An Improved Pattern Matching Algorithm for Strings in terms of Straight-line Programs

MASAMICHI MIYAZAKI [1],AYUMI SHINOHARA, MASAYUKI TAKEDA, *Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan,* {masamich, ayumi, takeda}@i.kyushu-u.ac.jp

ABSTRACT: We show an efficient pattern-matching algorithm for strings that are succinctly described in terms of straight-line programs, in which the constants are symbols and the only operation is the concatenation. In this paper, both text T and pattern P are given by straight-line programs $\mathcal{T}$ and $\mathcal{P}$. The length of the text T (pattern P, resp.) may grow exponentially with respect to its description size $||\mathcal{T}|| = n$ ($||\mathcal{P}|| = m$, resp.). We show a new combinatorial property concerning with the periodic occurrences of a pattern in a text. Based on this property, we develop an $O(n^2m^2)$ time algorithm using $O(nm)$ space, which outputs a compact representation of all occurrences of P in T. This is superior to the algorithm proposed by Karpinski *et al.*[13], which runs in $O((n+m)^4 \log (n+m))$ time using $O((n+m)^3)$ space. Moreover, our algorithm is much simpler, and the experimental results show that our algorithm is more efficient than theirs from the practical view point.

Keywords: Compressed string matching, Periodicity, Combinatorial pattern matching, Straight-line program.

1 Introduction

The string pattern-matching is a task to find all occurrences of a pattern in a text. In practice the text is large and is often stored in compressed form in secondary storage. In order to find a pattern in the text quickly, we usually use some text indexing data structures [9, 15]. However, it requires extra-space, which is often undesirable. Moreover, decompressing the original text also consumes extra-space. Thus, it is very attractive to develop an efficient pattern-matching algorithm for searching a compressed text without any extra data structures nor decompression process. Moreover, if the text is stored in some compressed form, the data transmission time from the secondary storage to the main memory is decreased according to the compression ratio. Text compression thus possibly speeds up pattern-matching. It also justifies the importance of an efficient pattern-matching algorithm which searches a compressed text directly.

The problem of pattern-matching in compressed text is of not only practical interest but also of theoretical interest. Several researchers have studied it for various com-

[1]The author is currently working at NEC.

pression methods. For example, [1, 2, 3, 4, 7] are for the run-length coding, [5] for the LZW coding, [8, 10, 11] for the LZ77 coding. Refer to an excellent survey [17] for recent development on this topic.

A straight-line program is a compact representation of string. It is a context-free grammar in the Chomsky normal form that derives only one string. The length of the string represented by a straight-line program can be exponentially long with respect to the size of the straight-line program. In this sense, conversion of string into straight-line program can be viewed as a kind of text compressions.

In this paper we concentrate on the pattern-matching problem where both text and pattern are represented in terms of straight-line programs. Karpinski *et al.* [12] showed the first polynomial-time algorithm. Later in [13] they proposed an $O((n+m)^4 \log(n+m))$ time algorithm using $O((n+m)^3)$ space, where n and m are the sizes of straight-line programs representing the text and the pattern, respectively. However, the algorithm is rather complicated. In this paper we exploit a new combinatorial property concerning with the periodic occurrences of a pattern in a text, and then present an $O(n^2m^2)$ time algorithm using $O(nm)$ space, which is based on this property. Our algorithm is simpler, and outputs an $O(n)$ representation of all occurrences. Moreover, we implemented both our algorithm and the algorithm in [13]. Comparing these two algorithms from practical view point, we verified that our algorithm is more efficient than the algorithm in [13].

A preliminary version of this paper was presented at [16].

2 Preliminary

In this paper, both text and pattern are described in terms of straight-line programs. A *straight-line program* $\mathcal{R}$ is a sequence of assignments as follows:

$$X_1 = expr_1;\ X_2 = expr_2;\ \ldots;\ X_n = expr_n,$$

where X_i are variables and $expr_i$ are expressions of the form:

- $expr_i$ is a symbol of a given alphabet Σ, or
- $expr_i = X_\ell \cdot X_r\ (\ell, r < i)$, where $\cdot$ denotes the concatenation of X_ℓ and X_r.

Denote by R the string which is derived from the last variable X_n of the program $\mathcal{R}$. The size of the straight-line program $\mathcal{R}$, denoted by $\|\mathcal{R}\|$, is the number n of assignments in $\mathcal{R}$. The length of a string w is denoted by $|w|$. We identify a variable X_i with the string represented by X_i if it is clear from the context.

EXAMPLE 2.1
Let us consider the following straight-line program $\mathcal{R}$:

$$X_1 = \mathtt{a};\ X_2 = \mathtt{b};\ X_3 = X_1 \cdot X_2;\ X_4 = X_3 \cdot X_1;\ X_5 = X_3 \cdot X_4;$$
$$X_6 = X_5 \cdot X_5;\ X_7 = X_4 \cdot X_6;\ X_8 = X_7 \cdot X_5.$$

We can see that $R = X_8 = \mathtt{abaababaabababaababa}$, and $\|\mathcal{R}\| = 8$, $|R| = 18$. The evaluation tree is shown in Fig. 1.

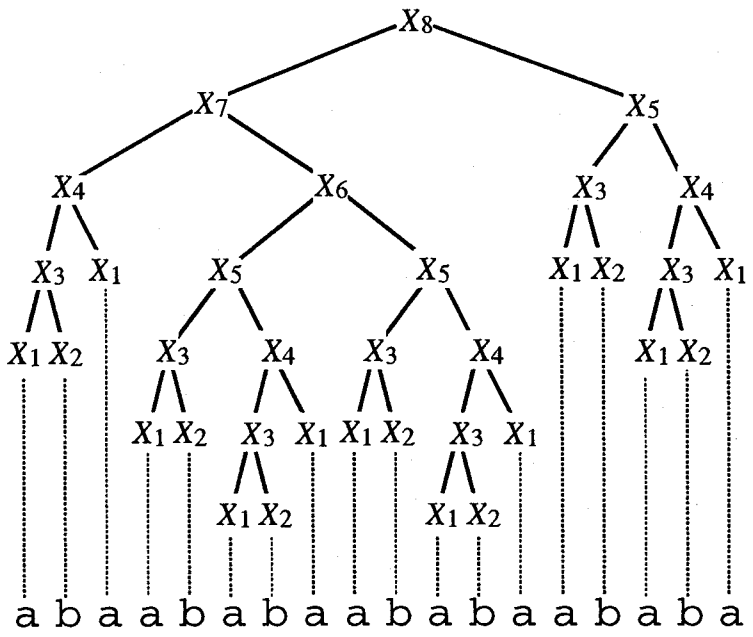

FIG. 1. Evaluation tree of $\mathcal{R}$ in Example 1.

We define the *depth* of a variable X in a straight-line program $\mathcal{R}$ by

$$depth(X) = \begin{cases} 1 & \text{if } X = a \in \Sigma, \\ 1 + \max(depth(X_\ell), depth(X_r)) & \text{if } X = X_\ell \cdot X_r. \end{cases}$$

It corresponds to the length of the longest path from X to a leaf in the tree.

For a string w denote by $w[i..j](1 \leq i \leq j \leq |w|)$ the subword of w starting at i and ending at j. A *period* of a string w is an integer p, $(0 < p \leq |w|)$, such that $w[i] = w[i+p]$ for any $i = 1, \ldots, |w| - p$.

The *pattern matching problem for strings in terms of straight-line programs* is, given straight-line programs $\mathcal{P}$ and $\mathcal{T}$ which are the descriptions of pattern P and text T respectively, to find all occurrences of P in T. Namely, we will compute the following set:

$$Occ(T, P) = \{i \mid T[i..i + |P| - 1] = P\}.$$

Hereafter, we use X and X_i for variables in $\mathcal{T}$ and Y and Y_j for variables in $\mathcal{P}$. We assume $\|\mathcal{T}\| = n$ and $\|\mathcal{P}\| = m$. For a set U of integers and an integer k, we denote $U \oplus k = \{i + k \mid i \in U\}$ and $U \ominus k = \{i - k \mid i \in U\}$. An *arithmetic progression* is a sequence in which each term after the first is determined by adding a constant, called a *step*, to the preceding term.

