An Efficient Algorithm to Test Square-Freeness of Strings Compressed by Balanced Straight Line Program

Wataru Matsubara[1], Shunsuke Inenaga[2], and Ayumi Shinohara[1]

[1] Graduate School of Information Science, Tohoku University, Japan
{matsubara@shino., ayumi@}ecei.tohoku.ac.jp
[2] Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan
inenaga@c.csce.kyushu-u.ac.jp

Abstract. In this paper we study the problem of deciding whether a given *compressed string* contains a *square*. A string x is called a square if $x = zz$ and $z = u^k$ implies $k = 1$ and $u = z$. A string w is said to be *square-free* if no substrings of w are squares. Many efficient algorithms to test if a given string is square-free, have been developed so far. However, very little is known for testing square-freeness of a given *compressed string*. In this paper, we give an $O(\max(n^2, n \log^2 N))$-time $O(n^2)$-space solution to test square-freeness of a given compressed string, where n and N are the size of a given compressed string and the corresponding decompressed string, respectively. Our input strings are compressed by *balanced straight line program* (*BSLP*). We remark that BSLP has exponential compression, that is, $N = O(2^n)$. Hence no decompress-then-test approaches can be better than our method in the worst case.

1 Introduction

Analyzing repetitive structure of strings has a wide range of applications, including bioinformatics [1,2], formal language theory [3] and combinatorics on words [4]. The most basic repetitive structure is zz, where z is a non-empty string. Such a string zz is called a *repetition*. In particular, when z is *primitive* ($z = u^k$ implies $k = 1$ and $u = z$), repetition zz is said to be a *square*.

A string w is said to be *square-free* or *repetition-free* if w contains no squares. It is easy to see that any string of length greater than three over a binary alphabet contains a square. However, there exists a square-free string over an alphabet of size greater than two. For instance, `abcacbabcb` is a square-free string over alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. It was shown in [5,6] that there exist square-free strings of infinite length over a ternary alphabet.

Since then, there have been extensive studies on testing square-freeness of a string as well as finding squares in a string. Main and Lorentz [7] presented

Dagstuhl Seminar Proceedings 08261
Structure-Based Compression of Complex Massive Data
http://drops.dagstuhl.de/opus/volltexte/2008/1680

an $O(N)$-time algorithm to test if a given string of length N is square-free. Crochemore [8] also proposed an $O(N)$-time algorithm for the same problem.

On the other hand, it is known that the maximum number of squares in a string of length N is $\Theta(N \log N)$ [9,10]. Optimal $O(N \log N)$ algorithms that detect all occurrences of squares of a given string have been proposed [9,8,11]. Kolpakov and Kucherov [12] showed that any string of length N can contain $O(N)$ *distinct* squares, and developed an $O(N)$-time algorithm to find all distinct squares from a given string.

There are also several efficient parallel algorithms for the above problems. Crochemore and Rytter [13] discovered a parallel algorithm to test square-freeness of a string, which runs in $O(\log N)$ time using N processors. Apostolico [14] showed an algorithm that can find all occurrences of squares with the same time bound and the number of processors. Then, Apostolico and Breslauer [15] presented parallel algorithms working in $O(\log \log N)$ time using $N \log N / \log \log N$ processors, which test square-freeness and find all occurrences of squares of a given string.

However, very little is known to the case where the input strings are given in *compressed forms*. To our knowledge, the only relevant result is an $O(n^6 \log^5 N)$ solution to find all occurrences of squares [16]. Their input is a string compressed by *composition systems*, and n in the above complexity is the size of the compressed input string. The matter about their solution is that no details of the algorithm have ever been appeared.

In this paper, we present an $O(\max(n^2, n \log^2 N))$-time $O(n^2)$-space algorithm to test square-freeness of a given compressed string. Our input string is compressed by *balanced straight line program* (*BSLP*) proposed by [17]. BSLP is a variant of *straight line program* (*SLP*) [18]. SLP is a kind of context-free grammar in the Chomsky normal form whose production rules are in either of the form $X \to YZ$ or $X \to \mathtt{a}$, and SLP derives only one string. We remark that BSLP has *exponential compression*, which means that $N = O(2^n)$. Therefore, no algorithms that decompress a given BSLP-compressed string can be better than our algorithm in the worst case.

2 Preliminaries

2.1 Notations

For any set S of integers and an integer k, let

$$S \oplus k = \{i + k \mid i \in S\} \text{ and}$$
$$S \ominus k = \{i - k \mid i \in S\}.$$

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively.

The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. Let $w^{[d]}$ denote the prefix of length $|w| - d$ of w, that is, $w^{[d]} = w[1 : |w| - d]$. For any string w, let w^R denote the reversed string of w, namely, $w^R = w[|w|] \cdots w[2]w[1]$.

For any strings w, x, and integer k, we define the set $Occ^{\triangle}(w, x, k)$ of all occurrences of x that cover or touch the position k of w, namely,

$$Occ^{\triangle}(w, x, k) = \left\{ s \,\middle|\, \begin{matrix} w[s : s + |x| - 1] = x, \\ k - |x| \leq s \leq k + 1 \end{matrix} \right\}.$$

We will heavily use the following lemma.

Lemma 1 ([19]). *For any strings w, x, and integer k, $Occ^{\triangle}(w, x, k)$ forms a single arithmetic progression.*

In what follows, we assume that $Occ^{\triangle}(w, x, k)$ is represented by a triple of the first element, the common difference, and the number of elements of the progression, which takes $O(1)$ space.

A non-empty string of the form xx is called a *repetition*. A string w is said to be *repetition-free* if no substrings of w are repetitions.

A string of x is said to be *primitive* if $x = u^k$ for some integer k implies that $k = 1$ and $x = u$. A repetition xx is called a *square* if x is primitive. A string w is said to be *square-free* if no substrings of w are squares. By definition, any string w is square-free if and only if w is repetition-free.

A repetition xx, which is a substring of a string w starting at position i, is said to be *centered* at position $i + |x| - 1$.

2.2 Straight Line Program

Definition 1. *A* straight line program $\mathcal{T}$ *is a sequence of assignments such that*

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

where each X_i is a variable and each $expr_i$ is an expression in either of the following form:

- $expr_i = a \quad (a \in \Sigma)$ *or*
- $expr_i = X_\ell X_r \quad (\ell, r < i)$.

Since the straight line program (SLP) $\mathcal{T}$ has no recursive structure, it describes exactly one string. That is, an SLP can be seen as a CFG in the Chomsky normal form which generates exactly one string. Denote by T the string derived from the last variable X_n of the program $\mathcal{T}$.

The *size* of the program $\mathcal{T}$ is the number n of assignments in $\mathcal{T}$.

We define the *height* of a variable X_i by

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

For any variable X_i of $\mathcal{T}$, we define X_i^R as follows:

$$X_i^R = \begin{cases} a & \text{if } X_i = a \quad (a \in \Sigma), \\ X_r^R X_\ell^R & \text{if } X_i = X_\ell X_r \quad (\ell, r < i). \end{cases}$$

Let $\mathcal{T}^R$ be the SLP consisting of variables X_i^R for $1 \leq i \leq n$.

