An improved pattern matching algorithm for strings in terms of straight-line programs

Miyazaki, Masamichi
Department of Informatics, Kyushu University

Shinohara, Ayumi
Department of Informatics, Kyushu University

Takeda, Masayuki
Department of Informatics, Kyushu University

出版情報：DOI Technical Report. 130, 1997-01. Department of Informatics, Kyushu University
バージョン：
権利関係：

An improved pattern matching algorithm for strings in terms of straight-line programs

Masamichi Miyazaki Ayumi Shinohara
Masayuki Takeda
{masamich, ayumi, takeda}@i.kyushu-u.ac.jp
Department of Informatics, Kyushu University 33,
Fukuoka 812-81, Japan

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 grows 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 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.*[11], which runs in $O((n+m)^4 \log(n+m))$ time using $O((n+m)^3)$ space, and finds only one occurrence. Moreover, our algorithm is much simpler than theirs.

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 stored in secondary storage, hence most of the time required for pattern matching is devoted to data transmission. If the text is stored in some compressed form, the data transmission time is decreased according to the compression ratio. Text compression thus speeds up pattern-matching. Of course, the processing time (excluding I/O time) may be much longer than searching the original text. Therefore it is important to give an efficient pattern matching algorithm for searching a compressed text directly.

The problem of pattern matching in compressed text is of not only practical interest but also of theoretical interest. It has been studied recently by several researchers for several compression methods. For example, [1, 2, 3, 5, 6] are for the run-length coding, [4] for the LZW coding, [7, 8, 9] for the LZ77 coding.

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 fact, any text compressed by the LZW coding can be transformed directly into a straight-line program within a constant factor.

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.*[10] showed the first

polynomial-time algorithm. Later in [11] 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 complicated and finds only one occurrence of pattern. In this paper we describe a new combinatorial property concerning with the periodic occurrences of pattern in text, and then present an $O(n^2m^2)$ time algorithm using $O(nm)$ space, which is based on this property. Our algorithm is simple and easy to understand, and outputs an $O(n)$ representation of all occurrences of pattern in text.

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 defined to be the number 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 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 $R = X_8 = \mathtt{abaababaababaababa}$, and $\|\mathcal{R}\| = 8$, $|R| = 18$. The evaluation tree is shown in Figure 1.

We introduce a measure *depth* of a variable X in a straight-line program $\mathcal{R}$ defined 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 the node X to a leaf in the tree.

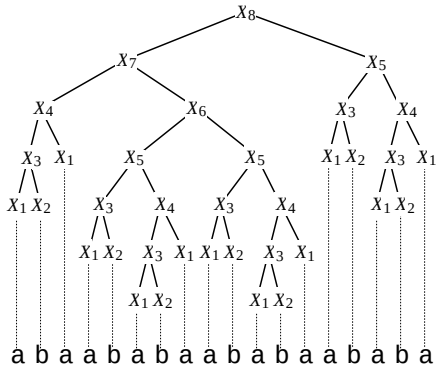


Figure 1: Evaluation tree of $\mathcal{R}$ in Example 1.

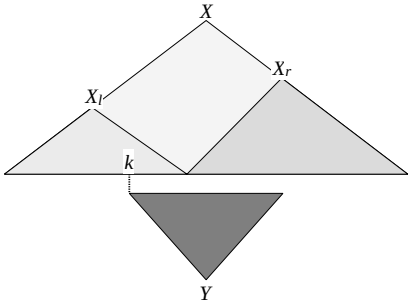


Figure 2: $k \in Occ^{\star}(X, Y)$, since Y covers the boundary between X_ℓ and X_r.

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. 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:

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

Hereafter, we use X_i for a variable in $\mathcal{T}$ and Y_j for a variable in $\mathcal{P}$. We also denote by n and m the sizes of $\mathcal{T}$ and $\mathcal{P}$, respectively. For a set U of integers and an integer k, we denote $U \oplus k = \{i + k : i \in U\}$ and $U \ominus k = \{i - k : i \in U\}$.

3 Overview of algorithm

In this section, we give an overview of our algorithm together with its basic idea. Let X be a variable which appears in $\mathcal{T}$, and Y be a variable in $\mathcal{P}$. First we consider a compact representation of the set $Occ(X, Y)$.

