CHICAGO JOURNAL OF THEORETICAL COMPUTER SCIENCE 2010, Article 7, pages 1–20
http://cjtcs.cs.uchicago.edu/

SPECIAL ISSUE FOR CATS 2009

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

Wataru Matsubara Shunsuke Inenaga Ayumi Shinohara

Received: January 15, 2009; revised: October 14, 2009; published: June 22, 2010.

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.

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 [11, 12], formal language theory [13] and combinatorics on words [25]. 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

DOI: 10.4086/cjtcs.2010.007

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 by Thue [30, 31] 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 [26] presented an $O(N)$-time algorithm to test if a given string of length N is square-free. Crochemore [7] 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 $\Theta(N \log N)$ [6, 9]. Optimal $O(N \log N)$ algorithms that detect all occurrences of squares of a given string have been proposed [6, 7, 3]. Kolpakov and Kucherov [21] showed that any string of length N can contain $O(N)$ *syntactically distinct* squares, i.e., the size of the set of squares included in the string is $O(N)$. They also developed an $O(N)$-time algorithm to find all syntactically distinct squares from a given string.

There are also several efficient parallel algorithms for the above problems. Crochemore and Rytter [8] discovered a parallel algorithm to test square-freeness of a string, which runs in $O(\log N)$ time using N processors. Apostolico [1] showed an algorithm that can find all occurrences of squares with the same time bound and the number of processors. Then, Apostolico and Breslauer [2] 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 in 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 by Gasieniec et al. [10]. 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 Hirao et al. [14]. BSLP is a variant of *straight line program* (*SLP*) which has widely been studied [17, 18, 29, 24, 23, 5]. SLP is regarded as a kind of context-free grammar (CFG) which generates a single string. SLP is a CFG in the Chomsky normal form, that is, the production rules are in either of the form $X \to YZ$ or $X \to \mathtt{a}$. We remark that BSLP has *exponential compression*, that is, $N = O(2^n)$ [14]. For instance, consider a BSLP $\mathcal{T}$ consisting of n variables such that $X_1 \to \mathtt{a}$, $X_2 \to \mathtt{b}$, $X_3 \to X_1X_2$ and $X_i \to X_{i-1}X_{i-1}$ for every $4 \leq i \leq n$. Note that BSLP $\mathcal{T}$ derives a string $(\mathtt{ab})^{2^{n-3}}$ of length $N = 2^{n-2}$. Therefore, our algorithm is more efficient than any algorithms that decompress a given BSLP-compressed string in the worst case.

2 Preliminaries

2.1 Notation

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

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

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,

$$Occ^{\triangle}(w,x,k) = \{s \mid w[s : s+|x|-1] = x, k-|x| \leq s \leq k+1\}.$$

We will heavily use the following lemma.

Lemma 2.1 ([28]). *For any strings w, x, and integer k, $Occ^{\triangle}(w,x,k)$ forms a single arithmetic progression.*

The above lemma implies that the set of all occurrences of x in w that touch or cover position k forms a single arithmetic progression.

Example 2.2. Consider string $w = \texttt{aaababababab}$ and $x = \texttt{ababa}$ and integer $k = 7$. We have $Occ^{\triangle}(w,x,k) = \{3,5,7\}$ which 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 2.3. 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 forms:

- $expr_i = a \quad (a \in \Sigma)$ 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 context free grammar 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:

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

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

Lemma 2.4 ([27]). *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.

2.3 Balanced Straight Line Program

We define a *balanced straight line program* (*BSLP*), which is a variant of an SLP of Definition 2.3.

Definition 2.5. 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 one of the following forms:

- $expr_i = a \quad (i < n, a \in \Sigma)$ 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 (for a concrete example, see the last paragraph of Section 1).

Example 2.6. 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 graph of BSLP $\mathcal{T}$ is shown in Figure 1.

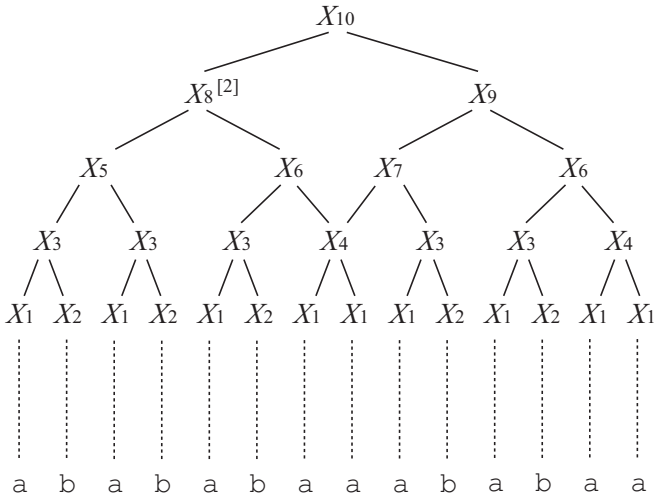

Figure 1: The derivation graph of BSLP $\mathcal{T}$ of Example 2.6 that generates string $T = \texttt{abababaaababaa}$. Recall that $X_8^{[2]}$ denotes the prefix of X_8 of length $|X_8| - 2$.