Lemma 2 ([20]). *For any SLP $\mathcal{T}$ which derives string T, SLP $\mathcal{T}^R$ derives string T^R and can be computed in $O(n)$ time from SLP $\mathcal{T}$.*

When it is not confusing, we identify a variable X_i with the string derived from X_i. Then, $|X_i|$ denotes the length of the string derived from X_i.

3 Balanced Straight Line Program

We define a *balanced straight line program* (*BSLP*), which is a variant of an SLP of Definition 1.

Definition 2. *A* balanced straight line program $\mathcal{T}$ *is a sequence of assignments such that*

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

where each X_i is a variable and each $expr_i$ is an expression in either of the following form:

- $expr_i = a \quad (i < n,\ a \in \Sigma)$ *or*
- $expr_i = X_\ell X_r$ *with* $|X_\ell| = |X_r| \quad (\ell, r \leq i < n)$, *and*
- $expr_n = X_\ell^{[d]} X_r$ *with* $X_\ell[|X_\ell| - d + 1 : |X_\ell|] = X_r[1 : d] \quad (\ell, r < n,\ d \geq 0)$.

Note that the derivation tree of any BSLP variable of the form $X_i = X_\ell X_r$ is a complete binary tree. BSLP is a compression scheme which has exponential compression, that is, $O(N) = O(2^n)$, where N is the length of the decompressed string.

Example 1. Consider BSLP $\mathcal{T} = \{X_i\}_{i=1}^{10}$ with $X_1 = \mathtt{a}$, $X_2 = \mathtt{b}$, $X_3 = X_1X_2$, $X_4 = X_1X_1$, $X_5 = X_3X_3$, $X_6 = X_3X_4$, $X_7 = X_4X_3$, $X_8 = X_5X_6$, $X_9 = X_7X_6$, and $X_{10} = X_8^{[2]}X_9$ that generates string $T = \mathtt{abababaaababaa}$. The derivation tree of BSLP $\mathcal{T}$ is shown in Figure 1.

4 Apostolico and Breslauer's Algorithm

In this section we recall a parallel algorithm of [15] that checks square-freeness of a string, on which our algorithm will be based.

Firstly, we define the *FM* function, as follows. Given strings X, Y, and integer k, $FM(X, Y, k)$ returns the first position of mismatch when we compare X with Y at position k, that is:

$$FM(X, Y, k) = \\ \min\{1 \leq i \leq |Y| \mid X[k + i - 1] \neq Y[i]\}. \tag{1}$$

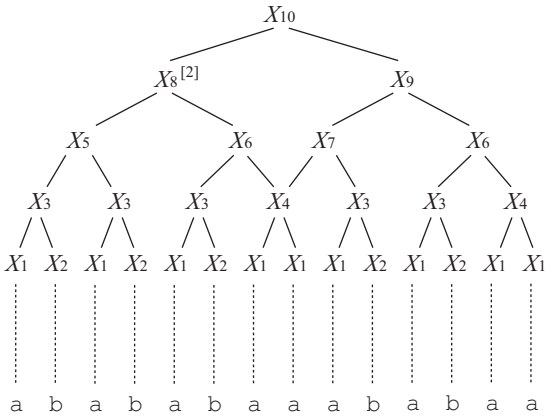

Fig. 1. The derivation tree of BSLP $\mathcal{T}$ of Example 1 that generates string $T =$ `abababaaababaa`. Recall that $X_8^{[2]}$ denotes the suffix of X_8 of length $|X_8| - 2$.

If there exists no such i, let $FM(X, Y, k) = 0$. In other words, $FM(X, Y, k)$ is the length of the common prefix of $X[k : |X|]$ and Y plus one.

Let w be any string of length N, where N is a power of 2. For each $0 \leq t \leq \log_2 N - 1$, partition w into consecutive blocks of length $m = 2^t$. Let $B = w[k : k + m - 1]$ be one of such blocks. A repetition zz, which is a substring of w, is said to be *hinged* on B if repetition zz satisfies the following conditions:

- $2m - 1 \leq |z| < 4m - 1$ and
- the first z of the repetition fully contains B, that is, $zz = w[h : h + 2|z| - 1]$ and $k - |z| + m \leq h \leq k$.

Let P_1 and P_2 be the set of positions where a copy of B occurs in $w[k+2m : k+4m-1]$ and $w[k+3m : k+5m-1]$, respectively. Let $p \in P_1 \cup P_2$, and let

$$\alpha = FM(w, w[k+m:p-1], p+m),$$
$$\gamma = FM(w^R, w^R[N-k+1:N-k+m], N-p+1).$$

Repetitions hinged on B can be detected based on the following lemma.

Lemma 3 ([15]). *There exist repetitions zz which are hinged on B with $|z| = p - k$, if and only if $p - \gamma \leq k + m + \alpha - 1$.*

The above lemma is useful when the size of $P_1 \cup P_2$ is at most two. In other cases, the next lemma is helpful:

Lemma 4 ([15]). *If $|P_1 \cup P_2| > 2$, then w is not repetition-free.*

A function to test if there is a square in string w hinged on a block $B = w[k : k + m - 1]$ is shown in Algorithm 1.

The algorithm of [15] consists of $\log_2 N$ stages, and in the stage number t ($0 \leq t \leq \log_2 N - 1$) it looks for repetitions hinged on any block of length

Algorithm 1: Function *HingedSq*($\mathsf{w}, \mathsf{k}, \mathsf{m}$) to test if there is a square in w hinged on $\mathsf{w}[\mathsf{k} : \mathsf{k}+\mathsf{m}+1]$.

Input: String w of length N and integers k, m.
Output: Whether there exists a square in w hinged on $\mathsf{w}[\mathsf{k} : \mathsf{k}+\mathsf{m}+1]$ or not.

```
1 B = w[k : k + m + 1];
2 P_1 = the set of occurrence positions of B in w[k + 2m : k + 4m − 1]);
3 P_2 = the set of occurrence positions of B in w[k + 3m : k + 5m − 1]);
4 if |P_1 ∪ P_2| > 2 then return true;
5 foreach p ∈ P_1 ∪ P_2 do
6 |   α = FM(w, w[k+m : p−1], p+m);
7 |   γ = FM(w^R, w^R[N−k+1 : N−k+m], N−p+1);
8 |   if p − γ ≤ k + m + α − 1 then return true;
9 return false;
```

$2^t = m$, based on Lemma 3 and Lemma 4. Their algorithm tests if a given string is square-free or not in $O(\log \log N)$ time using $N \log N / \log \log N$ processors.

5 Testing Square-freeness of BSLP-Compressed Strings

In this section, we present our algorithms to test square-freeness of a given BSLP-compressed strings.

