Fully Compressed Pattern Matching Algorithm for Balanced Straight-line Programs

Masahiro Hirao   Ayumi Shinohara   Masayuki Takeda   Setsuo Arikawa
Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
{ hirao, ayumi, takeda, arikawa } @i.kyushu-u.ac.jp

Abstract

We consider a fully compressed pattern matching problem, where both text T and pattern P are given by its succinct representation, in terms of straight-line programs and its variant. The length of the text T and pattern P may grow exponentially with respect to its description size n and m, respectively. The best known algorithm for the problem runs in $O(n^2m^2)$ time using $O(nm)$ space. In this paper, we introduce a variant of straight-line programs, called balanced straight-line programs *so that we establish a faster fully compressed pattern matching algorithm. Although the compression ratio of balanced straight-line programs may be worse than the original straight-line programs, they can still express exponentially long strings. Our algorithm runs in $O(nm)$ time using $O(nm)$ space.*

1 Introduction

Pattern matching is a task to find all occurrences of a pattern P in a text T. It is one of the most fundamental problem in string processing. In the last decade, pattern matching on compressed objects attracts more and more interests. The basic problems are the *compressed pattern matching* and the *fully compressed pattern matching* [8]:

Compressed Pattern Matching
Instance: pattern P and $\mathcal{T} = Compress(T)$, representing the compressed text.
Question: does P occur in T ?

Fully Compressed Pattern Matching
Instance: $\mathcal{P} = Compress(P)$ and $\mathcal{T} = Compress(T)$, representing the compressed pattern and compressed text.
Question: does P occur in T ?

For recent developments on this topic, refer to an excellent survey paper [8].

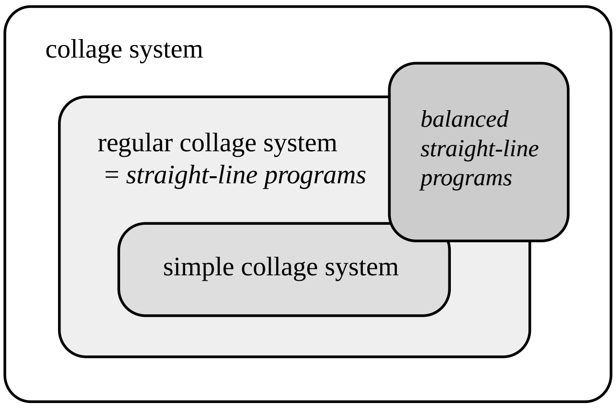

Figure 1. Collage systems and balanced straight-line programs

Studies on (fully) compressed pattern matching have introduced a new measure to compare the performance of various compression methods. In addition to the traditional measures such as compression ratio, compression time and decompression time, a new measure "(fully) compressed pattern matching time" is also considered to be an important factor. For example, byte pair encoding [3], a simple universal text compression scheme based on the pattern substitution, has been reconsidered as a practically good choice from the new viewpoint of compressed pattern matching, despite the fact that the compression ratio is not as good as other competitors such as Lempel-Ziv type compression [9, 10].

In this paper, we concentrate on strings described in terms of straight-line programs and its variant. A straight-line program is a context-free grammar in the Chomsky normal form that derives only one string. It is a sequence of assignments of either a character or a concatenation of two variables defined before. 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.

Table 1. Comparison of the complexity

	Compressed Pattern Matching	Fully Compressed Pattern Matching
Collage Systems [6]	$O(n\ \text{height}(T) + m^2)$ [6]	unknown
Straight-line programs	$O(n + m^2)$ [6]	$O(n^2m^2)$ [7]
Balanced straight-line programs	$O(n + m^2)$ [6]	$O(nm)$

We note that the class of straight-line programs is the central subclass of *collage systems*, that was recently introduced by Kida et al. [6] as a unifying framework for various (not fully) compressed pattern matching algorithms. In addition to the concatenation operation, collage systems have expressions of repetition and truncation. Their results imply that the compressed pattern matching for straight-line programs can be solved in $O(n + m^2)$ time using $O(n+m^2)$ space, where n and m are the description sizes of text and pattern, respectively. According to the fully compressed pattern matching, among several studies [4, 5, 7], the best known algorithm for straight-line programs runs in $O(n^2m^2)$ time using $O(nm)$ space. Concerning with the 2D-text described by 2-dimensional straight-line programs, see [1, 8].

In this paper, we introduce a variant of straight-line programs where faster pattern matching is possible. Here, we only allow concatenations of variables which represent strings of the same length except the last assignment. We call them *balanced* straight-line programs, since the evaluation trees form complete binary trees, except the root node. Because of this restriction, the compression ratio becomes potentially worse than the original straight-line programs, while balanced straight-line programs can still express exponentially long strings. Moreover, compression process in balanced straight-line programs is easier and faster than that in the original straight-line programs.