3 Overview of algorithm

In this section, we give an overview of our algorithm together with its basic idea. First we consider a compact representation of the set $Occ(X, Y)$.

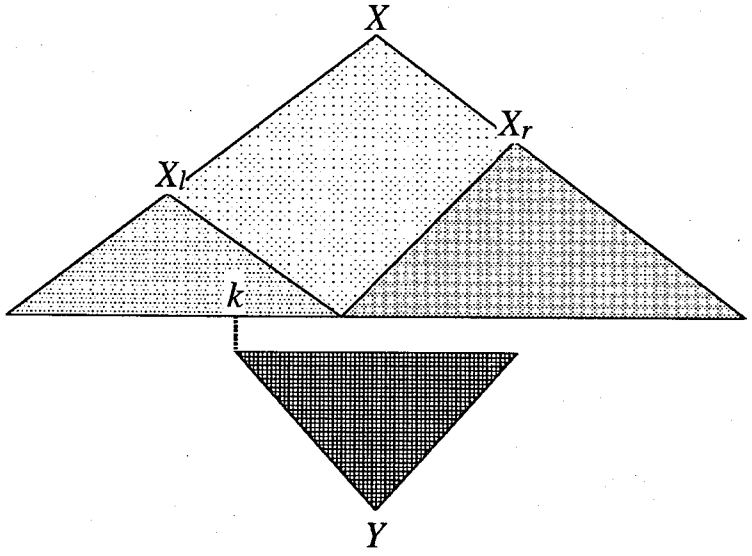

FIG. 2: $k \in Occ^{\star}(X,Y)$, since Y covers or touches the boundary between X_ℓ and X_r.

Suppose $X = X_\ell \cdot X_r$. We define $Occ^{\star}(X,Y)$ to be the set of occurrences of Y in X such that Y covers or touches the boundary between X_ℓ and X_r (see Fig. 2):

$$Occ^{\star}(X,Y) = \{s \in Occ(X,Y) : |X_\ell| - |Y| + 1 \leq s \leq |X_\ell| + 1\}.$$

For convenience, let $Occ^{\star}(X,Y) = Occ(X,Y)$ for $X = a \in \Sigma$. Then we have the following lemma, which is informally stated in [10]. For the sake of completeness, we give a proof here.

LEMMA 3.1
For any X in $\mathcal{T}$ and any Y in $\mathcal{P}$, $Occ^{\star}(X,Y)$ forms a single arithmetic progression.

PROOF. It is enough to consider the case $Occ^{\star}(X,Y)$ contains at least three elements, since any set of integers with cardinality at most two forms a single arithmetic progression. Let i, j, and k be consecutive elements arbitrarily chosen from $Occ^{\star}(X,Y)$ in increasing order. We will show that $j - i = k - j$, which implies that $Occ^{\star}(X,Y)$ forms a single arithmetic progression. Since Y occurs in X at positions i and k and both the occurrences cover the same boundary, we have $k - i \leq |Y|$. Let p_0 be the smallest period of Y, and let $p_1 = j - i$ and $p_2 = k - j$. Since both p_1 and p_2 are periods of Y, and p_0 is the smallest period of Y, we have $p_0 \leq p_1$ and $p_0 \leq p_2$. Thus $p_1 + p_0 \leq p_1 + p_2 = (j - i) + (k - j) = k - i \leq |Y|$. By the periodicity lemma (See [6], p. 24), the greatest common divisor d of p_1 and p_0 is also a period of Y. Since p_0 is the smallest period, we have $d = p_0$, which implies that $p_1 = l \cdot p_0$ for some $l \geq 1$. Suppose $l \geq 2$. Then $j = i + l \cdot p_0 > i + p_0 > i$. Since both i and j are in $Occ^{\star}(X,Y)$, and p_0 is a period of Y, we have $i + p_0 \in Occ^{\star}(X,Y)$. This contradicts the assumption that i and j are consecutive elements in $Occ^{\star}(X,Y)$. Therefore $l = 1$, that is $p_1 = p_0$. In the same way, we can see that $p_2 = p_0$. This completes the proof. ∎

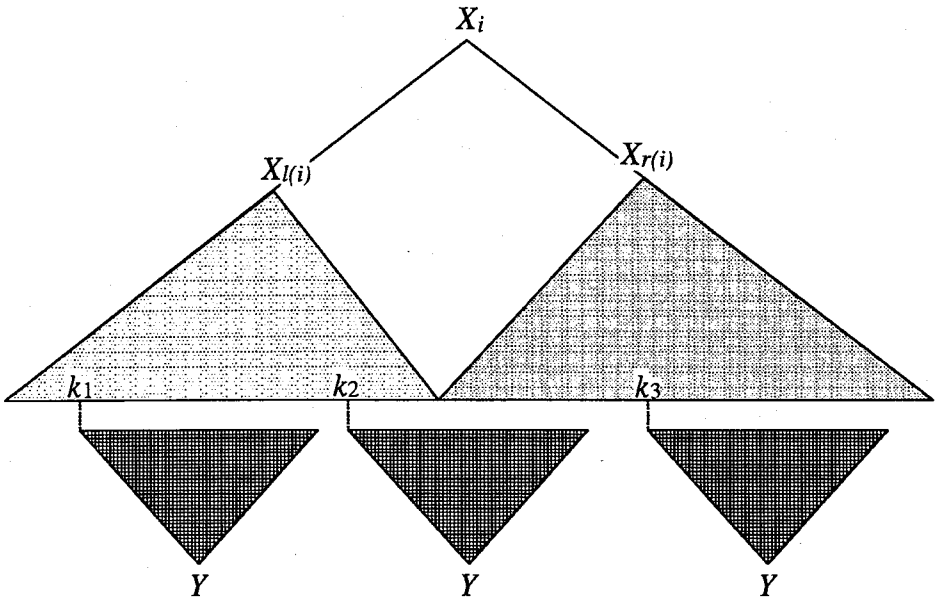

FIG. 3: $k_1, k_2, k_3 \in Occ(X_i, Y)$, while $k_1 \in Occ(X_{\ell(i)}, Y)$, $k_2 \in Occ^\star(X_i, Y)$, and $k_3 - |X_{\ell(i)}| \in Occ(X_{r(i)}, Y)$.

We have the following observation (see Fig. 3):

OBSERVATION 3.2 (decomposition of text variables)
For $X_i = X_{\ell(i)} \cdot X_{r(i)}$ in $\mathcal{T}$ and Y in $\mathcal{P}$,

$$Occ(X_i, Y) = Occ^\star(X_i, Y) \cup Occ(X_{\ell(i)}, Y) \cup (Occ(X_{r(i)}, Y) \oplus |X_{\ell(i)}|).$$

The above observation suggests that $Occ(X_n, Y)$ can be represented by a combination of the sets $\{Occ^\star(X_i, Y)\}_{i=1}^n = Occ^\star(X_1, Y), Occ^\star(X_2, Y), \ldots, Occ^\star(X_n, Y)$. By Lemma 3.1, each $Occ^\star(X_i, Y)$ forms a single arithmetic progression, which can be stored in $O(1)$ space as a triple of the first element, the last element, and the step of the progression. We remark that in our definition of $Occ^\star$, the occurrence of Y in X at position $|X_\ell| - |Y| + 1$ can be listed both in $Occ^\star(X, Y)$ and in $Occ(X_\ell, Y)$. It does not cause a big problem since we can eliminate the duplication in constant time. Thus the desired output, a compact representation of the set $Occ(T, P) = Occ(X_n, Y_m)$ is given as a combination of $\{Occ^\star(X_i, Y_m)\}_{i=1}^n$, which occupies $O(n)$ space. Moreover, as we will show in Lemma 5.3 in Section 5, the membership to the set $Occ(X_i, Y_j)$ can be answered in $O(depth(X_i)) = O(n)$ time using this representation. Therefore the computation of the set $Occ(T, P)$ is reduced to the computation of each set $Occ^\star(X_i, Y_m)$, $i = 1, \ldots, n$. The next observation gives us a recursive procedure to compute the set $Occ^\star(X_i, Y_j)$ (see Fig. 4):