3 Apostolico and Breslauer's Algorithm

In this section we recall a parallel algorithm of [2] that checks square-freeness of a string, on which our algorithm will be based.

For any strings X, Y, and any integer k with $1 \leq k \leq |X|$, we define the function $FM(X,Y,k)$ which returns the length of the longest common prefix of $X[k : |X|]$ and Y, that is,

$$FM(X,Y,k) = LCP(X[k : |X|], Y), \tag{3.1}$$

where $LCP(X[k : |X|], Y)$ denotes the length of the longest common prefix of $X[k : |X|]$ and Y.

Example 3.1. Let $X = \texttt{aa}\underline{\texttt{abbab}}\texttt{ba}$, $Y = \underline{\texttt{abbab}}\texttt{a}$, and $k = 3$. Then $FM(X,Y,k) = 6$. The longest common prefix $\texttt{abbab}$ of $X[3 : |X|]$ and Y is underlined above.

Let w be any string of length N, where N is a power of 2. For each t with $0 \leq t \leq \log_2 N - 1$, partition w into consecutive blocks of length $m = 2^t$. Now, for any t, let $B = w[k : k+m-1]$ be any block of length $m = 2^t$ with some $k = 1, m, 2m, \ldots, N-m+1$. 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 sets 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

$$\begin{aligned} \alpha &= FM(w, w[k+m : p-1], p+m), \\ \gamma &= FM(w^R, w^R[N-k+1 : N-k+m], N-p+1). \end{aligned}$$

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

Lemma 3.2 ([2]). *There exist repetitions zz which are hinged on B with $|z| = p-k$, if and only if $p-\gamma \leq k+m+\alpha$.*

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 3.3 ([2]). *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.

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: $\mathsf{B} = \mathsf{w}[\mathsf{k} : \mathsf{k}+\mathsf{m}-1]$;
2: $\mathsf{P}_1 =$ the set of occurrence positions of B in $[\mathsf{k}+2\mathsf{m} : \mathsf{k}+4\mathsf{m}-1]$);
3: $\mathsf{P}_2 =$ the set of occurrence positions of B in $[\mathsf{k}+3\mathsf{m} : \mathsf{k}+5\mathsf{m}-1]$);
4: **if** $\mathsf{P}_1 \cup \mathsf{P}_2 > 2$ **then return** true;
5: **for each** $\mathsf{p} \in \mathsf{P}_1 \cup \mathsf{P}_2$ **do**
6: $\quad \alpha = FM(\mathsf{w}, \mathsf{w}[\mathsf{k}+\mathsf{m} : \mathsf{p}-1], \mathsf{p}+\mathsf{m})$;
7: $\quad \gamma = FM(\mathsf{w}^R, \mathsf{w}^R\mathsf{N}-\mathsf{k}+1 : \mathsf{N}-\mathsf{k}+\mathsf{m}], \mathsf{N}-\mathsf{p}+1)$;
8: $\quad$ **if** $\mathsf{p}-\gamma \leq \mathsf{k}+\mathsf{m}+\alpha$ **then return** true;
9: **return** false;

The algorithm of [2] 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 $2^t = m$, based on Lemma 3.2 and Lemma 3.3. 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.

4 Testing Square-freeness of BSLP-Compressed Strings

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

4.1 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 4.1. 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).

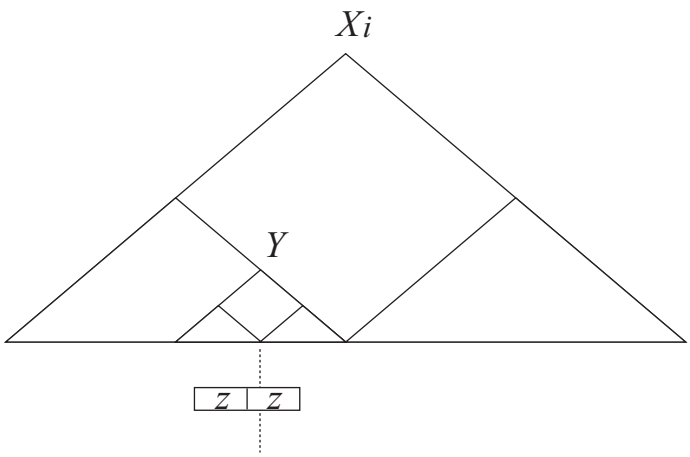

Figure 2: Illustration of Observation 1. Repetition zz is a substring of Y and covers the boundary of Y.

For any variable $X_i = X_\ell X_r$, we define the *boundary* of X_i to be the position $|X_\ell|$ in string X_i. We say that a string v *touches the boundary of* X_i if $v = X_r[1 : |v|]$. We say that a string s *covers the boundary of* X_i if $v = X_i[s : s + |v| - 1] = v$ for some $|X_\ell| - |v| \leq s \leq |X_\ell|$.

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 4.1 is reduced to the following sub-problem.

Problem 4.2. 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 4.2. The algorithm is based on the parallel algorithm of [2] summarized in Section 3.