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 the boundary between X_ℓ and X_r (see Figure 2):

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

For the sake of 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 [8].

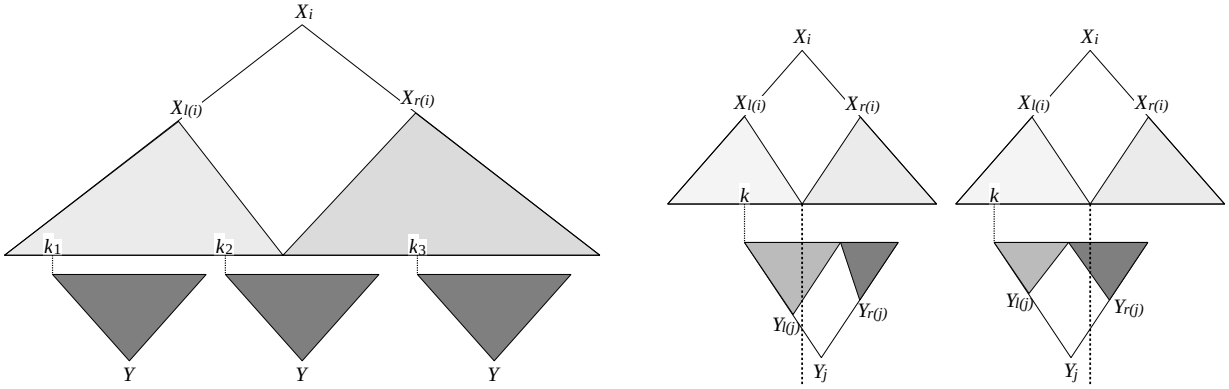
Lemma 1 *For any X in $\mathcal{T}$ and any Y in $\mathcal{P}$, $Occ^{\star}(X, Y)$ forms a single arithmetic progression.*

We have the following observation (see Figure 3 (a)):

Observation 1 (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 $\{Occ^{\star}(X_i, Y)\}_{i=1}^{n}$. By Lemma 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. 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 4 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 Figure 3 (b)):



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

(b) $k \in Occ^\star(X_i, Y_j)$ 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).

Figure 3: Decomposition of variables.

Observation 2 (decomposition of pattern variables)
For X_i in $\mathcal{T}$ and $Y_j = Y_{\ell(j)} \cdot Y_{r(j)}$ in $\mathcal{P}$,

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

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

Lemma 2 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 again. 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 6 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 1 *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 $O(n^2 m^2)$ 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 $\mathit{FirstMismatch}(T, P, k)$ 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 $1 \leq k \leq |T| - |P| + 1$. If there is no such i, the value of $\mathit{FirstMismatch}(T, P, k)$ is nil. The value is a witness of $k \notin \mathit{Occ}(T, P)$.

Lemma 2 (Key Lemma). *Let $T = u^p z$ $(u, z \in \Sigma^+, p \geq 0)$ and $P \in \Sigma^+$. The set $S = \mathit{Occ}(T, P) \cap \{1 + i|u| \ : \ i = 0, 1, ..., p\}$ forms a single arithmetic progression, which can be computed by at most three calls of FirstMismatch.*

Proof (sketch) We use the following notation in this proof: For two integers a and b, we denote by $\langle q, r\rangle = \mathit{div}(a,\ b)$ 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 $0 \leq r < b$.

Let $h = \max\{j \leq |T| - |P| + 1 \ : \ j = 1 + i|u| \text{ for } i = 0, 1, \ldots, p\}$. If no such j exist, $S = \phi$. The case $h \leq 1 + 2|u|$ is trivial, since S contains at most three positions. We consider the case $h \geq 1 + 3|u|$. At the beginning, we invoke the function *FirstMismatch* for the 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 $\mathit{miss}1$ and $\mathit{miss}2$, we have six cases as shown in Figure 4.