5.1 Testing Square-freeness of Variables Forming Complete Binary Trees

We begin with testing whether or not a string described by a variable forming a complete binary tree contains a square.

Problem 1. Given a variable $X_i = X_\ell X_r$ with $|X_\ell| = |X_r|$, determine whether the string derived by X_i is square-free (or equivalently, repetition-free).

Observation 1 *For any variable $X_i = X_\ell X_r$ and a repetition zz which is a substring of X_i, there always exists a descendant Y of X_i such that*

- *zz is a substring of Y and*
- *zz touches or covers the boundary of Y.*

See Figure 2 that illustrates the above observation.

Due to Observation 1, Problem 1 is reduced to the following sub-problem.

Problem 2. Given a variable $X_i = X_\ell X_r$ with $|X_\ell| = |X_r|$, determine whether or not there is a repetition that touches or covers the boundary of X_i.

In the sequel, we present our algorithm to solve Problem 2. The algorithm is based on the parallel algorithm of [15] summarized in Section 4.

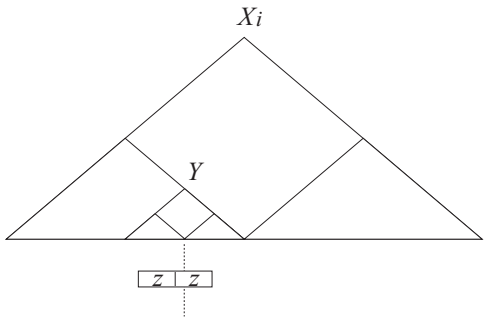

Fig. 2. Illustration of Observation 1. Repetition zz is a substring of Y and covers the boundary of Y.

Lemma 5. *Any repetition, which touches or covers the boundary of variable X_i, is hinged on some descendant of X_i. Moreover, there are at most 14 such descendants of height h for each $1 \leq h \leq height(X_i) - 2$. (See also Figure 3.)*

Proof. Consider repetitions zz that are centered within X_ℓ. Recall that repetition zz are hinged on a substring of length 2^{h-1} only if $2 \times 2^{h-1} - 1 = 2^h - 1 \leq |z| < 4 \times 2^{h-1} - 1 = 2^{h+1} - 1$. Hence repetition zz is of length at least $2^{h+1} - 2$ and at most $2^{h+2} - 4$. Therefore, for repetition zz to touch or cover the boundary of X_i and be centered in X_ℓ, the first z of the repetition has to occur in $X_i[|X_\ell| - 2^{h+2} + 5 : |X_\ell|] = X_\ell[|X_\ell| - 2^{h+2} + 5 : |X_\ell|]$, which is the suffix of X_ℓ of length $2^{h+2} - 4$. It is clear that the suffix contains 7 variables of length 2^{h-1}, each being of height h. The other case for repetitions centered within X_r is symmetric. That is, for the beginning position s of each such variable Y in X_i we have

$$|X_i| - |X_\ell| - 7|Y| \leq s \leq |X_\ell| + 6|Y|.$$

□

For any variables $X_i = X_\ell X_r$ and X_j, we abbreviate as

$$Occ^{\triangle}(X_i, X_j, |X_\ell|) = Occ^{\triangle}(X_i, X_j).$$

That is, $Occ^{\triangle}(X_i, X_j)$ is the set of occurrences of X_j that touch or cover the boundary of X_i.

The following theorem is critical to our algorithm, which shows the complexity of computing $Occ^{\triangle}(X_i, X_j)$ for variables X_i and X_j both forming complete binary trees.

Theorem 1 ([17]). *For every pair X_i and X_j of variables both forming complete binary trees, $Occ^{\triangle}(X_i, X_j)$ can be computed in total of $O(n^2)$ time and space.*

We are ready to state the next lemma.

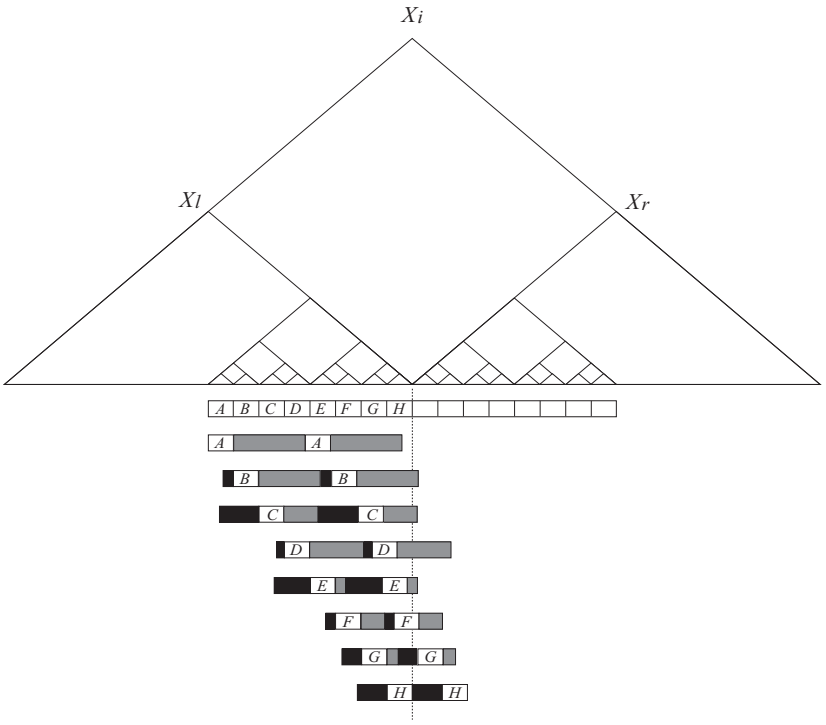

Fig. 3. Illustration of Lemma 5 for height $h = \mathit{height}(|X_i|) - 5$. No repetitions zz hinged on variable A can touch or cover the boundary of X_i, since repetitions zz are hinged on variable A only if $2 \times |A| - 1 = 2^h - 1 \leq |z| < 2^{h+1} - 1 = 4 \times |A| - 1$.

Lemma 6. *Problem 2 can be solved in $O(\log^2 |X_i|)$ time with $O(n^2)$ preprocessing.*

Proof. We process a given variable X_i in $\mathit{height}(X_i) - 2$ stages, where each stage is associated with height h, such that $1 \leq h \leq \mathit{height}(X_i) - 2$. In each stage with height h, there are at most 14 descendants to consider by Lemma 5. Let Y be one of such descendants, and let s be the beginning position of Y in X_i, that is, $Y = X_i[s : s + |Y| - 1]$. Also, let V and Z be a variable whose boundary is at position $s + 3|Y| - 1$ and at position $s + 4|Y| - 1$, respectively. It is easy to see that $|V| \geq 2|Y|$ and $|Z| \geq 2|Y|$. It follows from Lemma 1 that $\mathit{Occ}^{\triangle}(V, Y)$ and $\mathit{Occ}^{\triangle}(Z, Y)$ form a single arithmetic progression. Due to Lemma 4,

1. If $|\mathit{Occ}^{\triangle}(V, Y) \cup \mathit{Occ}^{\triangle}(Z, Y)| > 2$, then X_i is not repetition-free (see the left of Figure 4).
2. If $|\mathit{Occ}^{\triangle}(V, Y) \cup \mathit{Occ}^{\triangle}(Z, Y)| \leq 2$, then we compute the values of α and γ according to Lemma 3 and test if the conditions in the lemma is satisfied or not (see the right of Figure 4).