OBSERVATION 3.3 (decomposition of pattern variables)
For X_i in $\mathcal{T}$ and $Y_j = Y_{\ell(j)} \cdot Y_{r(j)}$ in $\mathcal{P}$,

$$\begin{aligned} Occ^\star(X_i, Y_j) &= Occ^\star_\ell(X_i, Y_j) \cup Occ^\star_r(X_i, Y_j), \text{where} \\ Occ^\star_\ell(X_i, Y_j) &= Occ^\star(X_i, Y_{\ell(j)}) \cap (Occ(X_i, Y_{r(j)}) \ominus |Y_{\ell(j)}|), \text{ and} \\ Occ^\star_r(X_i, Y_j) &= Occ(X_i, Y_{\ell(j)}) \cap (Occ^\star(X_i, Y_{r(j)}) \ominus |Y_{\ell(j)}|). \end{aligned}$$

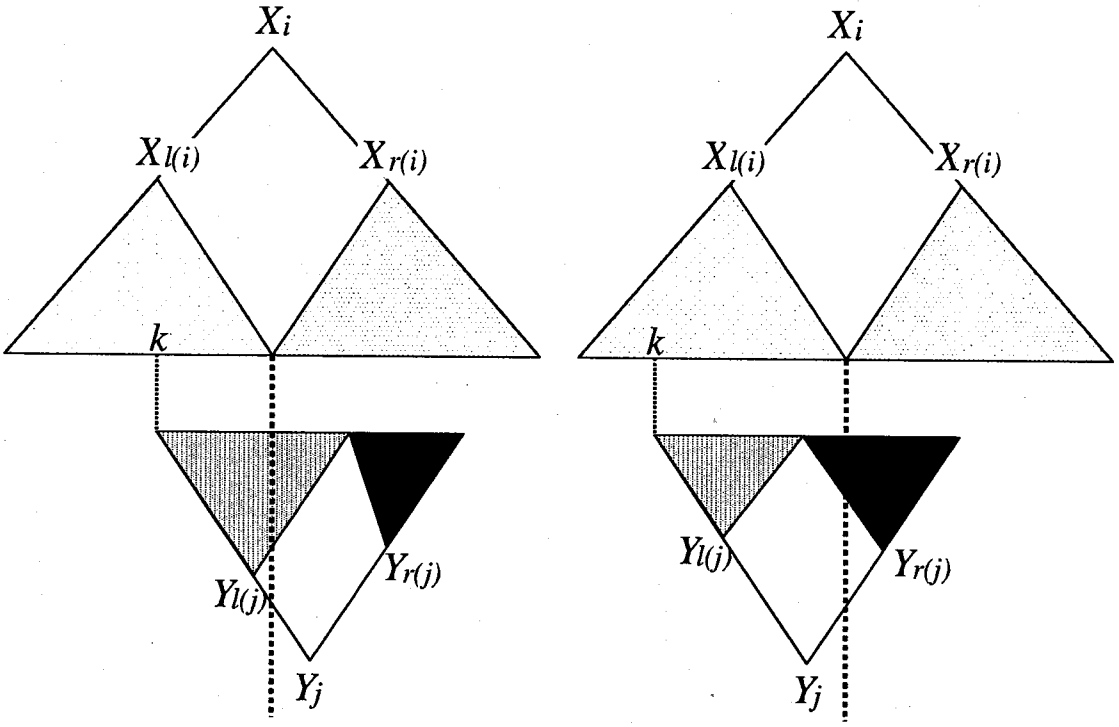

FIG. 4: $k \in Occ^{\star}(X_i, Y_j)$ if and only if either $k \in Occ^{\star}(X_i, Y_{\ell(j)})$ and $k + |Y_{\ell(j)}| \in Occ(X_i, Y_{r(j)})$ (left case), or $k \in Occ(X_i, Y_{\ell(j)})$ and $k + |Y_{\ell(j)}| \in Occ^{\star}(X_i, Y_{r(j)},)$ (right case).

The problem to be overcome is to perform the set operations, union and intersection efficiently, since each set possibly contains exponentially many elements.

Lemma 4.1 in the next section is a key to solving this problem. The key lemma concerns with the periodicities in strings. It guarantees that each of $Occ^{\star}_{\ell}(X_i, Y_j)$ and $Occ^{\star}_{r}(X_i, Y_j)$ forms a single arithmetic progression. This enables us to perform the union operation of these two sets in $O(1)$ time. At the same time, the key lemma gives us a basis to construct an efficient procedure of computing $Occ^{\star}_{\ell}(X_i, Y_j)$ from $Occ^{\star}(X_i, Y_{\ell(j)})$, assuming the function *FirstMismatch* which returns the first position of the mismatches between X_i and $Y_{r(j)}$. We can compute the set $Occ^{\star}_{r}(X_i, Y_j)$ in the same way. In Section 5, we will explain these procedures in detail.

When computing each $Occ^{\star}(X_i, Y_j)$ recursively, we may often refer to the same set $Occ^{\star}(X_{i'}, Y_{j'})$ repeatedly for $i' < i$ and $j' < j$. We take the dynamic programming strategy. Let us consider an $n \times m$ table *App* where each entry $App[i, j]$ at row i and column j stores the triple representing the set $Occ^{\star}(X_i, Y_j)$. We compute each $App[i, j]$ in bottom-up manner, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. As we will show in Lemma 5.5 in Section 5, each $App[i, j]$ is computable in $O(depth(X_i) \cdot depth(Y_j))$ time. Since $depth(X_i) \leq n$ and $depth(Y_j) \leq m$ for any X_i and Y_j, we can construct the whole table *App* in $O(n^2 m^2)$ time. The size of the whole table is $O(nm)$, since each triple occupies $O(1)$ space. Hence we have the main theorem of this paper.

THEOREM 3.4
Given two straight-line programs $\mathcal{T}$ and $\mathcal{P}$, we can compute an $O(n)$ size representation of the set $Occ(T, P)$ of all occurrences of the pattern P in the text T, in $O(n^2m^2)$ time using $O(nm)$ work space. For this representation, the membership to the set $Occ(T, P)$ can be determined in $O(n)$ time.

4 Key lemma

This section shows the key lemma on a property of periodic occurrences of a pattern in a text, which our algorithm based on. Let T and P be strings of a text and a pattern. At first we define the function *FirstMismatch*(T, P, k) which returns the first (leftmost) position of mismatches, when we compare P with T at position k. Formally,

$$\mathit{FirstMismatch}(T, P, k) = \min\{1 \leq i \leq |P| \ : \ T[k+i-1] \neq P[i]\},$$

for $1 \leq k \leq |T| - |P| + 1$. The value is a witness of $k \notin Occ(T, P)$. If there is no such i, we define *FirstMismatch*$(T, P, k) = nil$.

LEMMA 4.1 (Key Lemma)
Let $T = u^l z$ $(u, z \in \Sigma^+, l \geq 0)$ and $P \in \Sigma^+$. The set $S = Occ(T, P) \cap \{1 + i|u| \ : \ i = 0, 1, \ldots, l\}$ forms a single arithmetic progression, which can be computed by at most three calls of *FirstMismatch*.