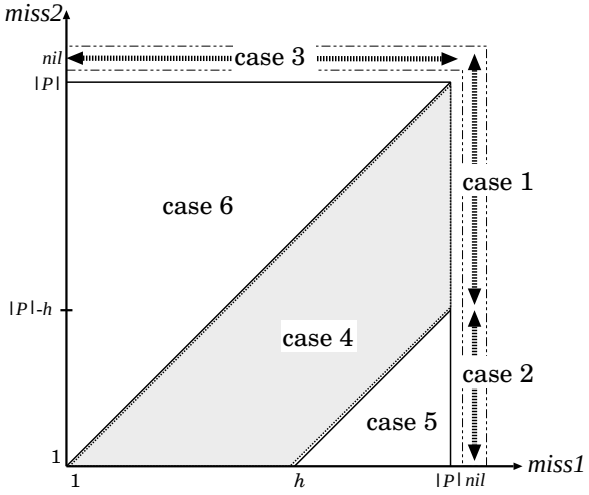


Figure 4: Six cases depending on $\mathit{miss}1$ and $\mathit{miss}2$.
(Cases 2 and 5 are vacant if $h > |P|$.)

case 1: $\mathit{miss}1 = \mathit{nil}$ and $\mathit{miss}2 = \mathit{nil}$ or $|P| - h + 1 < \mathit{miss}2$. See Figure 5.

Let $\langle q, r\rangle = \mathit{div}(|P|,\ |u|)$ and $\langle q', r'\rangle = \mathit{div}(h + \mathit{miss}2 - 2,\ |u|)$. We can show that $P = u^q u[1..r]$ and $T = u^{q'} u[1..r']w$ for some $w \in \Sigma^+$ with $u[r'+1] \neq w[1]$. We can show that $S = \{1 + i|u| \ : \ i \in \{0, .., t\}\}$, where $t = q' - q$ if $r' \geq r$ and $t = q' - q - 1$ otherwise. We note that such t can be directly computed by $\langle t, r''\rangle = \mathit{div}(h + \mathit{miss}2 - |P| - 2,\ |u|)$.

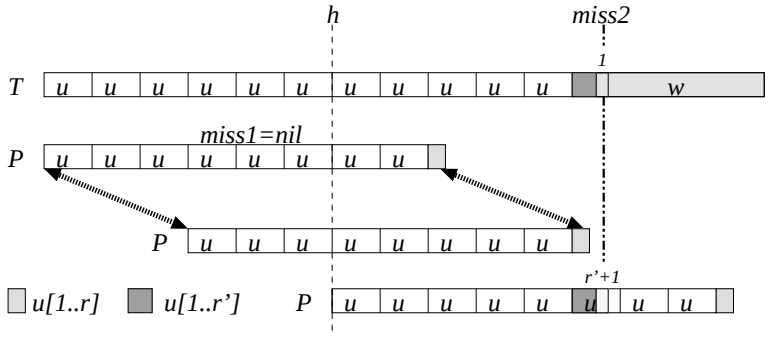


Figure 5: Case 1, $P = u^q u[1..r]$ and $T = u^{q'} u[1..r']w$.

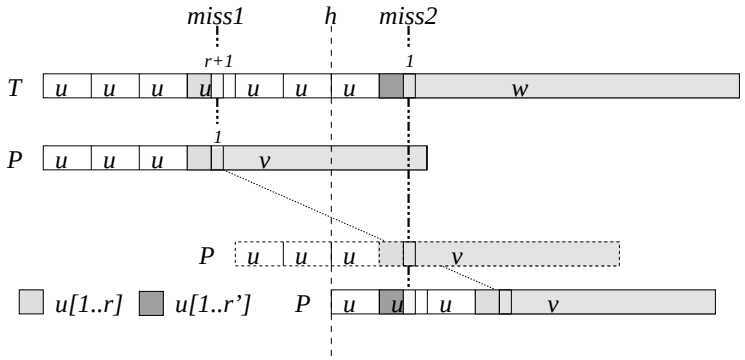


Figure 6: Case 4, $P = u^q u[1..r]v$ and $T = u^{q'} u[1..r']w$.

case 2: $miss1 = nil$ and $miss2 \leq |P| - h + 1$. (This is impossible if $h > |P|$).
Let $\langle q, r\rangle = div(h{+}miss2{-}2,\ |u|)$. 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: $miss1 \neq nil$ and $miss2 = nil$.
Let $\langle q, r\rangle = div(miss1{-}1,\ |u|)$, and $\langle q', r'\rangle = div(h{+}miss1{-}2,\ |u|)$. We can show 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: $miss1 - h + 1 < miss2 < miss1$. See Figure 6.
Let $\langle q, r\rangle = div(miss1{-}1,\ |u|)$ and $\langle q', r'\rangle = div(h{+}miss2{-}2,\ |u|)$. 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 $s = 1{+}p|u|{-}miss1{+}miss2$. 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 $FirstMismatch(T, P, s)$.