We show a fully compressed pattern matching algorithm for balanced straight-line programs which runs in $O(nm)$ time using $O(nm)$ space. In this sense, balanced straight-line programs have their own advantage compared with the original straight-line programs as a succinct representation of strings. In Fig. 1, we illustrate the relation of balanced SLP to collage systems and their subclasses. Regular collage systems directly correspond to straight-line programs. Simple collage systems consist of the concatenation of a string with a single character only. Note that LZ78-family compression can be described as simple collage systems. We summarize in Table. 1 the running times of related algorithms.

2 Preliminary

In this section, we introduce basic notations and definitions, and briefly summarize the known results.

A straight-line program $\mathcal{R}$ is a sequence of assignments as follows:

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

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.

In this paper, both text and pattern are described in terms of straight-line programs and its variant.

Denote by $X^{[d]}$ the string obtained by removing the length d suffix from the string represented by variable X. 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.

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

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

It corresponds to the length of the longest path that connects X to a leaf in the evaluation tree.

For a string w denote by $w[f : t] (1 \leq f \leq t \leq |w|)$ the subword of w starting at f and ending at t. We often abbreviate $w[1 : t]$ to $w[: t]$, and $w[f : |w|]$ to $w[f :]$.

The *fully compressed pattern matching* 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 find a compact representation of the following set:

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

Hereafter, we use X_i for a variable in $\mathcal{T}$ and Y_j for a variable in $\mathcal{P}$. We assume $\|\mathcal{T}\| = n$ and $\|\mathcal{P}\| = m$. For

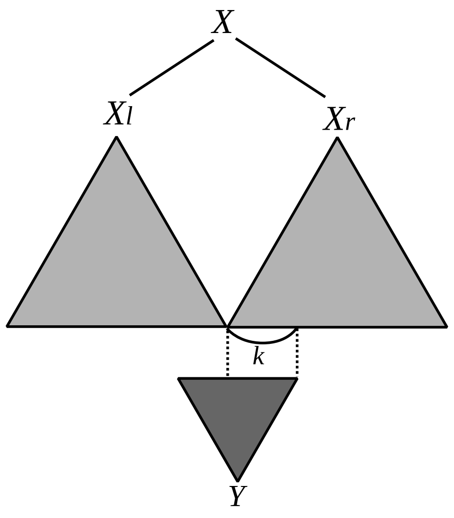

Figure 2. $k \in Occ^{\triangle}(X,Y)$**, since** Y **covers the boundary between** X_ℓ **and** X_r**.**

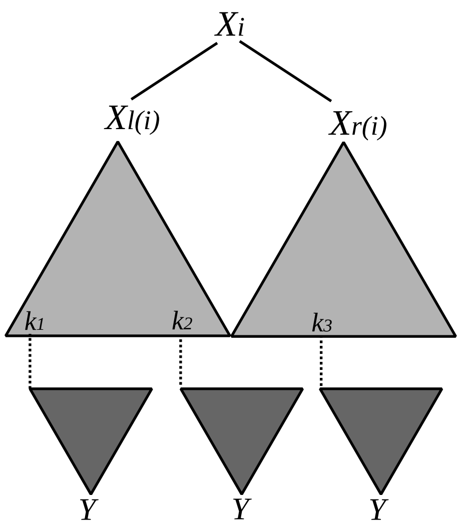

Figure 3. $k_1,\ k_2,\ k_3 \in Occ(X_i,\ Y)$**, while** $k_1 \in Occ(X_{l(i)},\ Y)$**,** $k_2 \in Occ^{\triangle}(X_i,Y) \oplus |X_{l(i)}| \ominus |Y|$ **and** $k_3 \in Occ(X_{r(i)},Y) \oplus |X_{l(i)}|$**.**

a set U of integers and an integer k, we denote $U \oplus k = \{i+k : i \in U\}$ and $U \ominus K = \{i-k : i \in U\}$.

We now give an overview that is common to our algorithm and that in [7]. First we consider a compact representation of the set $Occ(X,Y)$. Suppose $X = X_\ell \cdot X_r$. We define $Occ^{\triangle}(X,Y)$ to be the set of occurrences of Y in X such that Y covers the boundary between X_ℓ and X_r.

$$Occ^{\triangle}(X,Y) =$$

$$\left\{ s+|Y|-|X_\ell|-1 \;\middle|\; \begin{array}{c} s \in Occ(X,Y) \text{ and} \\ |X_\ell|-|Y|+1 \le s \le |X_\ell|+1 \end{array} \right\}.$$

It corresponds to the length from the boundary between X_ℓ and X_r to the right edge of Y (see Fig. 2). For the sake of convenience, let $Occ^{\triangle}(X,Y) = Occ(X,Y)$ for $X = a \in \Sigma$.