PROOF. Let A be the set $\{1+i|u| \ : \ i = 0, 1, \ldots, l\} \cap \{1, 2, \ldots, |T|-|P|+1\}$. Since $S = Occ(T, P) \cap A$, we have only to check whether $j \in Occ(T, P)$ for each $j \in A$. If A consists of at most three elements, we can trivially check it by directly calling *FirstMismatch*(T, P, j) for each $j \in A$. We now consider the cases that A contains more than four elements. The basic idea is as follows. Let t_1 be the first position at which T violates the form u^k. If T is of the form u^k, let $t_1 = nil$. Similarly, we define p_1 for P. First we try to infer t_1 and p_1 by twice calls of *FirstMismatch*. If both t_1 and p_1 are *nil*, we know $S = A$, since both T and P are of the form u^k. If $t_1 = nil$ and $p_1 \neq nil$, we have $S = \phi$ since P never matches with T. If $t_1 \neq nil$ and $p_1 = nil$, we see that $S = \{j \in A \mid j \leq t\}$ for some t since P only appears in the prefix of T. Finally, if neither t_1 nor p_1 is *nil*, P matches with T at most one position, and we can verify it by the third call of *FirstMismatch*. The essential idea is similar to the one introduced in [12, 13]. However, unfortunately, we cannot always identify both t_1 and p_1 exactly in our situation. Nevertheless, we can compute the set S based on the above idea.

Let h be the maximum element in A. At the beginning, we invoke the function *FirstMismatch* for two positions 1 and h as follows:

$$\begin{aligned} \mathit{miss}_1 &= \mathit{FirstMismatch}(T, P, 1), \text{ and} \\ \mathit{miss}_2 &= \mathit{FirstMismatch}(T, P, h). \end{aligned}$$

Note that $1 \leq \mathit{miss}_1, \mathit{miss}_2 \leq |P|$, if not *nil*. It is convenient that we regard *nil* as $|P| + 1$. Depending on the values of miss_1 and miss_2, we have six cases as shown

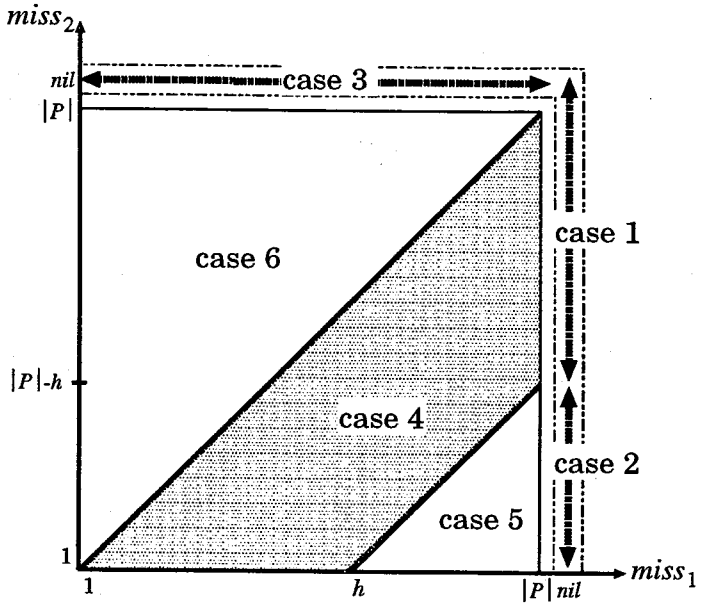

FIG. 5: Six cases depending on $miss_1$ and $miss_2$. (Cases 2 and 5 are vacant if $h > |P|$.)

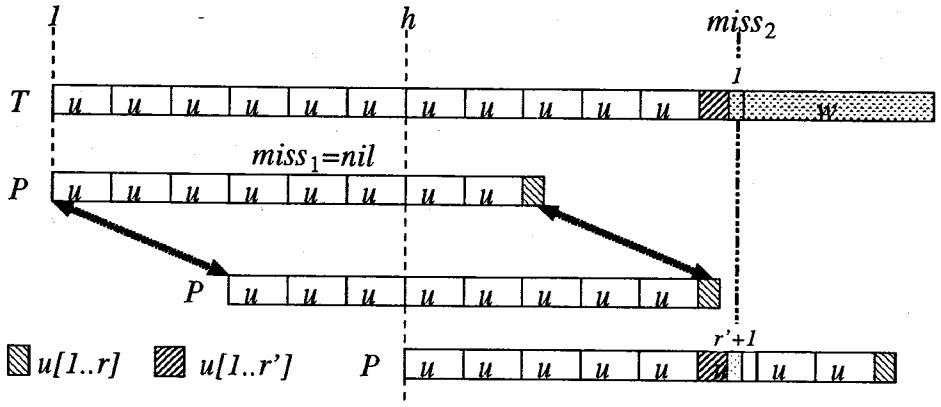

FIG. 6. Case 1, $miss_1 = nil$ and $miss_2 = nil$ or $|P| - h + 1 < miss_2$.

in Fig. 5. We will explain only the first case in detail, since the other cases would be understood similarly.

We use the following notation in the rest of the proof. For two integers a and b, we denote by $\langle q, r\rangle = div(a, b)$ that q is the quotient and r is the remainder of the division of a by b. That is, $a = b \cdot q + r$ and $0 \leq r < b$.

case 1: (Fig. 6) $miss_1 = nil$ and $miss_2 = nil$ or $|P| - h + 1 < miss_2$.

Let $\langle q, r\rangle = div(|P|, |u|)$ and $\langle q', r'\rangle = div(h + miss_2 - 2, |u|)$. First we show the following equations.

$$P = u^q u[1..r] \text{ and} \tag{4.1}$$

$$T = u^{q'} u[1..r']w \text{ for some } w \in \Sigma^+ \text{ with } u[r' + 1] \neq w[1] \tag{4.2}$$

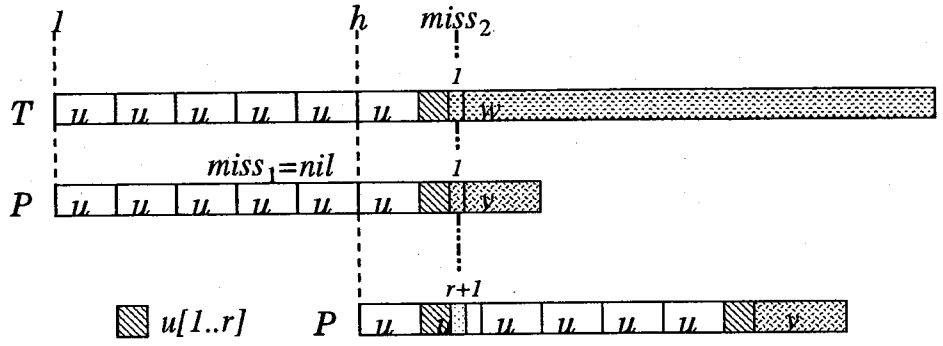

FIG. 7. Case 2, $miss_1 = nil$ and $miss_2 \leq |P| - h + 1$.

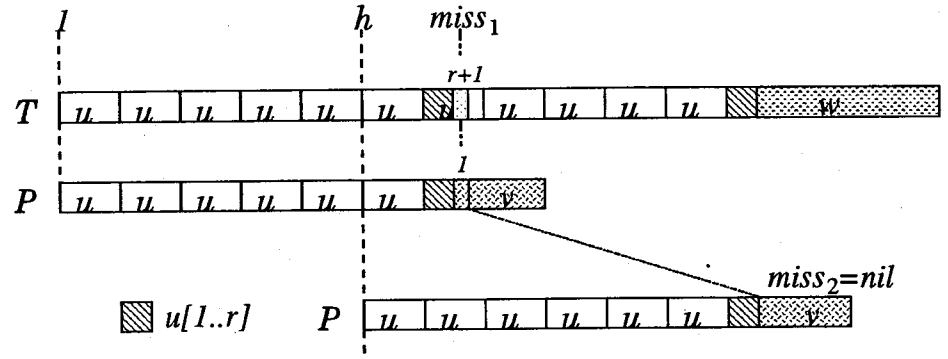

FIG. 8. Case 3, $miss_1 \neq nil$ and $miss_2 = nil$.