Let us analyze the time complexity. Computing $Occ^{\triangle}(\cdot,\cdot)$ for each pair of variables takes $O(n^2)$ time by Theorem 1. The variables V and Z can be found in $O(height(X_i))$ time by a binary search. Since $p-s \leq 3|Y|$ and $X_i[s+|Y| : s+4|Y|-1]$ can be represented by at most two BSLP variables, $\alpha = FM(X_i, X_i[s+|Y| : p-1], p+|Y|)$ can be computed by at most two calls of the FM function. The value of γ can be computed similarly by at most two calls of the FM function, provided that $\{X_i^R \mid 1 \leq i < n\}$ and $Occ^{\triangle}(X_i^R, X_j^R)$ for every $1 \leq i, j < n$ are already computed. By Lemma 2, these reversed variables can be precomputed in $O(n)$ time. As to be shown in Section 6, the FM function can be answered in $O(\log |X_i|)$ time. There are $height(X_i)-2$ stages. Since $height(X_i) = \log_2 |X_i|+1$, the total time complexity is $O(\log^2 |X_i|)$. □

Our algorithm to solve Problem 2 is shown in Algorithms 2 and 3.

Algorithm 2: Function $HingedSqBSLP(\mathsf{X}, \mathsf{Y})$ to test if there exists a square in X hinged on Y.

Input: BSLP variables X and Y.
Output: Whether there exists a square in X which is hinged on Y.

```
1  V = a variable whose boundary is at position s + 3|Y| − 1 in X;
2  Z = a variable whose boundary is at position s + 4|Y| − 1 in X;
3  if |Occ^△(V, Y) ∪ Occ^△(Z, Y)| > 2 then
4  |   return true;
5  foreach p ∈ Occ^△(V, Y) ∪ Occ^△(Z, Y) do
6  |   compute α by at most two calls of FM;
7  |   compute γ by at most two calls of FM;
8  |   if p − γ ≤ s + |Y| + α − 1 then
9  |   |   return true;
10 |
11 return false;
```

5.2 Testing Square-freeness of BSLP-compressed Strings

Here, we consider the next problem, which is the main problem of this paper.

Problem 3 (Square-freeness Test for BSLP). Given BSLP $\mathcal{T}$ that describes string T, determine whether T is square-free.

The two following lemmas are useful for establishing Theorem 2, which is the main result of this subsection.

Lemma 7 ([21]). *For any variables X_i and X_j forming complete binary trees and integer k, $Occ^{\triangle}(X_i, X_j, k)$ can be computed in $O(\log |X_i|)$ time, provided that $Occ^{\triangle}(X_{i'}, X_{j'})$ is already computed for every $1 \leq i' \leq i$ and $1 \leq j' \leq j$.*

Algorithm 3: Function *TestSqBSLPVar*($\mathsf{X_i}$) to test square-freeness of a BSLP variable $\mathsf{X_i}$.

Input: BSLP variable $\mathsf{X_i}$ with $\mathsf{i} \neq \mathsf{n}$.
Output: Whether there exists a square in $\mathsf{X_i}$.

```
1 for h = 1 to height(X) do
2     foreach descendant Y of X such that height(Y) = h, Y = X[s : s + |X| − 1],
      and |X|/2 − 7|Y| ≤ s ≤ |X|/2 + 6|Y| do
3         if HingedSqBSLP(X, Y) = true then
4             return true;
5
6
7 return false;
```

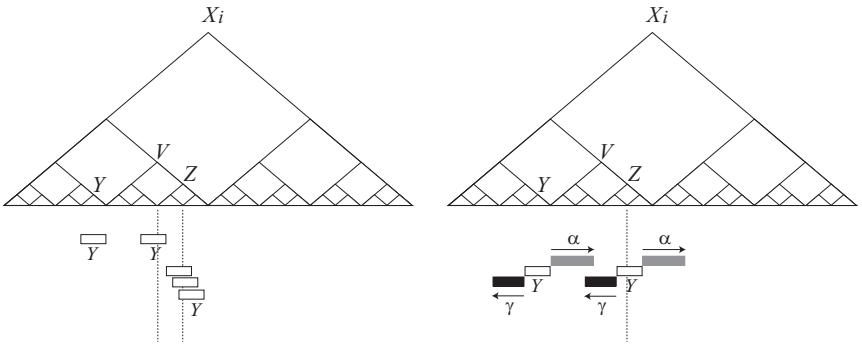

Fig. 4. Illustration of Lemma 6. The left is Case 1, and the right is Case 2.

For any variable $X_i = X_\ell X_r$ with $|X_\ell| = |X_r|$, we recursively define the *leftmost descendant* $lmd(X_i, h)$ and the *rightmost descendant* $rmd(X_i, h)$ of X_i with respect to height h ($\leq height(X_i)$), as follows:

$$lmd(X_i, h) = \begin{cases} lmd(X_\ell, h) & \text{if } height(X_i) > h, \\ X_i & \text{if } height(X_i) = h, \end{cases}$$

$$rmd(X_i, h) = \begin{cases} rmd(X_r, h) & \text{if } height(X_i) > h, \\ X_i & \text{if } height(X_i) = h. \end{cases}$$

For each variable X_i ($1 \leq i < n$) and height h ($< height(X_i)$), we precompute two tables of size $O(n \log N)$ storing $lmd(X_i, h)$ and $rmd(X_i, h)$ respectively. By looking up these tables, we can refer to any $lmd(X_i, h)$ and $rmd(X_i, h)$ in constant time. These tables can be constructed in $O(n \log N)$ time in a bottom-up manner [17].