Lemma 4.3. *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 10 such descendants of height h for each $1 \leq h \leq height(X_i) - 2$. (See also Figure 3.)*

Proof. Recall that repetition zz is 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, the first z of the repetition has to occur in $X_i[|X_\ell| - 2^{h+2} + 5 : |X_\ell| + 2^{h+1} - 1]$. It is clear that the substring contains 10 variables of length 2^{h-1}, each being of height h. That is, for the beginning position s of each such variable Y in X_i we have

$$|X_i| - |X_\ell| - 7|Y| \leq s \leq |X_\ell| + 2|Y|.$$

□

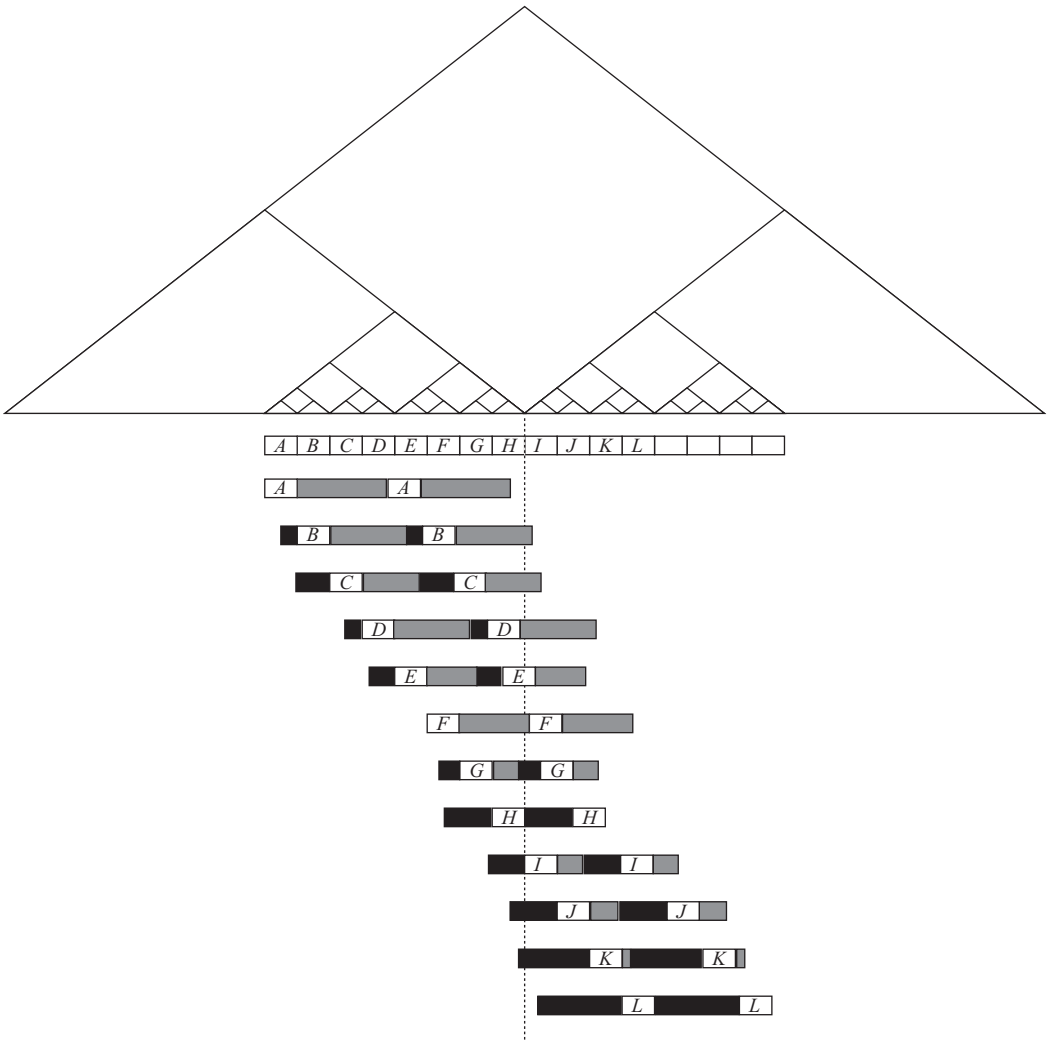

Figure 3: Illustration of Lemma 4.3 for height $h = height(|X_i|) - 5$. No repetitions zz hinged on variable A or L can touch or cover the boundary of X_i, since repetitions zz are hinged on variable Y only if $2 \times |Y| - 1 = 2^h - 1 \leq |z| < 2^{h+1} - 1 = 4 \times |Y| - 1$, where $|Y| = |A| = |L|$.