The following lemmas hold for any straight-line program.

Lemma 1 (Miyazaki et al. [7]) *For any X in $\mathcal{T}$ and any Y in $\mathcal{P}$, $Occ^{\triangle}(X,Y)$ forms a single arithmetic progression.*

Lemma 2 (Miyazaki et al. [7]) *For $X_i = X_{l(i)} \cdot X_{r(i)}$ in $\mathcal{T}$ and Y in $\mathcal{P}$,*

$$\begin{aligned} Occ(X_i,Y) &= Occ(X_{l(i)},Y) \\ &\cup \{Occ^{\triangle}(X_i,Y) \oplus |X_{l(i)}| \ominus |Y|\} \\ &\cup \{Occ(X_{r(i)},Y) \oplus |X_{l(i)}|\}. \end{aligned}$$

(*See Fig. 3*)

The above lemma suggests that $Occ(X_n,Y)$ can be represented by a combination of $\{Occ^{\triangle}(X_i,Y)\}_{i=1}^{n}$. By Lemma 1, each $Occ^{\triangle}(X_i,Y)$ 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. Thus the desired output, a compact representation of the set $Occ(T,P) = Occ(X_n,Y_m)$ is some combination of $\{Occ^{\triangle}(X_i,Y_m)\}_{i=1}^{n}$, which occupies $O(n)$ space. Therefore the computation of the set $Occ(T,P)$ is reduced to the computation of each set $Occ^{\triangle}(X_i,Y_m),\ i=1,..,n$.

The following is the best known result for fully compressed pattern matching on straight-line programs.

Theorem 1 (Miyazaki et al. [7]) *Fully compressed pattern matching problem for straight-line programs can be solved in $O(n^2m^2)$ time using $O(nm)$ space.*

3 Balanced SLP

In this section, we introduce a variant of straight-line programs we call balanced straight-line programs (Balanced SLP) for which we can develop a faster fully compressed pattern matching algorithm.

Definition 1 *A balanced straight-line programs $\mathcal{B}$ is a sequence of assignments as follows:*

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

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

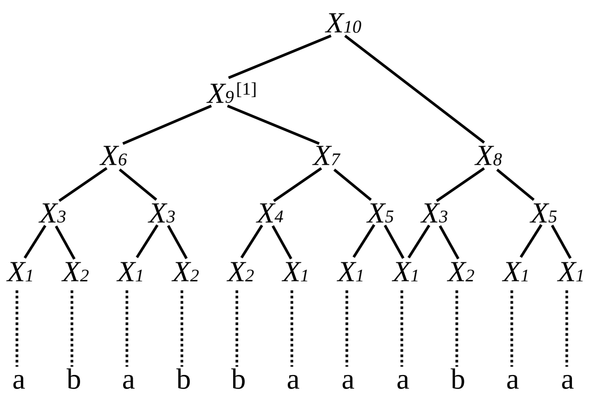

Figure 4. Evaluation tree of $\mathcal{R}$ in Example 1.

- $expr_i$ *is a symbol of a given alphabet* Σ*, or*
- $expr_i = X_\ell \cdot X_r$ *with* $|X_\ell| = |X_r|$, $(\ell, r < i < n)$.
- $expr_n = X_\ell^{[d]} \cdot X_r$ *with* $X_l[|X_l| - d + 1 :] = X_r[: d]$ *and* $d \geq 0$.

Only the variables which represent the same length can be concatenated, except for the last assignment. Therefore, for any string, once we choose the length d of overlap in the last assignment, the corresponding Balanced SLP is uniquely determined.

Example 1 *Let us consider the following balanced straight-line program* $\mathcal{B}$*:*

$$X_1 = \mathtt{a};\ X_2 = \mathtt{b};\ X_3 = X_1 \cdot X_2;\ X_4 = X_2 \cdot X_1;\ X_5 = X_1 \cdot X_1;\ X_6 = X_3 \cdot X_3;\ X_7 = X_4 \cdot X_5;\ X_8 = X_3 \cdot X_5;\ X_9 = X_6 \cdot X_7;\ X_{10} = X_9^{[1]} \cdot X_8.$$

We can see that $\mathcal{B} = X_{10} = \mathtt{ababbaaabaa}$*. The evaluation tree is shown in Fig. 4. Note that the subtrees rooted in* X_9 *and* X_8 *form complete binary trees. This enable us to develop a faster fully compressed pattern matching algorithm.*

We will prove the following main theorem in the rest of this paper.