Lemma 8. *For the last variable $X_n = X_\ell^{[d]} X_r$ and any variable $X_j = X_s X_t$ with $|X_s| = |X_t|$, $Occ^{\triangle}(X_n, X_j, |X_\ell|)$ can be computed in $O(\log^2 N)$ time, provided that $Occ^{\triangle}(X_{i'}, X_{j'})$ is already computed for every $1 \leq i' \leq n$ and $1 \leq j' \leq n$.*

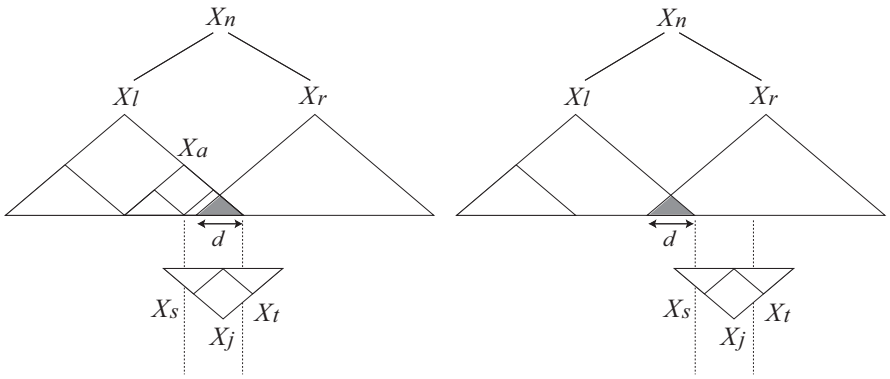

Fig. 5. Illustration of Lemma 8. If $|X_j| > d$, $Occ^{\triangle}(X_n, X_j, |X_\ell|)$ is equal to the union of $Occ^{\triangle}(X_a, X_s) \cap Occ^{\triangle}(X_n, X_t, |X_\ell|) \ominus |X_s|$ (the left) and $Occ^{\triangle}(X_n, X_s, |X_\ell|) \cap Occ^{\triangle}(X_r, X_t, |X_s| + d) \ominus |X_\ell|$ (the right).

Proof. Let $X_a = rmd(X_\ell, height(X_j))$. We can compute $Occ^{\triangle}(X_n, X_j, |X_\ell|)$ using the following recursion (see also Figure 5).

$$Occ^{\triangle}(X_n, X_j, |X_\ell|) = \begin{cases} Occ^{\triangle}(X_r, X_j, d) \ominus |X_\ell| \oplus d & \text{if } |X_j| \leq d, \\ (Occ^{\triangle}(X_a, X_s) \cap Occ^{\triangle}(X_n, X_t, |X_\ell|) \ominus |X_s|) \cup & \\ (Occ^{\triangle}(X_n, X_s, |X_\ell|) \cap Occ^{\triangle}(X_r, X_t, |X_s| + d) \ominus |X_\ell|) & \\ \qquad \text{if } |X_j| > d. & \end{cases}$$

It can be shown in a similar way to Lemma 5 of [17] that the intersection operations can be performed in $O(1)$ time and each of the resulting sets contains at most one element. This also implies that the union operation between the two resulting sets can be performed in $O(1)$ time. It follows from Lemma 7 that each of $Occ^{\triangle}(X_r, X_j, d)$, $Occ^{\triangle}(X_n, X_t, |X_\ell|)$, $Occ^{\triangle}(X_n, X_s, |X_\ell|)$, and $Occ^{\triangle}(X_r, X_t, |X_s| + d)$ can be computed in $O(\log N)$ time. Since the depth of the recursion is at most $height(X_j)$, the overall complexity is $O(\log^2 N)$. □

Theorem 2. *Problem 3 can be solved in $O(\max(n^2, n \log^2 N)$ time using $O(n^2)$ space.*

Proof. By Lemma 6, square-freeness of the $n - 1$ variables forming complete binary trees can be tested in total of $O(\max(n^2, n \log^2 N))$ time using $O(n^2)$ space.

What remains to show is how to test square-freeness of the last variable $X_n = X_\ell^{[d]} X_r$. We for now assume that no repetitions that touch or cover the boundary of X_i are found for every $1 \leq i < n$, since otherwise there is no way

for the last variable X_n to be repetition-free. Note that there is no repetition zz of length not greater than d in X_n, since such a repetition must touch or cover the boundary of some descendant of X_n, but this contradicts the above assumption. Hence all we need is to test whether there exists a repetition zz such that $zz = X_n[s : s + 2|z| - 1]$ with some $|X_\ell| - 2|z| + 1 < s \leq |X_\ell| - d$.

The testing algorithm is a modification of that of Lemma 6. We process X_n with at most $\max(\mathit{height}(X_\ell), \mathit{height}(X_r)) - 2$ stages. Let us focus on some variable Y of height $h < \max(\mathit{height}(X_\ell), \mathit{height}(X_r)) - 2$ on which a repetition satisfying the above condition may be hinged. See also Figure 6. We can compute $Occ^{\triangle}(X_n, Y, |X_\ell|)$ in $O(\log^2 N)$ time by Lemma 8 (the left arithmetic progression in Figure 6). By Lemma 7, $Occ^{\triangle}(X_r, Y, |Y| + d)$ can be computed in $O(\log N)$ time (the right arithmetic progression in Figure 6). As to be shown by Lemma 10 and Lemma 11 in Section 6, the FM function can be computed in $O(\log^2 N)$ time when testing square-freeness of the last variable X_n. Hence the values of α and γ can also be computed in $O(\log^2 N)$ time. Since $\max(\mathit{height}(X_\ell), \mathit{height}(X_r)) < \log_2 N + 1$, we can test square-freeness of the last variable X_n in $O(\log^3 N)$ time. Therefore, the overall time cost stays $O(\max(n^2, n \log^2 N))$. The space requirement remains $O(n^2)$ as we only used the precomputed values of $Occ^{\triangle}(X_i, X_j)$ and $Occ^{\triangle}(X_i^R, X_j^R)$ for each $1 \leq i < n$ and $1 \leq j < n$. □

Our algorithm to solve Problem 2 is shown in Algorithms 4 and 5.

Algorithm 4: Function $\mathit{HingedSqBSLPLast}(\mathsf{X_n}, \mathsf{Y})$ to test if there exists a square in $\mathsf{X_n}$ hinged on Y.

Input: BSLP variables X and Y.
Output: Whether there exists a square in last variable $\mathsf{X_n}$ which is hinged on Y.

```
1  compute Occ^△(X_n, Y, |X_ℓ|);
2  compute Occ^△(X_r, Y, |Y| + d);
3  if |Occ^△(X_n, Y, |X_ℓ|) ∪ Occ^△(X_r, Y, |Y| + d)| > 2 then
4  |   return true;
5  foreach p ∈ Occ^△(X_n, Y, |X_ℓ|) ∪ Occ^△(X_r, Y, |Y| + d) do
6  |   compute α by at most two calls of FM;
7  |   compute γ by at most two calls of FM;
8  |   if p − γ ≤ s + |Y| + α − 1 then
9  |   |   return true;
10 |
11 return false;
```

6 Computing the *FM* Function

The FM function in equation (1) plays a central role in our algorithms to compute squares from BSLP-compressed strings. In our problem setting, the first

Algorithm 5: Algorithm *TestSqBSLP*($\mathcal{T}$) to test square-freeness of string T given as BSLP $\mathcal{T}$.