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 4.4 ([14]). *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.*

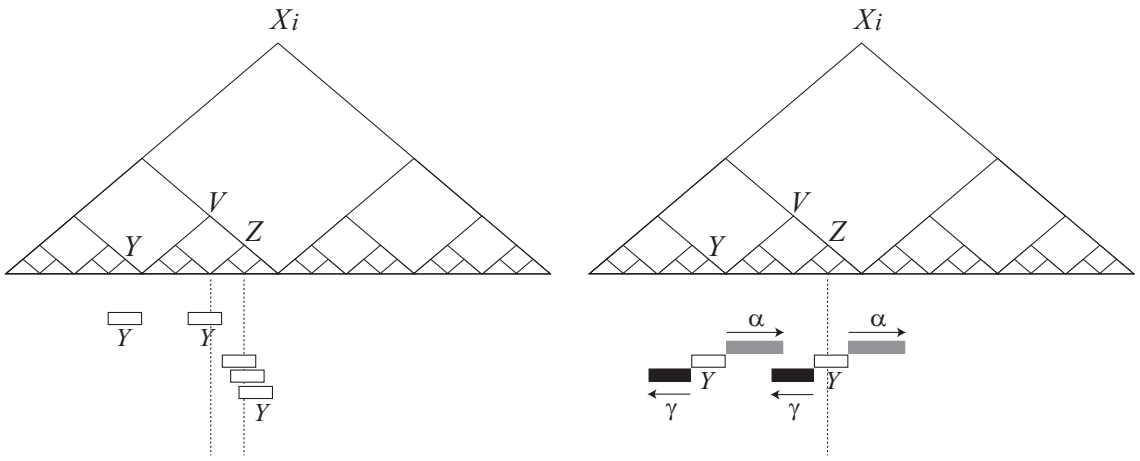

Figure 4: Illustration of Lemma 4.5. The left is Case 1, and the right is Case 2.

We are ready to state the next lemma.

Lemma 4.5. *Problem 4.2 can be solved in $O(\log^2 |X_i|)$ time with $O(n^2)$ preprocessing.*

Proof. We process a given variable X_i in $height(X_i) - 2$ stages, where each stage is associated with height h, such that $1 \leq h \leq height(X_i) - 2$. In each stage with height h, there are at most 10 descendants to consider by Lemma 4.3. 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 2.1 that $Occ^{\triangle}(V,Y)$ and $Occ^{\triangle}(Z,Y)$ form a single arithmetic progression. Due to Lemma 3.3,

1. If $|Occ^{\triangle}(V,Y) \cup Occ^{\triangle}(Z,Y)| > 2$, then X_i is not repetition-free (see the left of Figure 4).

2. If $|Occ^{\triangle}(V,Y) \cup Occ^{\triangle}(Z,Y)| \leq 2$, then we compute the values of α and γ according to Lemma 3.2 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 4.4. 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 2.4, these reversed variables can be precomputed in $O(n)$ time. As to be shown in Section 5, 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 4.2 is shown in Algorithms 2 and 3.

Algorithm 2. Function *HingedSqBSLP*(X, 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 $\mathsf{s}+3|\mathsf{Y}|-1$ in X;
2: Z = a variable whose boundary is at position $\mathsf{s}+4|\mathsf{Y}|-1$ in X;
3: **if** $|Occ^{\triangle}(\mathsf{V},\mathsf{Y}) \cup Occ^{\triangle}(\mathsf{Z},\mathsf{Y})| > 2$ **then return** true;
4: **for each** $\mathsf{p} \in Occ^{\triangle}(\mathsf{V},\mathsf{Y}) \cup Occ^{\triangle}(\mathsf{Z},\mathsf{Y})$ **do**
5: compute α by at most two calls of *FM*;
6: compute γ by at most two calls of *FM*;
7: **if** $\mathsf{p}-\gamma \leq \mathsf{s}+|\mathsf{Y}|+\alpha$ **then return** true;
8: **return** false;

Algorithm 3. Function *TestSqBSLPVar*(X_i) to test square-freeness of a BSLP variable X_i.

Input: BSLP variable X_i with $\mathsf{i} \neq \mathsf{n}$.
Output: Whether there exists a square in X_i.
1: **for each** $\mathsf{h} = 1$ **to** $height(\mathsf{X})-2$ **do**
2: **for each** descendant Y of X such that $height(\mathsf{Y}) = \mathsf{h}$,
$\mathsf{Y} = \mathsf{X}[\mathsf{s} : \mathsf{s}+2^{h-1}-1]$, and $|\mathsf{X}|/2-7|\mathsf{Y}| \leq \mathsf{s} \leq |\mathsf{X}|/2+2|\mathsf{Y}|$ **do**
3: **if** *HingedSqBSLP*(X, Y) =true **then return** true;
4:
5: **return** false;

4.2 Testing Square-freeness of BSLP-compressed Strings

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

Problem 4.6 (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 4.9, which is the main result of this subsection.

Lemma 4.7 ([16])**.** *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 $1 \leq j' \leq j$.*

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 ($1 \le i < n$) and height h ($< height(X_i)$), we pre-compute 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 [14].