Let d be the quotient of the division of $h-1$ by $|u|$. Remind that $T[1:h-1] = u^d$ from the definition of h. Since $miss_1 = FirstMismatch(T, P, 1) = nil$, we know $P[1:|P|] = T[1:|P|]$. Since $miss_2 = FirstMismatch(T, P, h)$, we have $T[h : h + miss_2 - 2] = P[1 : miss_2 - 1]$ and $T[h + miss_2 - 1] \neq P[miss_2]$. Therefore, the equations 4.1 and 4.2 hold when $|P| \leq h - 1$. On the other hand, when $|P| > h - 1$, since $miss_2 > |P| - h + 1$ (or nil, which is treated as $|P| + 1$), we have $P[h : |P|] = T[h : |P|] = P[1 : |P| - h + 1]$, that implies that $h - 1$ is a period of P. Since $T[1 : h - 1] = P[1 : h - 1] = u^d$, the equations 4.1 and 4.2 hold.

We now see that $S = \{1 + i|u| \ : \ i \in \{0, .., t\}\}$, where $t = q' - q$ if $r' \geq r$ and $t = q' - q - 1$ otherwise. This is immediate from the equations 4.1 and 4.2. We note that such t can be directly computed by $\langle t, r'' \rangle = div(h+miss_2-|P|-2, |u|)$. Totally, twice calls of *FirstMismatch* are enough to determine the set S.

case 2: (Fig. 7) $miss_1 = nil$ and $miss_2 \leq |P| - h + 1$. (This is impossible if $h > |P|$).

Let $\langle q, r \rangle = div(h + miss_2 - 2, |u|)$. We can show that $P = u^q u[1..r]v$ and $T = u^q u[1..r]w$ for some $v, w \in \Sigma^+$ such that $u[r+1] \neq v[1] = w[1]$ and v is a prefix of w. Thus we have $S = \{1\}$.

case 3: (Fig. 8) $miss_1 \neq nil$ and $miss_2 = nil$.

Let $\langle q, r \rangle = div(miss_1 - 1, |u|)$, and $\langle q', r' \rangle = div(h+miss_1-2, |u|)$. We can show

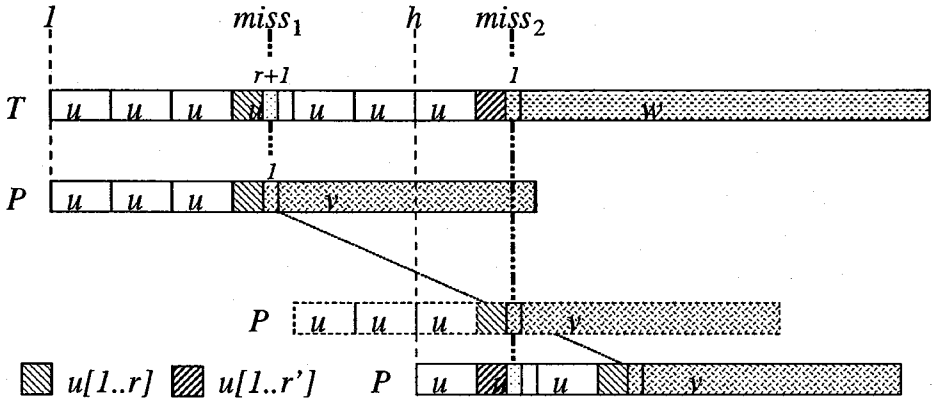

FIG. 9. Case 4, $miss_1 - h + 1 < miss_2 < miss_1$

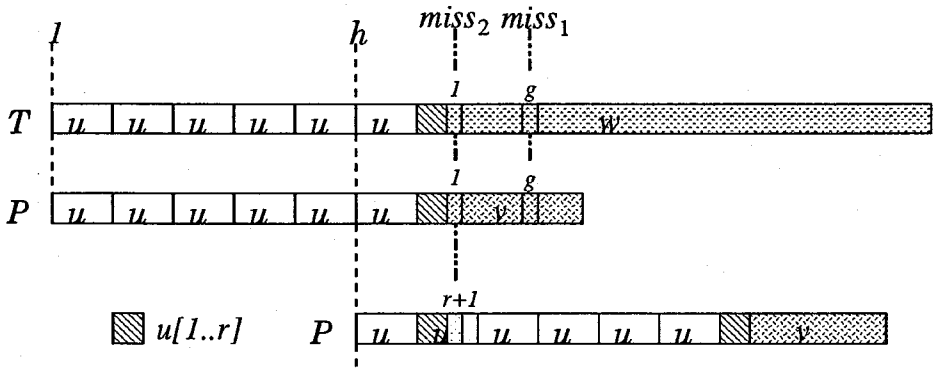

FIG. 10. Case 5, $miss_2 \leq miss_1 - h + 1$.

that $r = r'$, $P = u^q u[1..r]v$ and $T = u^{q'} u[1..r]w$ for some $v, w \in \Sigma^+$ such that $u[r+1] \neq v[1]$ and v is a prefix of w. Thus we have $S = \{h\}$.

case 4: (Fig. 9) $miss_1 - h + 1 < miss_2 < miss_1$.

Let $\langle q, r\rangle = div(miss_1 - 1,\ |u|)$ and $\langle q', r'\rangle = div(h + miss_2 - 2,\ |u|)$. We can show that $P = u^q u[1..r]v$ and $T = u^{q'} u[1..r']w$ for some $v, w \in \Sigma^+$ such that $u[r+1] \neq v[1]$ and $u[r'+1] \neq w[1]$. If $r = r'$ and v is a prefix of w, then S is a singleton of $s = 1 + l|u| - miss_1 + miss_2$. Otherwise $S = \phi$. That is, the only candidate for the elements in S is s. We can verify whether $S = \{s\}$ or $S = \phi$ by the third call of *FirstMismatch*(T, P, s): If *FirstMismatch*$(T, P, s) = nil$, we have $S = \{s\}$, and otherwise, $S = \phi$. Only in this case, we need to call *FirstMismatch* three times.

case 5: (Fig. 10) $miss_2 \leq miss_1 - h + 1$. (This is impossible if $h > |P|$).

Let $\langle q, r\rangle = div(h + miss_2 - 2,\ |u|)$, and $s = miss_1 - h - miss_2 + 2$. Since we can show that $P = u^q u[1..r]v$ and $T = u^q u[1..r]w$ for some $v, w \in \Sigma^+$ such that $u[r+1] \neq v[1] = w[1]$ and $v[s] \neq w[s]$, we have $S = \phi$.

case 6: (Fig. 11) $miss_1 \leq miss_2$.

Let $\langle q, r\rangle = div(miss_1 - 1,\ |u|)$, and $\langle q', r'\rangle = div(h + miss_1 - 2,\ |u|)$. Let $s =$

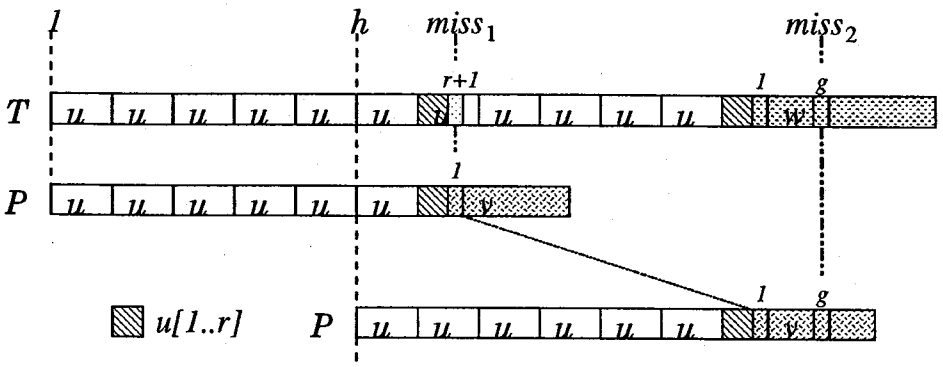

FIG. 11. Case 6, $miss_1 \leq miss_2$.