Input: BSLP $\mathcal{T} = \{\mathsf{X_i}\}_{\mathsf{i}=1}^{\mathsf{n}}$ describing string T.
Output: Whether there exists a square in T.
/* Assume $\mathsf{X_n} = \mathsf{X}_\ell^{[\mathsf{d}]}\mathsf{X_r}$. */
for $\mathsf{i} = 1$ **to** $\mathsf{n} - 1$ **do**
  for $\mathsf{j} = 1$ **to** $\mathsf{n} - 1$ **do**
    compute $Occ^{\triangle}(\mathsf{X_i}, \mathsf{X_j})$;
    compute $Occ^{\triangle}(\mathsf{X_i}^R, \mathsf{X_j}^R)$;
if *TestSqBSLPVar*(X_ℓ) = true **then**
  return true;
if *TestSqBSLPVar*($\mathsf{X_r}$) = true **then**
  return true;
for $\mathsf{h} = 1$ **to** $\max(height(\mathsf{X}_\ell), height(\mathsf{X_r})) - 2$ **do**
  foreach *descendant* Y *of* $\mathsf{X_n}$ *such that* $height(\mathsf{Y}) = \mathsf{h}$, $\mathsf{Y} = \mathsf{X_n}[\mathsf{s} : \mathsf{s} + |\mathsf{X}| - 1]$, *and* $|\mathsf{X_n}|/2 - 7|\mathsf{Y}| \leq \mathsf{s} \leq |\mathsf{X_n}|/2 + 6|\mathsf{Y}|$ **do**
    /* Lemma 5 */
    if *HingedSqBSLPLast*($\mathsf{X_n}, \mathsf{Y}$) = true **then**
      return true;
return false;

two inputs of the function are compressed forms. Given general SLP variables X and Y, $FM(X, Y, k)$ can be answered in $O(n^2)$ time with $O(n^3)$-time preprocessing [19,22]. In this section, we show that if X and Y form complete binary trees, then $FM(X, Y, k)$ can be answered in $O(\log |X|)$ time with $O(n^2)$-time preprocessing.

Lemma 9. *For any variables $X_i = X_\ell X_r$, $X_j = X_s X_t$ with $1 \leq i, j < n$ and integer k, $FM(X_i, X_j, k)$ can be computed in $O(\log |X_i|)$ time, provided that $Occ^{\triangle}(X_{i'}, X_{j'})$ is already computed for every $1 \leq i' \leq i$ and $1 \leq j' \leq j$.*

Proof. We can recursively compute $FM(X_i, X_j, k)$, as follows (see also Figure 7):

1. If $k + |X_j| \leq |X_\ell|$, then
   $FM(X_i, X_j, k) = FM(X_\ell, X_j, k)$.
2. If $k > |X_\ell|$, then
   $FM(X_i, X_j, k) = FM(X_r, X_j, k)$.
3. If $k + |X_s| \leq |X_\ell| < k + |X_j|$, then we have the two following sub-cases. Let $X_a = rmd(X_\ell, height(X_j))$.
   (a) If $k - |X_\ell| + |X_a| \notin Occ^{\triangle}(X_a, X_s)$, then
   $FM(X_i, X_j, k) = FM(X_a, X_s, k - |X_\ell| + |X_a|)$.
   (b) If $k - |X_\ell| + |X_a| \in Occ^{\triangle}(X_a, X_s)$, then
   $FM(X_i, X_j, k) = FM(X_i, X_t, k + |X_s|) + |X_s|$.

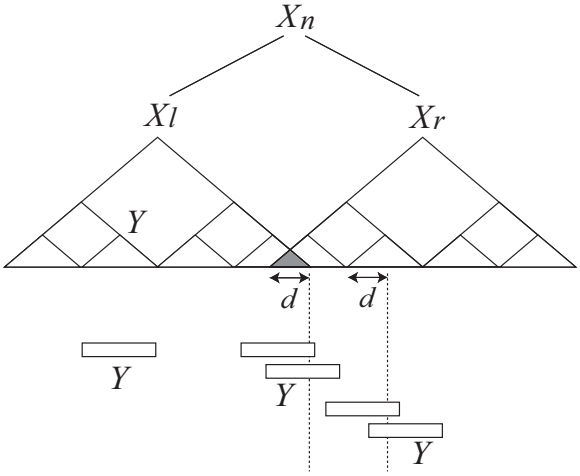

Fig. 6. Illustration of Theorem 2.

4. If $k < |X_\ell| < k + |X_s|$, then we have the two following sub-cases.
   (a) If $k \notin Occ^{\triangle}(X_i, X_s)$, then
   $FM(X_i, X_j, k) = FM(X_i, X_s, k)$.
   (b) If $k \in Occ^{\triangle}(X_i, X_s)$, then
   $FM(X_i, X_j, k) = FM(X_r, X_t, k + |X_s| - |X_\ell|) + |X_s|$.

In each recursion, either or both of the first and second variables in the FM function decrease the height by at least one. Since $|X_j| \leq |X_i|$ and $height(X_i) = \log_2 |X_i| + 1$, we conclude that $FM(X_i, X_j, k)$ can be computed in $O(\log |X_i|)$ time. □

What remains is how to efficiently compute the FM function for the last variable of BSLPs.

Lemma 10. *For any BSLP variables $X_n = X_\ell^{[d]} X_r$, $X_j = X_s X_t$ and integer k, $FM(X_n, X_j, k)$ can be computed in $O(\log N)$ time, provided that $Occ^{\triangle}(X_{i'}, X_{j'})$ is already computed for every $1 \leq i' < n$ and $1 \leq j' \leq j$.*

Proof. We can recursively compute $FM(X_n, X_j, k)$, as follows (see also Figure 8):

1. If $k \leq |X_\ell| - |X_j| + 1$, then
   $FM(X_n, X_j, k) = FM(X_\ell, X_j, k)$.
2. If $k \geq |X_\ell| - d + 1$, then
   $FM(X_n, X_j, k) = FM(X_r, X_j, k)$.
3. If $|X_\ell| - |X_j| + 1 < k < |X_\ell| - d + 1$, then we have the following sub-cases.
   Let $X_a = rmd(X_\ell, height(X_j))$.
   (a) If $k + |X_s| - 1 \leq |X_\ell| - |X_a|$, then
      i. If $k - |X_\ell| + |X_a| \notin Occ^{\triangle}(X_a, X_s)$, then
      $FM(X_n, X_j, k) = FM(X_a, X_s, k - |X_\ell| + |X_a|)$.

