters 'on the fly'. The programs are abbreviated as `gunzip`+KMP, `uncompress`+KMP, and `unBPE`+KMP, respectively.

(B) *Ordinary search in original text.*
KMP, UT (the Uratani-Takeda variant [21] of BM), and `agrep`.

(C) *Compressed pattern matching.*
AC on LZW [12], Shift-Or on LZW [11], AC on Huffman [15], AC on BPE [19], and BM on BPE (the algorithm proposed in this paper).

The automata in KMP, UT, AC on Huffman, AC on BPE, and BM on BPE were realized as two-dimensional arrays of size $\ell \times 256$, where ℓ is the number of states. The texts used are Medline and Genbank mentioned in Section 4.3, and

the patterns searched for are text substrings randomly gathered from them. Our experiment was carried out on an AlphaStation XP1000 with an Alpha21264 processor at 667MHz running Tru64 UNIX operating system V4.0F. Figure 6 shows the running times (CPU time). We excluded the preprocessing times since they are negligible compared with the running times. We observed the following facts.

- The differences between the running times of KMP and the three programs of (A) correspond to the decompression times. Decompression tasks for LZ77 and LZW are thus time-consuming compared with pattern matching task. Even when we use the BM algorithm instead of KMP, the approach (A) is still slow for LZ77 and LZW.
- The proposed algorithm (BM on BPE) is faster than all the others. Especially, it runs about 1.2 times faster than `agrep` for Medline, and about 3 times faster for Genbank.

6 Conclusion

We have presented a BM type algorithm for compressed pattern matching in collage system, and shown that an instance of the algorithm searches in BPE compressed texts $1.2 \sim 3.0$ faster than `agrep` does in the original texts. For searching a very long pattern (e.g., $m > 30$), a simplified version of the backward-dawg-matching algorithm [7] is very fast as reported in [2]. To develop its compressed matching version will be our future work.

References

1. A. V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975.
2. C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle, suffix oracle. Technical Report IGM-99-08, Institut Gaspard-Monge, 1999.
3. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1996.
4. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Comm. ACM*, 35(10):74–82, 1992.
5. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):62–72, 1977.
6. M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *Proc. 26th Internationial Colloquim on Automata, Languages and Programming*, pages 261–270. Springer-Verlag, 1999.
7. M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, New York, 1994.
8. E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. 5th International Symp. on String Processing and Information Retrieval*, pages 90–95. IEEE Computer Society, 1998.
9. P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

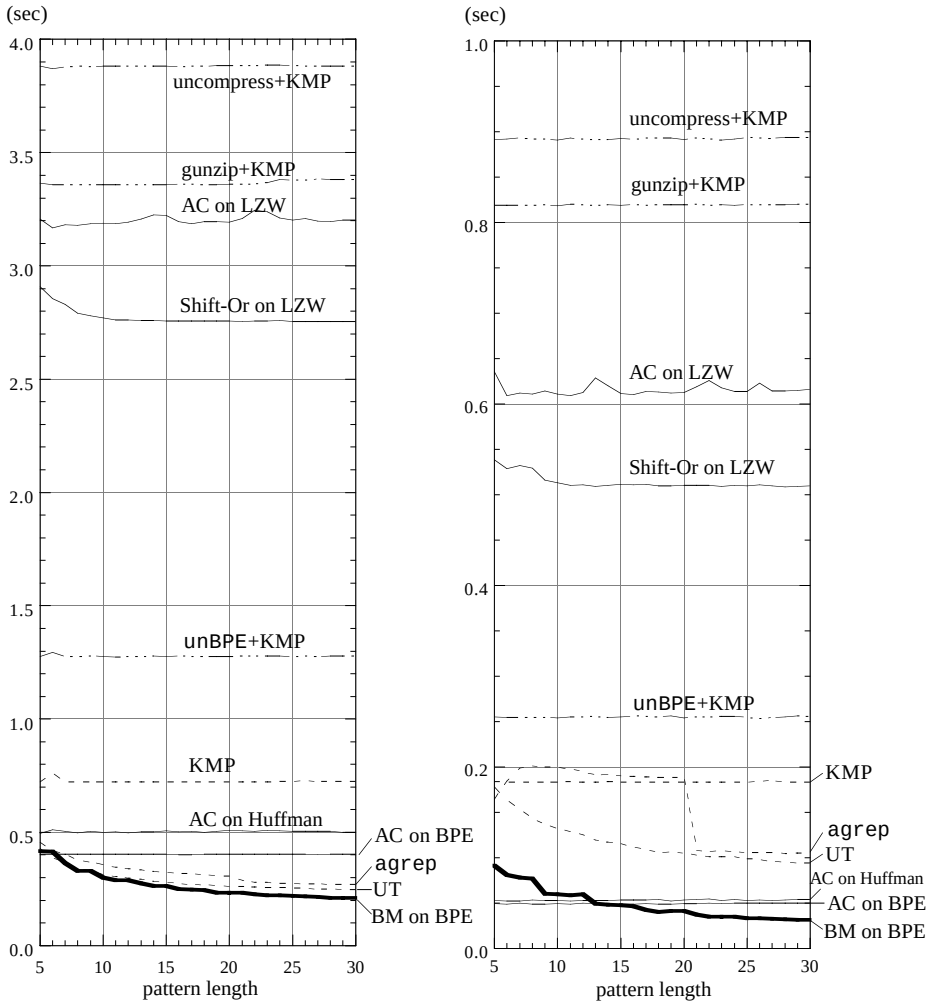

Fig. 6. Running times.

10. T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th International Symp. on String Processing and Information Retrieval*, pages 89–96. IEEE Computer Society, 1999.

11. T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 1–13. Springer-Verlag, 1999.

12. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. Data Compression Conference (DCC'98)*, pages 103–112. IEEE Computer Society, 1998.

13. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput*, 6(2):323–350, 1977.
14. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc. 5th Ann. Symp. on Combinatorial Pattern Matching*, pages 113–124. Springer-Verlag, 1994.
15. M. Miyazaki, S. Fukamachi, M. Takeda, and T. Shinohara. Speeding up the pattern matching machine for compressed texts. *Transactions of Information Processing Society of Japan*, 39(9):2638–2648, 1998. (in Japanese).
16. G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 14–36. Springer-Verlag, 1999.
17. G. Navarro and J. Tarhio. Boyer–Moore string matching over Ziv-Lempel compressed text. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*. Springer-Verlag, 2000. to appear.
18. W. Rytter. Algorithms on compressed strings and arrays. In *Proc. 26th Ann. Conf. on Current Trends in Theory and Practice of Infomatics*. Springer-Verlag, 1999.
19. 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*, pages 306–315. Springer-Verlag, 2000.
20. Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 37–49. Springer-Verlag, 1999.
21. N. Uratani and M. Takeda. A fast string-searching algorithm for multiple patterns. *Information Processing & Management*, 29(6):775–791, 1993.
22. T. A. Welch. A technique for high performance data compression. *IEEE Comput.*, 17:8–19, June 1984.
23. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, 1992.
24. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35(10):83–91, October 1992.
25. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inform. Theory*, IT-23(3):337–349, May 1977.