Lemma 4.8. *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 $1 \le i' \le n$ and $1 \le j' \le n$.*

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).

$$Occ^{\triangle}(X_n, X_j, |X_\ell|) = \begin{cases} Occ^{\triangle}(X_r, X_j, d) \oplus (|X_\ell| - d) & \text{if } |X_j| \le d, \\ p_1 \cup p_2 & \text{if } |X_j| > d, \end{cases}$$

where

$$\begin{aligned} p_1 &= (Occ^{\triangle}(X_a, X_s) \oplus (|X_\ell| - |X_a|) \cap Occ^{\triangle}(X_n, X_t, |X_\ell|) \ominus |X_s|), \\ p_2 &= (Occ^{\triangle}(X_n, X_s, |X_\ell|) \cap Occ^{\triangle}(X_r, X_t, |X_s| + d) \oplus (|X_\ell| - d - |X_s|). \end{aligned}$$

It can be shown in a similar way to Lemma 5 of [14] 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 4.7 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 4.9. *Problem 4.6 can be solved in $O(\max(n^2, n \log^2 N))$ time using $O(n^2)$ space.*

Proof. By Lemma 4.5, 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 $1 \le i < n$, 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 $|X_\ell| - 2|z| + 1 < s \le |X_\ell| - d$.

The testing algorithm is a modification of that of Lemma 4.5. We process X_n with at most $\max(height(X_\ell), height(X_r)) - 2$ stages. Let us focus on some variable Y of height

$$h \le \max(height(X_\ell), height(X_r)) - 2$$

on which a repetition satisfying the above condition may be hinged. See also Figure **??**. We can compute $Occ^{\triangle}(X_n, Y, |X_\ell|)$ in $O(\log^2 N)$ time by Lemma 4.8 (the left arithmetic progression in Figure **??**). By

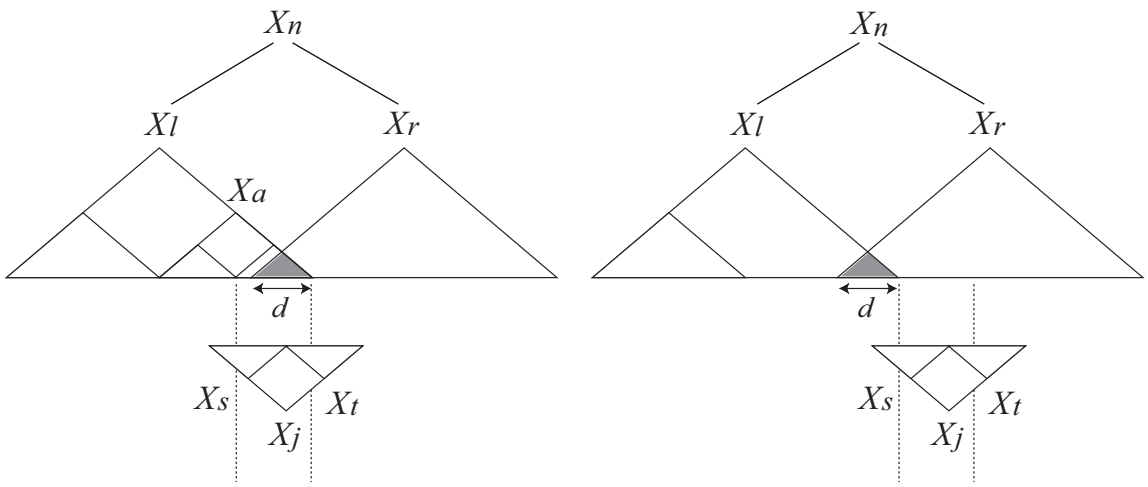

Figure 5: Illustration of Lemma 4.8. If $|X_j| > d$, $Occ^{\triangle}(X_n, X_j, |X_\ell|)$ is equal to the union of p_1 (the left) and p_2 (the right).

Lemma 4.7, $Occ^{\triangle}(X_r, Y, |Y|+d)$ can be computed in $O(\log N)$ time (the right arithmetic progression in Figure **??**). As to be shown by Lemma 5.2 and Lemma 5.3 in Section 5, 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 $\max(height(X_\ell), height(X_r)) < \log_2 N + 1$, 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 4.6 is shown in Algorithms 4 and 5.

Algorithm 4. Function *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 $Occ^{\triangle}(\mathsf{X_n}, \mathsf{Y}, |\mathsf{X}_\ell|)$;
2: compute $Occ^{\triangle}(\mathsf{X_r}, \mathsf{Y}, |\mathsf{Y}| + \mathsf{d})$;
3: **if** $|Occ^{\triangle}(\mathsf{X_n}, \mathsf{Y}, |\mathsf{X}_\ell|) \cup Occ^{\triangle}(\mathsf{X_r}, \mathsf{Y}, |\mathsf{Y}| + \mathsf{d})| > 2$ **then return** true;
4: **for each** $\mathsf{p} \in Occ^{\triangle}(\mathsf{X_n}, \mathsf{Y}, |\mathsf{X}_\ell|) \cup Occ^{\triangle}(\mathsf{X_r}, \mathsf{Y}, |\mathsf{Y}| + \mathsf{d})$ **do**
5: compute α by at most two calls of *FM*;
6: compute γ by at most two calls of *FM*;
7: **if** $\mathsf{p} - \gamma \leq \mathsf{s} + |\mathsf{Y}| + \alpha$ **then return** true;
8: **return** false;

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 $\mathsf{X_n} = \mathsf{X}_\ell^{[\mathsf{d}]}\mathsf{X_r}$.