$miss_2 - miss_1 + 1$. Since $r = r'$, $P = u^q u[1..r]v$ and $T = u^{q'} u[1..r]w$ for some $v, w \in \Sigma^+$ such that $u[r+1] \neq v[1]$ and $v[s] \neq w[s]$, we have $S = \phi$.

For any case, S forms a single arithmetic progression, and we can compute its representation by calling *FirstMismatch* at most three times. ∎

5 Algorithm in detail

In this section, we explain the details on the algorithm. That is, how to compute each entry $App[i, j]$ of the table, which represents the set $Occ^*(X_i, Y_j)$. The computation is done in bottom-up manner.

If either X_i or Y_j is a symbol, we can compute the entry $App[i, j]$ in a trivial way. We show how to compute $App[i, j]$ for $X_i = X_{\ell(i)} \cdot X_{r(i)}$ and $Y_j = Y_{\ell(j)} \cdot Y_{r(j)}$, assuming that all preceding entries $App[i', j']$ for $i' < i$ and $j' < j$ are already computed. We can also assume that we know all lengths $|X_{i'}|$ and $|Y_{j'}|$. As we have explained in Section 3, the critical point is the computation of $Occ^*_\ell(X_i, Y_j) = Occ^*(X_i, Y_{\ell(j)}) \cap (Occ(X_i, Y_{r(j)}) \ominus |Y_{\ell(j)}|)$.

LEMMA 5.1
Independently of the cardinality of the set $Occ^*(X_i, Y_{\ell(j)})$, we can compute the set $Occ^*_\ell(X_i, Y_j)$ by using the function $FirstMismatch(X_i, Y_{r(j)}, k)$ at most three times.

PROOF. If the cardinality of the set $Occ^*(X_i, Y_{\ell(j)})$ is at most two, we can compute the set $Occ^*_\ell(X_i, Y_j)$ easily: For each $s \in Occ^*(X_i, Y_{\ell(j)})$, check whether or not $s \in Occ(X_i, Y_{r(j)}) \ominus |Y_{\ell(j)}|$ by using $FirstMismatch(X_i, Y_{r(j)}, s + |Y_{\ell(j)}|)$.

For the case that the set $Occ^*(X_i, Y_{\ell(j)})$ contains more than two positions, we apply Lemma 4.1 as follows. Let $e_{\min}$ and $e_{\max}$ be the minimum and the maximum elements in $Occ^*(X_i, Y_{\ell(j)}) \oplus |Y_{\ell(j)}|$, respectively (Fig. 12).

Let d be the step of the arithmetic progression of $Occ^*(X_i, Y_{\ell(j)})$, and let $l = (e_{\max} - e_{\min})/d$. Then we can see that the string $X_i[e_{\min}..|X_i|]$ is of the form $u^l z$, where u is the suffix of $Y_{\ell(j)}$ of length d. By Lemma 4.1, we can compute the set

$$S = Occ(X_i[e_{\min}..|X_i|], Y_{r(j)}) \cap \{1, 1+d, \ldots, 1+l \cdot d\}$$

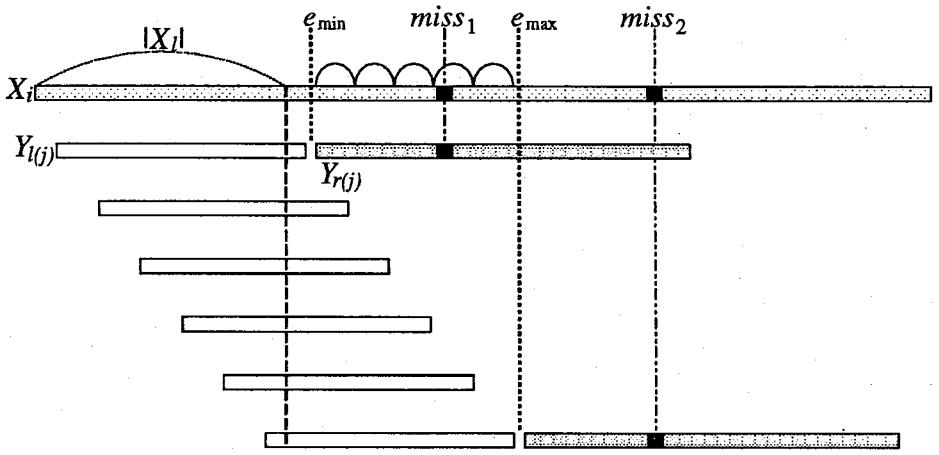

FIG. 12. *FirstMismatch*$(X_i, Y_{r(j)}, e_{\min})$ and *FirstMismatch*$(X_i, Y_{r(j)}, e_{\max})$.

by calling the function *FirstMismatch*$(X_i, Y_{r(j)}, k)$ at most three times. Since

$$\begin{aligned} & S \oplus (e_{\min} - 1) \\ &= (Occ(X_i[e_{\min}..|X_i|], Y_{r(j)}) \cap \{1, 1+d, \ldots, 1+l\cdot d\}) \oplus (e_{\min} - 1) \\ &= (Occ(X_i[e_{\min}..|X_i|], Y_{r(j)}) \oplus (e_{\min} - 1)) \cap \{e_{\min}, e_{\min}+d, \ldots, e_{\min}+l\cdot d\} \\ &= Occ(X_i, Y_{r(j)}) \cap (Occ^*(X_i, Y_{\ell(j)}) \oplus |Y_{\ell(j)}|), \end{aligned}$$

we have $S \oplus (e_{\min} - 1 - |Y_{\ell(j)}|) = (Occ(X_i, Y_{r(j)}) \ominus |Y_{\ell(j)}|) \cap Occ^*(X_i, Y_{\ell(j)}) = Occ^*_\ell(X_i, Y_j)$, which is the desired set. ∎

We show how to realize the function *FirstMismatch*(X, Y, k) for variables X in $\mathcal{T}$ and Y in $\mathcal{P}$ and an integer k. Remark the following recursive property:

OBSERVATION 5.2
For two variables X in $\mathcal{T}$ and Y with $Y = Y_\ell \cdot Y_r$ in $\mathcal{P}$,

$$FirstMismatch(X, Y, k) = \begin{cases} FirstMismatch(X, Y_\ell, k) & \text{if } k \notin Occ(X, Y_\ell), \\ |Y_\ell| + FirstMismatch(X, Y_r, k) & \text{if } k \in Occ(X, Y_\ell). \end{cases}$$

We show a pseudo-code of the function *FirstMismatch* in Fig. 13, where the function *Match*(X, Y, k) returns true if and only if $k \in Occ(X, Y)$. The correctness of *Match*(X, Y, k) is directly derived from Observation 3.2.

LEMMA 5.3
The function *Match*(X_i, Y_j, k) answers in $O(depth(X_i))$ time.