case 5: $miss2 \leq miss1 - h + 1$. (This is impossible if $h > |P|$).
Let $\langle q, r\rangle = div(h{+}miss2{-}2, |u|)$, and $s = miss1{-}h{-}miss2{+}2$. 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: $miss1 \leq miss2$.
Let $\langle q, r\rangle = div(miss1{-}1,\ |u|)$, and $\langle q', r'\rangle = div(h{+}miss1{-}2,\ |u|)$. Let $s = miss2 - miss1 + 1$. Since we can show 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 $v[s] \neq w[s]$, 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. □

The above lemma is the heart of our algorithm. It gives us an efficient way of computing the periodic pattern occurrences in the set $Occ^\star(X_i, Y_{\ell(j)})$, assuming that the function *FirstMismatch* can be realized efficiently. In the next section, we will give such a realization of $FirstMismatch(X, Y, k)$ for variables X in $\mathcal{T}$ and Y in $\mathcal{P}$.

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 $Occ^\star(X_i, Y_j)$. 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 $Occ^\star_\ell(X_i, Y_j) = Occ^\star(X_i, Y_{\ell(j)}) \cap (Occ(X_i, Y_{r(j)}) \ominus |Y_{\ell(j)}|)$.

Lemma 3 *Independently of the cardinality of the set $Occ^\star(X_i, Y_{\ell(j)})$, we can compute the set $Occ^\star_\ell(X_i, Y_j)$ by using the function $FirstMismatch(X_i, Y_{r(j)}, k)$ at most three times.*

Proof In case that the cardinality of the set $Occ^\star(X_i, Y_{\ell(j)})$ is at most two, we can compute the set $Occ^\star_\ell(X_i, Y_j)$ easily: For each $s \in Occ^\star(X_i, Y_{\ell(j)})$ we 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 $Occ^\star(X_i, Y_{\ell(j)})$ contains more than two positions, we apply Lemma 2 as follows. Let L and R be the minimum and the maximum elements in $Occ^\star(X_i, Y_{\ell(j)}) \oplus |Y_{\ell(j)}|$, respectively (Figure 7). Let d be the step of the arithmetic progression of $Occ^\star(X_i, Y_{\ell(j)})$, and let $p = (R - L)/d$. Then we can see that the string $X_i[L..|X_i|]$ is of the form $u^p z$, where u is the suffix of $Y_{\ell(j)}$ of length d. By Lemma 2, we can compute the set

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

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

$$\begin{array}{rcll} S \oplus (L-1) & = & (Occ(X_i[L..|X_i|], Y_{r(j)}) \oplus (L-1)) & \cap \quad \{L, L+d, \ldots, L+p\cdot d\} \\ & = & Occ(X_i, Y_{r(j)}) & \cap \quad (Occ^\star(X_i, Y_{\ell(j)}) \oplus |Y_{\ell(j)}|), \end{array}$$

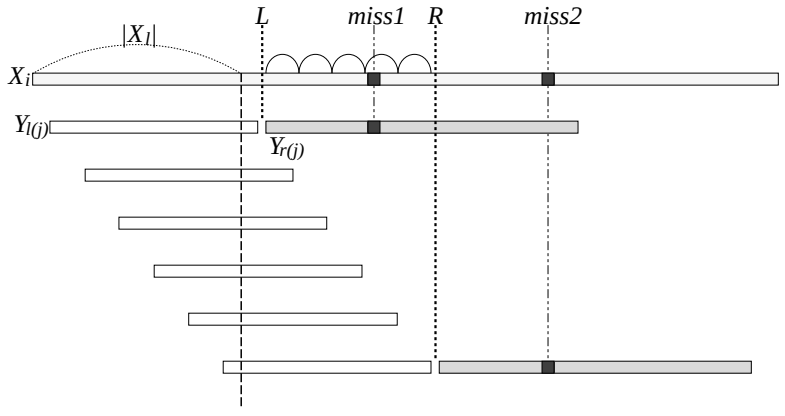


Figure 7: $FirstMismatch(X_i, Y_{r(j)}, L)$ and $FirstMismatch(X_i, Y_{r(j)}, R)$.

```
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
```