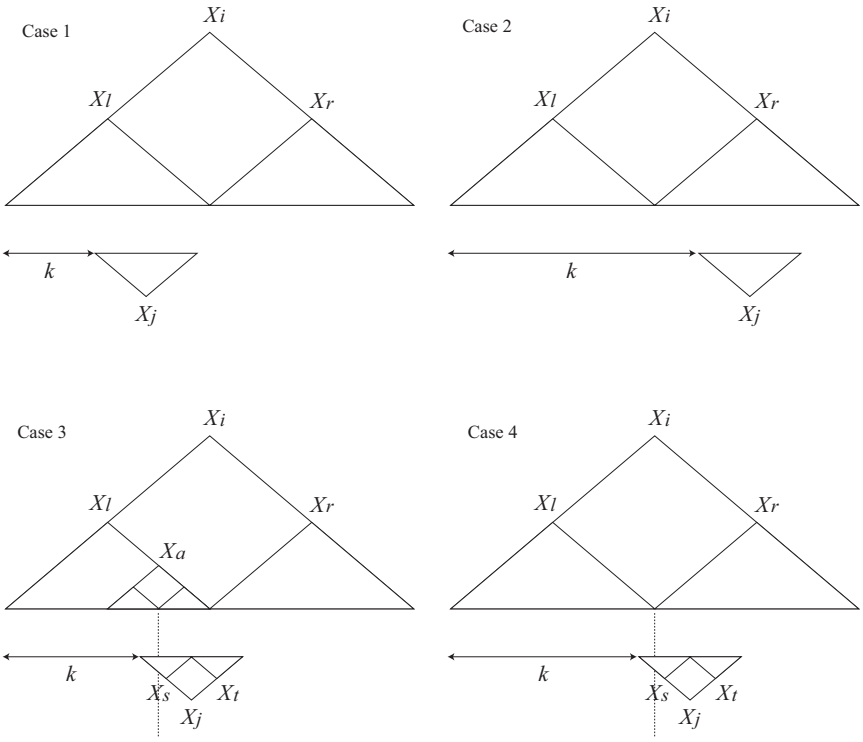

Fig. 7. Four possible cases in computing $FM(X_i, X_j, k)$, where X_i and X_j both form complete binary trees (see Lemma 9).

ii. If $k - |X_\ell| + |X_a| \in Occ^{\triangle}(X_a, X_s)$, then
$FM(X_n, X_j, k) = FM(X_n, X_t, k + |X_s|) + |X_s|$.

(b) If $|X_\ell| - d < k + |X_s| - 1 < |X_\ell| + 1$, then
i. If $k \notin Occ^{\triangle}(X_a, X_s)$, then
$FM(X_n, X_j, k) = FM(X_a, X_s, k - |X_\ell| + |X_a|)$.
ii. If $k \in Occ^{\triangle}(X_a, X_s)$, then
$FM(X_n, X_j, k) = FM(X_r, X_t, k + |X_s| - |X_\ell| + d) + |X_s|$.

(c) If $k + |X_s| - 1 \geq |X_\ell| + 1$, then let $X_{s(h)} = lmd(X_j, h)$ for each $1 \leq h \leq height(X_j)$. Also let $X_{s(h)} = X_{s(h-1)}X_{t(h-1)}$. We first compute $f = FM(X_n, X_{s(g)}, k)$, where $|X_\ell| - d < k + |X_{s(g)}| - 1 < |X_\ell| + 1$.
i. If $f \leq |X_{s(g)}|$, then $FM(X_n, X_j, k) = f$.
ii. If $f = |X_{s(g)}|+1$, then let $X_{r(h+1)} = lmd(X_r, h+1)$. Find the smallest $h \geq g$ such that $k - |X_\ell| + d + \sum_{p=g}^{h-1} |X_{s(p)}| \notin Occ^{\triangle}(X_{r(h+1)}, X_{t(h)})$.
A. If there is no such h, then
$FM(X_n, X_j, k) = k + |X_j|$.
B. Otherwise, $FM(X_n, X_j, k) = FM(X_{r(h+1)}, X_{t(h)}, k - |X_\ell| + d + \sum_{p=g}^{h-1} |X_{s(p)}|)$.

In each recursion except for Case 3c, either or both of the first and second variables in the FM function decrease the height by at least one. Hence it takes $O(\log N)$ time like Lemma 9. In Case 3c, the value of f is computable in $O(\log N)$ time by Case 3b. Finding the smallest h takes $O(height(X_j)) = O(\log |X_j|)$ time. Since computing $FM(X_{r(h+1)}, X_{t(h)}, k - |X_\ell| + d + \sum_{p=g}^{h-1} |X_{s(p)}|)$ will fall into one of the cases of Lemma 9, we can manage Case 3c in $O(\log N)$ time. □

When we test square-freeness of the last variable X_n, we sometimes need to compute the extended version of FM function for given strings X, Y, and two integers k, p, as follows:

$$FM(X, Y, k, p) = \\ \min\{1 \leq i \leq |Y| - p \mid X[k + i - 1] \neq Y[p + i]\}.$$

Lemma 11. *For any variables X_i, X_j with $1 \leq i, j < n$ and any integers k, p, $FM(X_i, X_j, k, p)$ can be computed in $O(\log^2 |X_i|)$ time.*

Proof. It is not difficult to see that $X_j[p : |X_j|]$ can be represented by a concatenation of variables $X_{j_1}, X_{j_2}, \ldots, X_{j_h}$ such that $|X_{j_1}| < |X_{j_2}| < \cdots < |X_{j_h}|$ and $h = O(height(X_j))$. Find the leftmost variable X_{j_s} such that $FM(X_i, X_{j_s}, k + \sum_{r=1}^{s-1} |X_{j_r}|) \neq 0$. Then clearly $FM(X_i, X_j, k, p) = \sum_{r=1}^{s-1} |X_{j_r}| + FM(X_i, X_{j_s}, k + \sum_{r=1}^{s-1} |X_{j_r}|)$. If such variable does not exist, then $FM(X_i, X_j, k, p) = |X_j| - p$. Since each of the variables $X_{j_1}, X_{j_2}, \ldots, X_{j_h}$ can be found in $O(height(X_i))$ time and each call of the FM function takes $O(\log |X_i|)$ time by Lemma 9, $FM(X_i, X_j, k, p)$ can be computed in total of $O(\log^2 |X_i|)$ time. □

7 Conclusions and Future Work

In this paper, we presented an $O(\max(n^2, n \log^2 N))$-time $O(n^2)$-space algorithm to test if a given BSLP-compressed string is square-free. Here, n is the size of BSLP and N is the length of the decompressed string.

Our future work includes the following.

- Apostolico and Breslauer [15] also presented a parallel algorithm to find the set of *all* squares from a given (uncompressed) string. Therefore, a natural question is if it is possible to extend our algorithm to detecting the set of all squares from BSLP-compressed string. A major task is how to represent the resulting set in polynomial space in the compressed size, since there are $\Theta(N \log N)$ occurrences of squares in a string of length N.
- Is it possible to extend our algorithm to general SLPs? Gasieniec et al. [16] claimed a polynomial time algorithm to find all squares from a given string compressed by composition systems, a generalization of SLPs. However, details of their algorithm have never been published unfortunately. Our algorithm is heavily dependant that the variables except for the last one form complete binary trees. Hence dealing with general SLPs does not seem as easy.

– Can we extend our algorithm to testing if a given BSLP-compressed string is *cube-free*? A cube is a string of the form xxx. If a string is square-free, then it is always cube-free. But the opposite is not true. A cube-free string may contain squares.

Acknowledgments

The authors thank Wojciech Rytter for leading us to reference [15].