Theorem 2 *Given two balanced 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* $O(nm)$ *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 Computation for complete binary trees

In this section, we will describe the method to compute $Occ^\triangle(X_i, Y_j)$ for $i < n$ and $j < m$. For $X_i = X_{\ell(i)} \cdot X_{r(i)}$ and $h < height(X_i)$, we recursively define the *rightmost descendant of* X_i *at height* h, denoted by $rmd(X_i, h)$, as follows.

$$rmd(X_i, h) = \begin{cases} rmd(X_{r(i)}, h) & \text{if } height(X_i) > h + 1 \\ X_{r(i)} & \text{if } height(X_i) = h + 1. \end{cases}$$

In the same way, we define the *leftmost descendant of* X_i *at height* h by

$$lmd(X_i, h) = \begin{cases} lmd(X_{l(i)}, h) & \text{if } height(X_i) > h + 1 \\ X_{l(i)} & \text{if } height(X_i) = h + 1. \end{cases}$$

For example, $rmd(X_9, 2) = X_5$, $rmd(X_8, 1) = X_1$, and $lmd(X_6, 2) = X_3$ (in Fig 4). We compute $rmd(X_i, h)$ and $lmd(X_i, h)$ for each variable X_i with $h < height(Y)$ and store them as a table in advance, so that we can look up in $O(1)$ time. The construction of the table can be done in $O(nm)$ time.
The next lemma gives a recursive relation of $Occ^\triangle(X_i, Y_j)$. The proof is rather straightforward from the illustration in Fig. 5.

Lemma 3 *Assume that both* $\mathcal{T}$ *and* $\mathcal{P}$ *are balanced straight-line programs. Let* $X_{\ell'(i)} = rmd(X_{l(i)}, height(Y_j))$ *and* $X_{r'(i)} = lmd(X_{r(i)}, height(Y_j))$*. For* X_i *in* $\mathcal{T}$ *and* $Y_j = Y_{\ell(j)} \cdot Y_{r(j)}$ *in* $\mathcal{P}$*,*

$$Occ^\triangle(X_i, Y_j) = Occ_\ell^\triangle(X_i, Y_j) \cup Occ_r^\triangle(X_i, Y_j), \text{ where}$$

$$\begin{aligned} Occ_\ell^\triangle(X_i, Y_j) &= Occ^\triangle(X_{\ell'(i)}, Y_{\ell(j)}) \\ &\quad \cap Occ^\triangle(X_i, Y_{r(j)}), \text{ and} \\ Occ_r^\triangle(X_i, Y_j) &= Occ^\triangle(X_i, Y_{\ell(j)}) \\ &\quad \cap Occ^\triangle(X_{r'(i)}, Y_{r(j)}) \oplus |Y_{r(j)}|. \end{aligned}$$

Since $Occ^\triangle(X_i, Y_j)$ forms a single arithmetic progression, union operation can be answered in $O(1)$ time. Thus, the problem to be overcome is to perform the intersection operation efficiently. Lemma 5 is a key to solve this problem. It utilizes the periodicity lemma below.

Lemma 4 (Periodicity Lemma (see [2], p. 29)**)** *Let* p *and* q *be two periods of a string* w*. If* $p + q - \gcd(p, q) \leq |w|$*, then the greatest common divisor* $\gcd(p, q)$ *is also a period of* w*.*

Let $\langle a, d, b \rangle$ be the set formed by an arithmetic progression where the first element is a, the step is d, and the last element is b.

Lemma 5 *Assume* $\langle a_1, d_1, b_1 \rangle = Occ^\triangle(X_{\ell'(i)}, Y_{\ell(j)})$ *and* $\langle a_2, d_2, b_2 \rangle = Occ^\triangle(X_i, Y_{r(j)})$ *are given. Then we can compute* $\langle a, d, b \rangle = \langle a_1, d_1, b_1 \rangle \cap \langle a_2, d_2, b_2 \rangle$ *in* $O(1)$ *time.*

Proof. Since it is trivial for the case $d_1 = d_2$, we can assume that $d_1 < d_2$ without loss of generality. We will prove the following two claims.

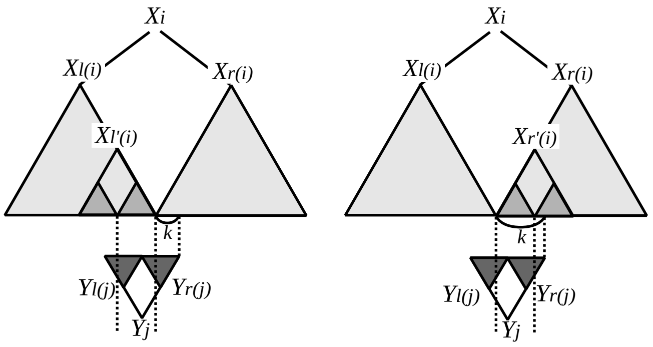

Figure 5. $k \in Occ^{\Delta}(X_i, Y_j)$ **if and only if either** $k - |Y_{r(j)}| \in Occ^{\Delta}(X_{\ell'(i)}, Y_{\ell(j)})$ **and** $k \in Occ^{\Delta}(X_i, Y_{r(j)})$ **(left case), or** $k - |Y_{r(j)}| \in Occ^{\Delta}(X_i, Y_{\ell(j)})$ **and** $k - |Y_{r(j)}| \in Occ^{\Delta}(X_{r'(i)}, Y_{r(j)})$ **(right case).**