Figure 8: Pseudo-codes of the functions *FirstMismatch* and *Match*.

we have $S \oplus (L-1-|Y_{\ell(j)}|) = (Occ(X_i, Y_{r(j)}) \ominus |Y_{\ell(j)}|) \cap Occ^\star(X_i, Y_{\ell(j)}) = Occ^\star_\ell(X_i, Y_j)$, which is the desired set. □

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

Observation 3 *For two variables X in $\mathcal{T}$ and Y with $Y = Y_\ell \cdot Y_r$ in $\mathcal{P}$,*

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

Figure 8 shows a pseudo-code of the function *FirstMismatch*, 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 1.

Lemma 4 *The function $Match(X_i, Y_j, k)$ answers in $O(depth(X_i))$ time.*

Proof The membership query of the form $k \in Occ^\star(X_{i'}, Y_{j'})$ 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 *The function $FirstMismatch(X_i, Y_j, k)$ answers in $O(depth(X_i) \cdot depth(Y_j))$ time.*

Proof We can verify easily that the number of recursive calls of $FirstMismatch(X_i, Y_j, k)$ is at most $depth(Y_j)$. At each call, $FirstMismatch(X_i, Y_j, k)$ calls the function $Match(X_i, Y_j, k)$ once. By Lemma 4, it answers in $O(depth(Y_j))$ time. Thus the lemma holds. □

By Lemma 3 and Lemma 5, we have the following result.

Lemma 6 *We can compute each entry $App[i, j]$ in $O(depth(X_i) \cdot depth(Y_j))$ time.*

6 Conclusion

We have shown an improved pattern matching algorithm for strings in terms of straight-line programs. In Table 1, we summarize the results compared to the previous ones [10, 11], from the view points of time complexity, space complexity, and pattern detection ability when the pattern occurs in the text more than twice.

We briefly state the improvement of our algorithm compared to the that in [11]. The latter algorithm consists of two phases: At the first phase, it computes two sets: $Pref(X_i, Y_j)$ of the lengths of prefixes of Y_j that are suffixes of X_i, and $Suff(X_i, Y_j)$ of the lengths of suffixes of Y_j that are prefixes of X_i. At the second phases, it computes the set $Occ(X_i, Y_j)$ from $Pref(X_i, Y_j)$ and $Suff(X_i, Y_j)$ by solving certain linear Diophantine equations with using Euclid's algorithm. Each $Suff(X_i, Y_j)$ and $Pref(X_i, Y_j)$ can be stored in $O(depth(X_i) + depth(Y_j))$ space, although $Occ(X_i, Y_j)$ occupies only $O(1)$ space. On the other hand, our algorithm directly computes $Occ(X_i, Y_j)$. The property of periodic occurrences of a pattern in a text shown in the key lemma enabled the direct computation.

Both of these two algorithms use the information on the first position of mismatches. In our algorithm, pattern-to-text comparisons are enough, while the previous algorithm also requires text-to-text comparisons and pattern-to-pattern comparisons. This is the reason why the previous algorithm requires $(n+m) \times (n+m)$ table with $O(n+m)$ size entries in order to store the information on overlaps, while our algorithms requires only $n \times m$ table with $O(1)$ size entries. This is also the contribution of the key lemma.

Recently, Gąsieniec *et al.*[8] have developed a series of efficient algorithms, including pattern matching algorithm, for strings in terms of composition systems. A composition system is an extension of the straight-line program, where substring selection is also allowed as well as concatenation, in order to simulate the LZ77 coding scheme. We will adapt our algorithm to treat composition systems in future works, hopefully combined with the randomized approaches [9].

Table 1: Summary

algorithm	time	space	detection
[10]	$O((n+m)^7)$	not estimated	some one
[11]	$O((n+m)^4 \log (n+m))$	$O((n+m)^3)$	some one
Ours	$O(n^2 m^2)$	$O(nm)$	all

References

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

[2] A. Amir and G. Benson. Two-dimensional periodicity and its application. In *Proc. 3rd Symposium on Discrete Algorithms*, page 440, 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. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1996.

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

[6] 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.

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

[8] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. 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.

[9] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. 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.

[10] 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.

[11] M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing*, 1997. (to appear).