References

1. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
2. Gusfield, D., Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. J. Comput. Syst. Sci. **69**(4) (2004) 525–546
3. Harrison, M.: Introduction to Formal Language Theory. Addison-Wesley (1978)
4. Lothaire, M.: Combinatorics on Words. Addison-Wesley (1983)
5. Thue, A.: Über unendliche Zeichenreihen. Norske Vid Selsk. Skr. I Mat-Nat Kl. (Christiana) **7** (1906) 1–22
6. Thue, A.: Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Norske Vid Selsk. Skr. I Mat-Nat Kl. (Christiana) **1** (1912) 1–67
7. Main, M.G., Lorentz, R.J.: Linear time recognition of squarefree strings. In: Combinatorial Algorithms on Words. Volume F12 of NATO ASI Series., Springer (1985) 271–278
8. Crochemore, M.: Transducers and repetitions. Theoretical Computer Science **12** (1986) 63–86
9. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. Information Processing Letters **12**(5) (1981) 244–250
10. Crochemore, M., Rytter, W.: Squares, cubes, and time-space efficient string searching. Algorithmica **13**(5) (1995) 405–425
11. Apostolico, A., Preparata, F.P.: Optimal off-line detection of repetitions in a string. Theoretical Computer Science **22** (1983) 297–315
12. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS'99). (1999) 596–604
13. Crochemore, M., Rytter, W.: Efficient parallel algorithms to test square-freeness and factorize strings. Information Processing Letters **38**(2) (1991) 57–60
14. Apostolico, A.: Optimal parallel detection of squares in strings. Algorithmica **8** (1992) 285–319
15. Apostolico, A., Breslauer, D.: An optimal $O(\log \log N)$-time parallel algorithm for detecting all squares in a string. SIAM J. Comput. **25**(6) (1996) 1318–1331
16. Gasieniec, L., Karpinski, M., Plandowski, W., Rytter, W.: Efficient algorithms for Lempel-Ziv encoding. In: Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT'96). Volume 1097 of Lecture Notes in Computer Science., Springer-Verlag (1996) 392–403
17. Hirao, M., Shinohara, A., Takeda, M., Arikawa, S.: Faster fully compressed pattern matching algorithm for balanced straight-line programs. In: Proc. 7th International Symp. on String Processing and Information Retrieval (SPIRE'00), IEEE Computer Society (2000) 132–138

18. Karpinski, M., Rytter, W., Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions. Nordic Journal of Computing **4** (1997) 172–186
19. Miyazaki, M., Shinohara, A., Takeda, M.: An improved pattern matching algorithm for strings in terms of straight-line programs. In: Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97). Volume 1264 of Lecture Notes in Computer Science., Springer-Verlag (1997) 1–11
20. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Computing longest common substring and all palindromes from compressed strings. In: Proc. 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'08). Volume 4910 of Lecture Notes in Computer Science., Springer-Verlag (2008) 364–375
21. Inenaga, S., Shinohara, A., Takeda, M.: An efficient pattern matching algorithm on a subclass of context free grammars. In: Proc. Eighth International Conference on Developments in Language Theory (DLT'04). Volume 3340 of Lecture Notes in Computer Science., Springer-Verlag (2004) 225–236
22. Lifshits, Y.: Processing compressed texts: A tractability border. In: Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM'07). Volume 4580 of Lecture Notes in Computer Science., Springer-Verlag (2007) 228–240

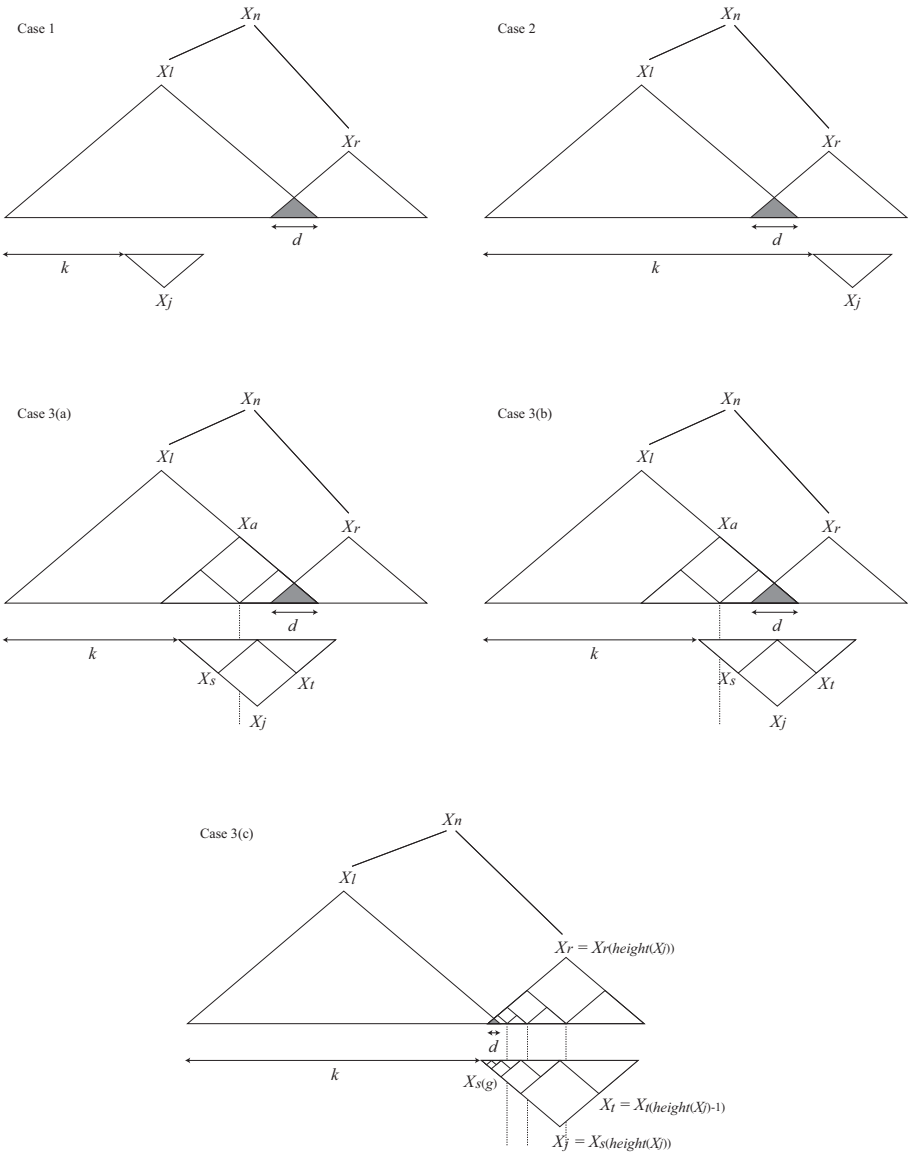

Fig. 8. Five possible cases in computing $FM(X_n, X_j, k)$, where $X_n = X_\ell^{[d]} X_r$ is the last variable of a BSLP (see Lemma 10).