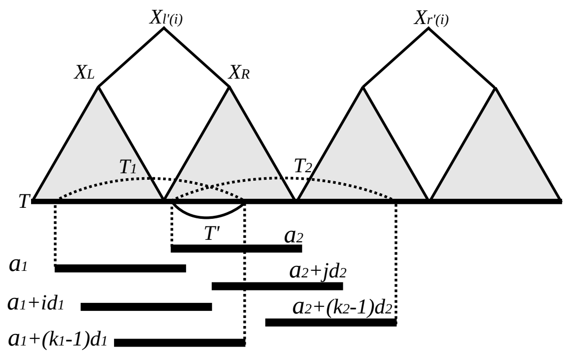

Figure 6. Strings T_1**,** T_2 **and** T'**. Note that** d_1 **is a period of** T_1 **and** d_2 **is a period of** T_2**.**

Claim 1. $\langle a, d, b\rangle$ contains at most one element.

Claim 2. $\langle a, d, b\rangle \subseteq \{a_1, a_2, b_1, b_2\}$.

By Claim 2, we can compute $\langle a, d, b\rangle$ by checking whether $a_1, b_1 \in \langle a_2, d_2, b_2\rangle$ and $a_2, b_2 \in \langle a_1, d_1, b_1\rangle$. Claim 1 guarantees that at most one checking will succeed. Thus we can compute $\langle a, d, b\rangle$ from $\langle a_1, d_1, b_1\rangle$ and $\langle a_2, d_2, b_2\rangle$ in O(1) time. In order to prove these claims, we pay our attention to the strings T_1 and T_2 as follows (See Fig. 6), where $X_{l'(i)} = X_L \cdot X_R$.

$$\begin{aligned} T_1 &= X_{\ell'(i)}[a_1 + 1 : b_1 + |X_L|] \\ T_2 &= X_R[a_2 + 1 :] \cdot X_{r(i)}[: b_2] \end{aligned}$$

Let T' be the common substring of T_1 and T_2. Then $T' = X_R[a_2 + 1 : b_1]$.

Proof of Claim 1. Remark that d_1 and d_2 are the smallest periods of T_1 and T_2, respectively. Suppose $\langle a, d, b\rangle$ contains two elements t_1 and t_2 ($t_1 < t_2$). Since $t_1 \in \langle a_2, d_2, b_2\rangle$ and $t_2 \in \langle a_1, d_1, b_1\rangle$, we have $t_1 \geq a_2$ and $t_2 \leq b_1$. These imply that $|T'| = b_1 - a_2 \geq t_2 - t_1$. We now show that $t_2 - t_1 \geq d_1 + d_2 - \gcd(d_1, d_2)$. Since $t_1, t_2 \in \langle a_1, d_1, b_1\rangle \cap \langle a_2, d_2, b_2\rangle$, we have $t_2 - t_1 = d_1 \cdot c_1 = d_2 \cdot c_2$ for some $c_1, c_2 \geq 1$. We have two cases based on whether c_1 is divided by c_2 or not.

(Case 1) If c_1 is divided by c_2, we have $\gcd(d_1, d_2) = d_1$.

(Case 2) Otherwise, $c_2 > 1$. Then $t_2 - t_1 = d_2 \cdot c_2 \geq 2d_2 > d_1 + d_2$.

In both cases, we have $t_2 - t_1 \geq d_1 + d_2 - \gcd(d_1, d_2)$, which implies $|T'| \geq d_1 + d_2 - \gcd(d_1, d_2)$. Since both d_1 and d_2 are periods of T', we know that $\gcd(d_1, d_2)$ is also a period of T' by Periodicity Lemma. Since $\gcd(d_1, d_2) \leq d_1 < d_2$, this contradicts with the fact d_2 is the smallest period of T_2. Therefore, $\langle a, d, b\rangle$ cannot contain more than one element. ■