1: **for each** $\mathsf{i} = 1$ **to** $\mathsf{n}-1$ **do**
2: **for each**$\mathsf{j} = 1$ **to** $\mathsf{n}-1$ **do**
3: compute $Occ^{\triangle}(\mathsf{X_i}, \mathsf{X_j})$;
4: compute $Occ^{\triangle}(\mathsf{X_i}^R, \mathsf{X_j}^R)$;
5:
6: **if** $IsSqfreeBSLPBin(\mathsf{X}_\ell) = \mathsf{true}$ **then return** true;
7: **if** $IsSqfreeBSLPBin(\mathsf{X_r}) = \mathsf{true}$ **then return** true;
8: **for** $\mathsf{h} = 1$ **to** $\max(height(\mathsf{X}_\ell), height(\mathsf{X_r})) - 2$ **do**
9: **for each** descendant Y of $\mathsf{X_n}$ such that $height(\mathsf{Y}) = \mathsf{h}$ **do**
$\mathsf{Y} = \mathsf{X_n}[\mathsf{s} : \mathsf{s} + 2^{h-1} - 1]$, and $|\mathsf{X_n}|/2 - 7|\mathsf{Y}| \le \mathsf{s} \le |\mathsf{X_n}|/2 + 2|\mathsf{Y}|$;
• Lemma 4.3
10: **if** $HingedSqBSLPLast(\mathsf{X_n}, \mathsf{Y}) = \mathsf{true}$ **then return** true;
11:
12: **return** false;

---

5 Computing the *FM* Function

The *FM* function in equation (3.1) plays a central role in our algorithms to compute squares from BSLP-compressed strings. In our problem setting, the first 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 [28, 23]. 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 5.1. *For any variables $X_i = X_\ell X_r$, $X_j = X_s X_t$ with $1 \le i, j < n$ and integer $1 \le k \le |X_i|$, $FM(X_i, X_j, k)$ can be computed in $O(\log|X_i|)$ time, provided that $Occ^{\triangle}(X_{i'}, X_{j'})$ is already computed for every $1 \le i' \le i$ and $1 \le j' \le j$.*

Proof. We can recursively compute $FM(X_i, X_j, k)$, as follows (see also Figure 6):

1. If $k + |X_j| \le |X_\ell|$, 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 $k + |X_s| \le |X_\ell| < k + |X_j|$, 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 $FM(X_i, X_j, k) = FM(X_i, X_t, k + |X_s|) + |X_s|$.
4. If $k < |X_\ell| < k + |X_s|$, then we have the two following sub-cases.