PROOF. The membership query of the form $k \in Occ^*(X_{i'}, Y_{j'})$ can be answered in $O(1)$ time by simple calculations for any $i' < i$ and $j' < j$, since it is already computed and stored in the entry $App[i', j']$. Moreover, the number of recursive calls of *Match*(X_i, Y_j, k) is at most $depth(X_i)$. Thus the lemma holds. ∎

LEMMA 5.4
The function *FirstMismatch*(X_i, Y_j, k) answers in $O(depth(X_i) \cdot depth(Y_j))$ time.

```
function FirstMismatch(X, Y, k): integer;
/* returns the minimum s such that X[k + s − 1] ≠ Y[s] if exists,
     and nil otherwise */
begin
    if |Y| = 1 then
        if X[k] = Y then return 1 else return nil
    else      /* assume Y = Y_ℓ · Y_r */
        if Match(X, Y_ℓ, k) then
            return |Y_ℓ| + FirstMismatch(X, Y_r, k + |Y_ℓ|)
        else
            return FirstMismatch(X, Y_ℓ, k)
end

function Match(X, Y, k): boolean;
/* returns true iff X[k..k + |Y| − 1] = Y. */
begin
    if (k < 0) or (|X| < k + |Y|) then return false;
    if |X| = 1 then
        if Y = X then return true else return false
    else      /* assume X = X_ℓ · X_r */
        if k + |Y| < |X_ℓ| then return Match(X_ℓ, Y, k)
        else if |X_ℓ| < k then return Match(X_r, Y, k − |X_ℓ|)
        else
            if k ∈ Occ*(X, Y) then return true
            else return false
end
```

FIG. 13. Pseudo-codes of the functions *FirstMismatch* and *Match*.

PROOF. The number of recursive calls of the function $FirstMismatch(X_i, Y_j, k)$ is at most $depth(Y_j)$. At each call, the function $Match(X_i, Y_j, k)$ is called once. By Lemma 5.3, it answers in $O(depth(X_i))$ time. Thus the lemma holds. ∎

By Lemma 5.1 and Lemma 5.4, we have the following result.

LEMMA 5.5
Each entry $App[i, j]$ is computable in $O(depth(X_i) \cdot depth(Y_j))$ time.

6 Performance Comparison

First we compare the performance of our algorithm with the previous ones [12, 13], from the theoretical view points. Table 1 shows time complexity and space complexity.

TABLE 1. Summary

algorithm	time	space
KRS'95 [12]	$O((n+m)^7)$	not estimated
KRS'97 [13]	$O((n+m)^4 \log (n+m))$	$O((n+m)^3)$
Ours	$O(n^2m^2)$	$O(nm)$

We briefly state the improvement of our algorithm compared to the one in [13]. The latter algorithm consists of two phases: At the first phase, it computes two sets: $\mathit{Pref}(X_i, Y_j)$ of the lengths of prefixes of Y_j that are suffixes of X_i, and $\mathit{Suff}(X_i, Y_j)$ of the lengths of suffixes of Y_j that are prefixes of X_i. At the second phases, it computes the set $\mathit{Occ}(X_i, Y_j)$ from $\mathit{Pref}(X_i, Y_j)$ and $\mathit{Suff}(X_i, Y_j)$ by solving certain linear Diophantine equations with using Euclid's algorithm. Each $\mathit{Suff}(X_i, Y_j)$ and $\mathit{Pref}(X_i, Y_j)$ can be stored in $O(\mathit{depth}(X_i) + \mathit{depth}(Y_j))$ space, although $\mathit{Occ}(X_i, Y_j)$ occupies only $O(1)$ space. On the other hand, our algorithm directly computes $\mathit{Occ}(X_i, Y_j)$. The property of periodic occurrences of a pattern in a text shown in the key lemma enabled the direct computation.

Our algorithm uses the first position of mismatches only between text and pattern, whereas the previous algorithm additionally requires text-to-text and pattern-to-pattern comparisons. This is the reason why the previous algorithm makes $(n+m) \times (n+m)$ table with $O(n+m)$ size entries, while our algorithm needs only $n \times m$ table with $O(1)$ size entries. This is also the contribution of the key lemma.

In order to compare our algorithm with the algorithm in [13] from the practical view point, we implemented these two algorithms on a Sun SPARCstation 20. In this experiment, we choose patterns and texts are two series of Fibonacci words [6] defined in Table 2, because of the following two reasons. First, $|T|$ and $|P|$ are exponentially long with respect to $n = \|T\|$ and $m = \|P\|$, since they are the n-th and m-th Fibonacci numbers, respectively. Thus any string matching algorithm which explicitly expands text or pattern inevitably requires exponential time and space. The second reason is due to the difference of pattern detection ability for multiple occurrence. Since our algorithm finds all occurrences of a pattern in a text, the running time naturally depends on the number of occurrences, while the previous algorithm finds only one occurrences. Therefore it is hard to make a fair judgment if a pattern occurs more than once in the text. Fortunately, we can see that Y_{n-1} occurs in X_n exactly once for any n, so that the performance comparison can be made on the same condition.

The experimental results on the running time and work space are shown in Fig. 14 and Fig. 15 respectively, where n varies from 21 to 46 and m is set to $n-1$. We can see that both the running time and work space curves of our algorithm rise much more slowly than those of the previous algorithm.

Let us notice that these two algorithms are much more efficient than any algorithm expanding a text or a pattern. Since the lengths $|X_{46}|$ and $|Y_{45}|$ are approximately 1.8×10^9 and 1.1×10^9, respectively, it is impractical to expand explicitly and keep them on a memory. Moreover, we verified that it takes 152 seconds to execute only character-to-character comparison 1.8×10^9 times on the same machine. That implies

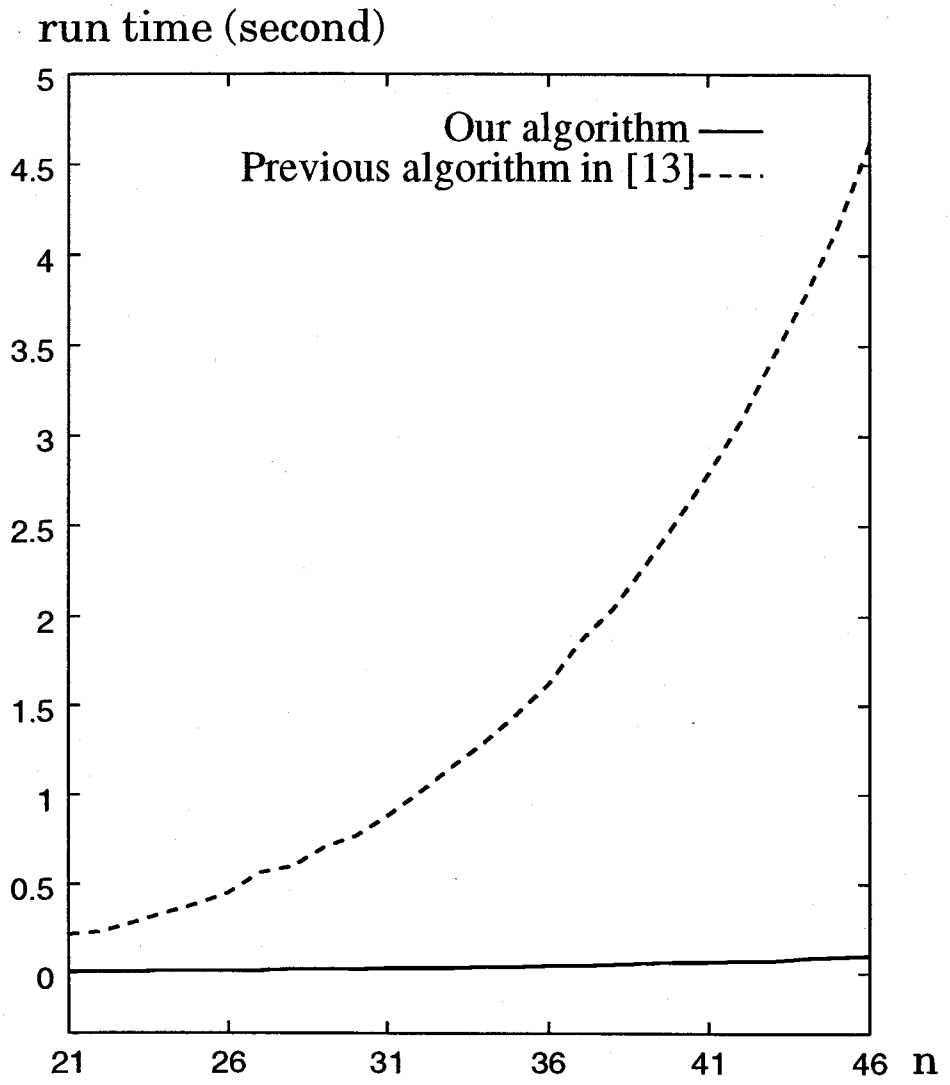

FIG. 14. The comparison of run time

that, once we expand the text explicitly, any string matching algorithm shall consume at least 152 seconds.