Proof of Claim 2. Suppose there exists an element $t \in \langle a, d, b\rangle - \{a_1, a_2, b_1, b_2\}$. Let k_1 and k_2 be the cardinalities of the sets $\langle a_1, d_1, b_1\rangle$ and $\langle a_2, d_2, b_2\rangle$, respectively. Remark that $b_1 = a_1 + (k_1 - 1) \cdot d_1$ and $b_2 = a_2 + (k_2 - 1) \cdot d_2$. Since $\langle a, d, b\rangle = \langle a_1, d_1, b_1\rangle \cap \langle a_2, d_2, b_2\rangle$, for some integers i and j with $1 \leq i \leq k_1 - 2$ and $1 \leq j \leq k_2 - 2$), we have $t = a_1 + i \cdot d_1 = a_2 + j \cdot d_2$. We see that

$$\begin{aligned} |T'| &= b_1 - a_2 \\ &= a_1 + (k_1 - 1) \cdot d_1 - a_2 \\ &= (t - a_2) + a_1 + (k_1 - 1) \cdot d_1 - t \\ &= (a_2 + j \cdot d_2 - a_2) + a_1 \\ &\qquad + (k_1 - 1) \cdot d_1 - (a_1 + i \cdot d_1) \\ &= j \cdot d_2 + (k_1 - 1 - i) \cdot d_1 \\ &\geq d_1 + d_2. \end{aligned}$$

Since both d_1 and d_2 are periods of T', we know that $\gcd(d_1, d_2)$ is also a period of T' by Periodicity Lemma. Since $\gcd(d_1, d_2) \leq d_1 < d_2$, we see $\gcd(d_1, d_2)$ is a smaller period of T_2, which is a contradiction. Therefore, $\langle a, d, b\rangle \subseteq \{a_1, a_2, b_1, b_2\}$. ■

This completes the proof of Lemma 5. ■

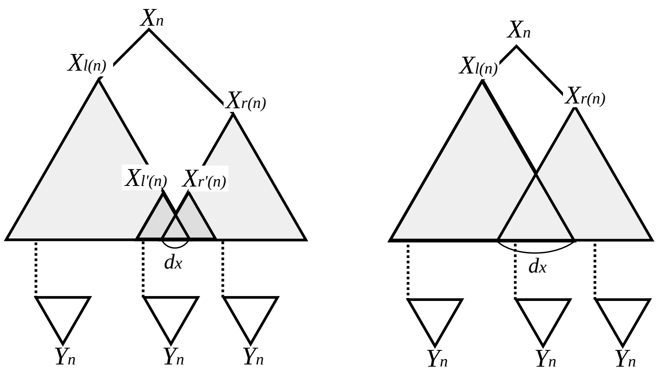

Figure 7. $Occ(X_n, Y_m)$ **in Lemma 6**

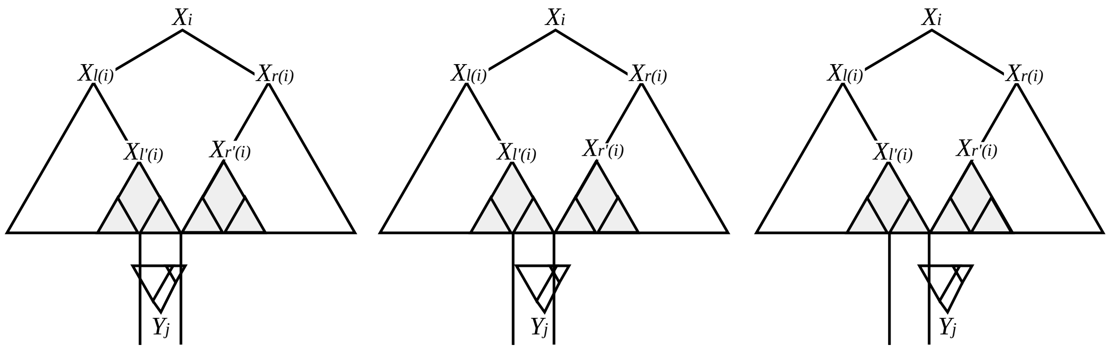

Figure 8. $Occ_1^{\triangle}(X_i, Y_j)$**(left),** $Occ_2^{\triangle}(X_i, Y_j)$**(center) and** $Occ_3^{\triangle}(X_i, Y_j)$**(right)**

The above Lemma 5 enables us to perform the intersection operation to compute $Occ_\ell^{\triangle}(X_i, Y_j)$ and $Occ_r^{\triangle}(X_i, Y_j)$ in $O(1)$ time. When computing each $Occ^{\triangle}(X_i, Y_j)$ recursively, we may often refer to the same set $Occ^{\triangle}(X_i', Y_j')$ repeatedly for $i' < i$ and $j' < j$. We take dynamic programing 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 representation of the set $Occ^{\triangle}(X_i, Y_j)$. We compute each $App[i, j]$ in bottom-up manner, for $i = 1, \ldots, n-1$ and $j = 1, \ldots, m-1$. We can construct the whole table *App* in $O(nm)$. The size of the whole table is $O(nm)$, since each triple occupies $O(1)$ space.