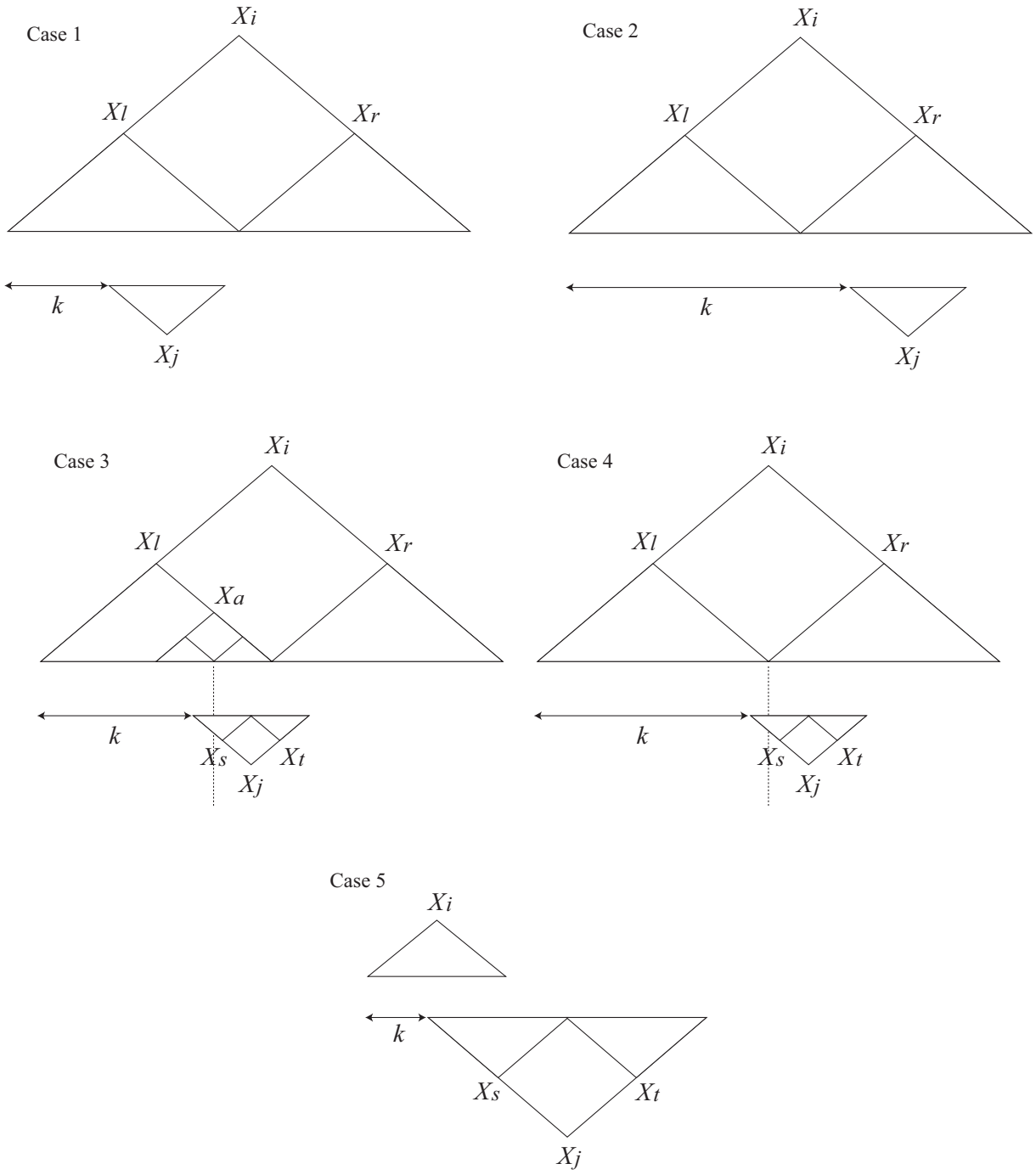

Figure 6: Five possible cases in computing $FM(X_i, X_j, k)$, where X_i and X_j both form complete binary trees (see Lemma 5.1).

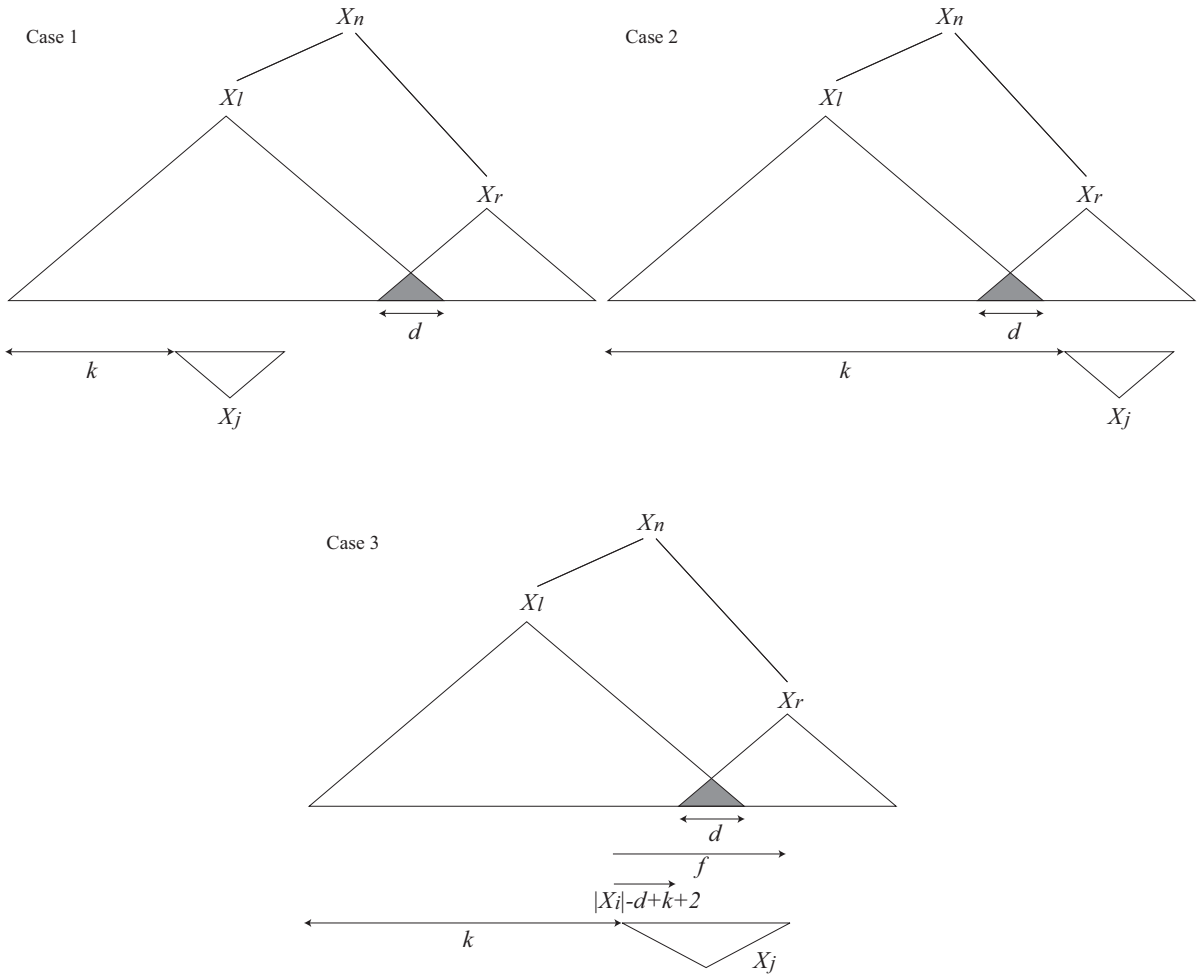

Figure 7: Five possible cases in computing $FM(X_n, X_j, k)$, where $X_n = X_\ell^{[d]} X_r$ is the last variable of a BSLP (see Lemma 5.2).