On the contrary, even the previous algorithm runs in 4.6 seconds, and our algorithm only 0.1 seconds for $n = 46$. Concerning with work space, the previous algorithm requires 351 kilobytes, while our algorithm uses only 8.6 kilobytes. Therefore we can conclude that these algorithms are efficient enough in practice if the expanded string is long, and our algorithm drastically reduces both running time and work space.

7 Conclusion

In this paper we showed a fast pattern matching algorithm, where both text and pattern are described in terms of straight-line programs. We verified that our algorithm is more

TABLE 2. Text and Pattern

$\mathcal{T}$	$\mathcal{P}$
$X_1 = \mathtt{b}$	$Y_1 = \mathtt{b}$
$X_2 = \mathtt{a}$	$Y_2 = \mathtt{a}$
$X_3 = X_2 \cdot X_1$	$Y_3 = Y_1 \cdot Y_2$
$X_4 = X_3 \cdot X_2$	$Y_4 = Y_2 \cdot Y_3$
$\vdots$	$\vdots$
$\vdots$	$Y_m = Y_{m-2} \cdot Y_{m-1}$
$X_n = X_{n-1} \cdot X_{n-2}$	$(n, m \geq 3)$

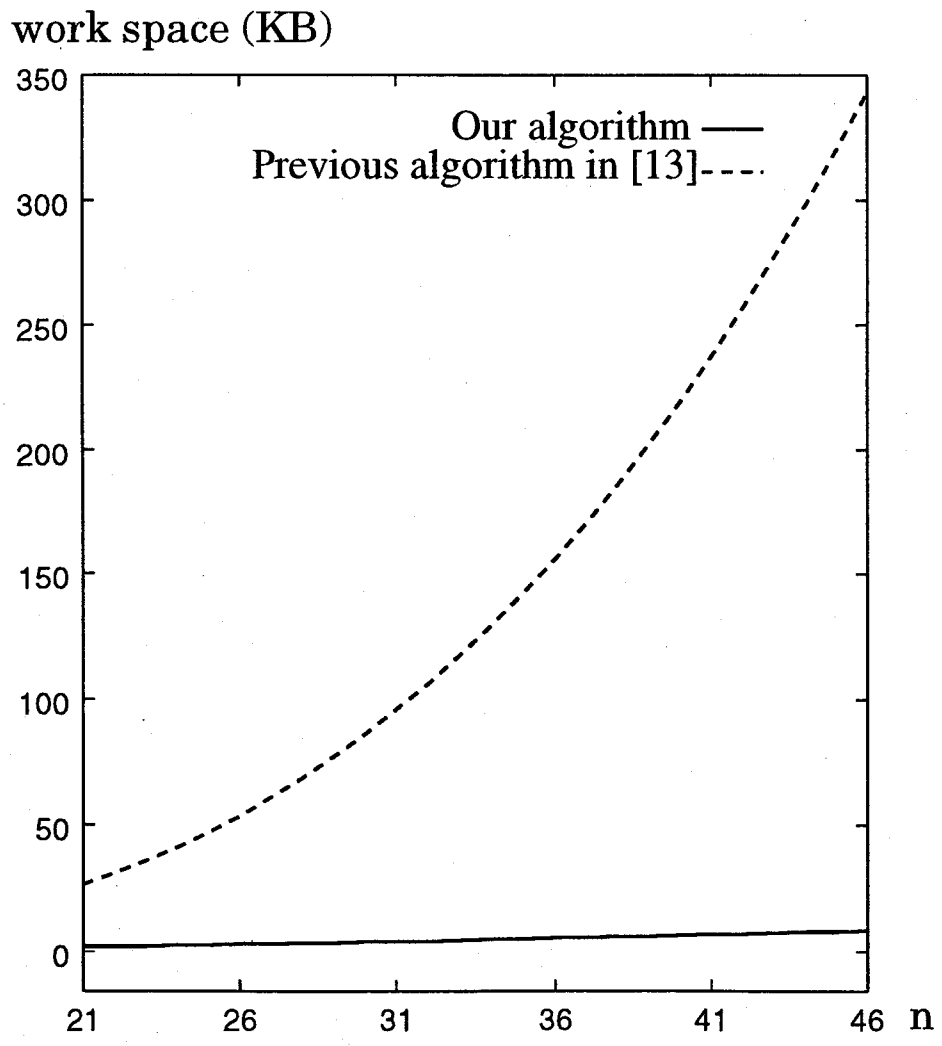

FIG. 15. The comparison of work space

efficient than the previous algorithm in [13] both from theoretical and practical points of view.

Recently, Kida et al. [14] introduced a *Collage System* as a general framework which is suitable to capture an essence of compressed pattern matching according to various dictionary based compressions. They gave a compressed pattern matching algorithm where the text is compressed but the pattern is given explicitly. The framework includes such compression methods as Lempel-Ziv family (LZ77, LZSS, LZ78, LZW), byte-pair encoding, and the static dictionary based method. Since the collage systems contain straight-line programs as a special subclass, their general results gave us a pattern matching algorithm which finds a explicitly given pattern in the text given as a straight-line program. The running time is $O(n + M^2)$, where n is the size of straight-line program for the text and M is the length of the pattern given explicitly. To generalize our results in this paper for collage systems will be interesting future works.

Acknowledgements

We would like to thank the referees for their helpful comments and suggestions.

References

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. Data Compression Conference*, pages 279–288, 1992.

[2] A. Amir and G. Benson. Two-dimensional periodicity and its application. In *Proc. 3rd Symposium on Discrete Algorithms*, pages 440–452, 1992.

[3] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. In *Proc. 21st International Colloquium on Automata, Languages and Programming*, 1994.

[4] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms*, 13(1):2–32, 1992.

[5] Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1996.

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

[7] T. Eilam-Tsoreff and U. Vishkin. Matching patterns in a string subject to multilinear transformations. In *Proc. International Workshop on Sequences, Combinatorics, Compression, Security and Transmission*, 1988.

[8] M. Farach and M. Thorup. String-matching in Lempel-Ziv compressed strings. In *27th ACM STOC*, pages 703–713, 1995.

[9] P. Ferragina and R. Grossi. The string B-Tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[10] Leszek Gąsieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 4th Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403. Springer-Verlag, 1996.

[11] Leszek Gąsieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Randomized efficient algorithms for compressed strings: the finger-print approach. In *Proc. Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 39–49. Springer-Verlag, 1996.

[12] M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In *Proc. Combinatorial Pattern Matching*, volume 637 of *Lecture Notes in Computer Science*, pages 205–214. Springer-Verlag, 1995.

[13] M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing*, 4(2):172–186, 1997.

[14] T. Kida, Y. Shibata, M. Takeda, A.Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th International Symposium on String Processing and Information Retrieval*, pages 89–96, 1999.

[15] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *Siam Journal on Computing*, 22(5):935–948, 1993.

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

[17] W. Rytter. Algorithms on compressed strings and arrays. In *Proc. 26th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM99)*, volume 1725 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

Received November 15, 1999.