5 Computation for the last assignment

In this section, we will show how to compute $Occ(X_n, Y_m)$ and $Occ^{\triangle}(X_i, Y_j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$.

In the sequel, we assume that $Y_{\ell(j)} > Y_{r(j)}$, since the other case is symmetric. $Occ(X_n, Y_m)$ can be computed by next lemma.

Lemma 6 *For* $X_n = X_{\ell(n)}^{[d_x]} \cdot X_{r(n)}$ *in* $\mathcal{T}$ *and* $Y_m = Y_{\ell(m)}^{[d_y]} \cdot Y_{r(m)}$ *in* $\mathcal{P}$*, the following recursive relation holds.*
$Occ(X_n, Y_m) =$

$$\begin{cases} Occ(X_{\ell(n)}, Y_m) \cup Occ(X_{r(n)}, Y_m) \oplus |X_{\ell(n)}| \ominus d_x \\ \cup Occ(X_{\ell'(n)}^{[d_x]} \cdot X_{r'(n)}, Y_m) \oplus |X_{\ell(n)}| \ominus |X_{\ell'(n)}| \\ \qquad\qquad\qquad\qquad \textit{if } d_x < |Y_m|, \\ Occ(X_{\ell(n)}, Y_m) \cup Occ(X_{r(n)}, Y_m) \oplus |X_{\ell(n)}| \ominus d_x \\ \qquad\qquad\qquad\qquad \textit{if } d_x > |Y_m|. \end{cases}$$

For $1 \leq i \leq n$ *and* $1 \leq j < m$*,*
$Occ(X_i, Y_j) =$

$$\begin{cases} Occ(X_{\ell(i)}, Y_j) \cup Occ(X_{r(i)}, Y_j) \oplus |X_{\ell(i)}| \\ \cup Occ^{\triangle}(X_i, Y_j) \oplus |X_{\ell(i)}| \ominus |Y_j| \\ \qquad\qquad\qquad \textit{if } height(X_i) > height(Y_j), \\ Occ^{\triangle}(X_i, Y_{\ell(j)}) \cap (Occ^{\triangle}(X_i, Y_{r(j)}) \ominus |Y_{r(j)}| \oplus d_y \\ \cup Occ(X_{r(i)}, Y_{r(j)}) \oplus d_y \ominus |Y_{\ell(j)}|) \\ \qquad\qquad\qquad \textit{if } height(X_i) = height(Y_j). \end{cases}$$

$Occ^{\triangle}(X_i, Y_j)$ can be computed by next lemma.

Lemma 7 *For* $X_i = X_{\ell(i)} \cdot X_{r(i)}$ *in* $\mathcal{T}$ *and* $Y_j = Y_{\ell(j)}^{[d_y]} \cdot Y_{r(i)}$ *in* $\mathcal{P}$*, let* $X_{\ell'(i)} = rmd(X_{\ell(i)}, height(Y_j))$ *and* $X_{r'(i)} = lmd(X_{r'(i)}, height(Y_j))$*. Then we have*

$Occ^{\triangle}(X_i, Y) =$
$Occ_1^{\triangle}(X_i, Y_j) \cup Occ_2^{\triangle}(X_i, Y_j) \cup Occ_3^{\triangle}(X_i, Y_j)$*, where*

$$\begin{aligned} Occ_1^{\triangle}(X_i, Y_j) &= Occ^{\triangle}(X_{\ell'(i)}, Y_{\ell(j)}) \ominus d_y \oplus |Y_{r(j)}| \\ &\quad \ominus |Y_{\ell(j)}| \cap\ Occ^{\triangle}(X_i, Y_{r(j)}), \\ Occ_2^{\triangle}(X_i, Y_j) &= Occ^{\triangle}(X_i, Y_{\ell(j)}) \ominus d_y \oplus |Y_{r(j)}| \\ &\quad \cap\ Occ^{\triangle}(X_i, Y_r(j)), \textit{ and} \\ Occ_3^{\triangle}(X_i, Y_j) &= Occ^{\triangle}(X_i, Y_{\ell(j)}) \ominus d_y \oplus |Y_{r(j)}| \\ &\quad \cap Occ(X_{r'(i)}, Y_{r(j)}) \oplus |Y_{r(j)}|. \end{aligned}$$

(See Fig. 8)

$Occ_1^{\triangle}(X_i, Y_j)$ and $Occ_2^{\triangle}(X_i, Y_j)$ can be computed in $O(1)$ time by Lemma 5. Moreover, $Occ_3^{\triangle}(X_i, Y_j)$ can be computed in $O(m)$ time (See *Appendix*). Since each $Occ^{\triangle}(X_i, Y_j)$ forms a single arithmetic progression, the union operation can be computed in $O(1)$ time. Therefore, we can compute each $Occ^{\triangle}(X_i, Y_j)$ in $O(m)$ time. On the other hand, by Lemma 6 we can say that $Occ(X_n, Y_m)$ can be represented by a combination of $\{Occ^{\triangle}(X_i, Y_m)\}_{i=1}^n$ and $Occ(X_{\ell'(n)}^{[d_x]} \cdot X_{r'(n)}, Y_m)$. Both of them can be computed in $O(nm)$ time. Moreover, by Lemma 2 and Lemma 6, the membership to $Occ(X_n, Y_m)$ is answered in $O(n)$ time.

Acknowledgement

We would like to thank Professor Apostolico for insightful comments.

References

[1] P. Berman, M. Karpinski, L. L. Larmore, W. Plandowski, and W. Rytter. On the complex-

ity of pattern matching for highly compressed two-dimensional texts. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*, pages 40–51. Springer-Verlag, 1997.

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

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

[4] M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching*, volume 637 of *Lecture Notes in Computer Science*, pages 205–214. Springer-Verlag, 1995.

[5] 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.

[6] 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.

[7] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching*, volume 1264, pages 1–11, 1997. (to appear in Journal Discrete Algorithms).

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

[9] 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, 2000.

[10] 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*, volume 1848 of *Lecture Notes in Computer Science*, pages 181–194, 2000.

Appendix

In this appendix, we show the detail of the method to compute the set $Occ_3^{\triangle}(X_i, Y_j)$ in Lemma 7 and estimate its cost.

First, we describe the method of computing the following set.

$$S(X,Y,k) = \left\{ t \;\middle|\; \begin{array}{c} X[t : t+|Y|-1] = Y \text{ and} \\ t < k < t+|Y|-1 \end{array} \right\}$$

$S(X,Y,k)$ is the set of the occurrences of a pattern Y that covers some position k in the string X. It also forms a single arithmetic progression. This set can be computed from the following recursive relation and the operation requires $O(\mathit{height}(X))$ time.

$$S(X,Y,k) =$$

$$\begin{cases} S(X_\ell, Y, k) & \text{if } k+|Y| < |X_\ell|, \\ S(X_r, Y, k-|X_\ell|) \oplus |X_\ell| & \text{if } k-|Y| > |X_\ell|, \\ Occ^{\triangle}(X,Y) \oplus |X_\ell| \cup Occ^{\triangle}(rmd(X_\ell, |Y_\ell|+1), Y) & \\ \cap\{k-|Y|, ..., k, ..., k+|Y|\} & \text{if } k \le |X_\ell| \le k+|Y|, \\ Occ^{\triangle}(X,Y) \oplus |X_\ell| \cup Occ^{\triangle}(lmd(X_r, |Y_\ell|+1), Y) & \\ \cap\{k-|Y|, ..., k, ..., k+|Y|\} & \text{if } k-|Y| \le |X_\ell| \le k. \end{cases}$$

Remind that d_y is the length of overlaps between $Y_{\ell(j)}$ and $Y_{r(j)}$. In the equation of $Occ_3^{\triangle}(X_i, Y_j)$ in Lemma 7, we denote $Occ^{\triangle}(X_i, Y_{\ell(j)}) \ominus d_y \oplus |Y_{r(j)}|$ by triple $\langle a, d, k\rangle$ and denote $Occ(X_{r'(i)}, Y_{(j)}) \oplus |Y_{r(j)}|$ by C. The smallest element c in C can be computed from the table App in $O(\mathit{height}(X_{r'(i)}))$ time, by Lemma 2. We now consider the partitions A and B of $\langle a, d, k\rangle$, defined as follows.

$$A = \{x \mid x \in \langle a, d, k\rangle \text{ and } x \le b + |Y_\ell| - d_y\}$$

$$B = \{x \mid x \in \langle a, d, k\rangle \text{ and } x > b + |Y_\ell| - d_y\}$$

Since d is a period of $X[a : b + |Y|]$, we have

$$A \cap C =$$

$$\begin{cases} \{\, x \mid x \in \langle c, d, b\rangle \text{ and } x \le b + |Y_{\ell(j)}| - d_y\} & \text{if } c \in A \\ \phi & \text{other} \end{cases}$$

Moreover, we have

$$B \cap C = B \cap S(X_i, Y_{r(j)}, b + |Y_{\ell(j)}|).$$

Since both of A and B form single arithmetic progressions, we can answer the intersection operation in $O(1)$ time by Lemma 5.

The whole operation runs in $O(\mathit{height}(X_{r'(i)}))$ time. Therefore, we can compute the set of $Occ_3^{\triangle}(X_i, Y_j)$ in $O(m)$ time.