(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 $FM(X_i, X_j, k) = FM(X_r, X_t, k + |X_s| - |X_\ell|) + |X_s|$.

5. If $|X_i| < k + |X_s|$, then $FM(X_i, X_j, k) = FM(X_i, X_s, k)$.

In each recursion, at least one 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 5.2. *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 7):

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 first compute $f = FM(X_\ell, X_j, k)$.
   (a) If $f \leq |X_\ell| - k$, then $FM(X_n, X_j, k) = f$.
   (b) If $f > |X_\ell| - k$, then $FM(X_n, X_j, k) = FM(X_j, X_r, |X_\ell| - d - k + 2) + |X_\ell| - d - k + 1$.

In each recursion except for Case 3b, at least one of the first and second variables in the *FM* function decrease the height by at least one. Hence it takes $O(\log N)$ time as in Lemma 5.1.

Since computing the values f and $FM(X_j, X_r, |X_\ell| - d - k + 2)$ will fall into one of the cases of Lemma 5.1, we can manage Case 3b 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:

$$FM(X, Y, k, p) = LCP(X[k : |X|], Y[p : |Y|]).$$

Lemma 5.3. *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 $FM(X_i, X_{j_s}, k + \sum_{r=1}^{s-1} |X_{j_r}|) < |X_{j_s}|$. 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 $FM(X_i, X_j, k, p) = |X_j| - p + 1$. 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 5.1, $FM(X_i, X_j, k, p)$ can be computed in total of $O(\log^2 |X_i|)$ time. □

6 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 [2] 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. It might be helpful to consider to compute the set of all *runs* [21].

- It is of interest if we can detect more practical patterns such as e.g., *approximate repetitions* [22, 20] and *gapped repetitions* [4, 19], *quasiperiodic repetitions* [15] from compressed strings. It might be reasonable to start with finding such repetitions of fixed length, rather than finding repetitions of arbitrary length.

- Is it possible to extend our algorithm to general SLPs? Gasieniec et al. [10] claimed a polynomial time algorithm to find all squares from a given string compressed by composition systems, a generalization of SLPs. However, unfortunately details of their algorithm have never been published. 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 [2]. The authors also thank the anonymous referees for their fruitful comments and suggestions.

References

[1] ALBERTO APOSTOLICO: Optimal parallel detection of squares in strings. *Algorithmica*, 8:285–319, 1992. 2

[2] ALBERTO APOSTOLICO AND DANY BRESLAUER: An optimal $O(\log \log N)$-time parallel algorithm for detecting all squares in a string. *SIAM J. Comput.*, 25(6):1318–1331, 1996. 2, 5, 6, 7, 17

[3] ALBERTO APOSTOLICO AND FRANCO P. PREPARATA: Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983. 2

[4] GERTH S. BRODAL, RUNE B. LYNGSØ, CHRISTIAN N.S. PEDERSEN, AND JENS STOYE: Finding maximal pairs with bounded gap. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, volume 1645 of *Lecture Notes in Computer Science*, pp. 134–149. Springer-Verlag, 1999. 17

[5] FRANCISCO CLAUDE AND GONZALO NAVARRO: Self-indexed text compression using straight-line programs. In *Proc. 34th International Symposium on Mathematical Foundations of Computer Science (MFCS'09)*, Lecture Notes in Computer Science. Springer-Verlag, 2009. To appear. 2

[6] MAXIME CROCHEMORE: An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981. 2

[7] MAXIME CROCHEMORE: Transducers and repetitions. *Theoretical Computer Science*, 12:63–86, 1986. 2

[8] MAXIME CROCHEMORE AND WOJCIECH RYTTER: Efficient parallel algorithms to test square-freeness and factorize strings. *Information Processing Letters*, 38(2):57–60, 1991. 2

[9] MAXIME CROCHEMORE AND WOJCIECH RYTTER: Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. 2

[10] LESZEK GASIENIEC, MAREK KARPINSKI, WOJCIECH PLANDOWSKI, AND WOJCIECH RYTTER: Efficient algorithms for Lempel-Ziv encoding. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT'96)*, volume 1097 of *Lecture Notes in Computer Science*, pp. 392–403. Springer-Verlag, 1996. 2, 17

[11] DAN GUSFIELD: *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997. 1

[12] DAN GUSFIELD AND JENS STOYE: Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004. 1

[13] M. HARRISON: *Introduction to Formal Language Theory*. Addison-Wesley, 1978. 1

[14] MASAHIRO HIRAO, AYUMI SHINOHARA, MASAYUKI TAKEDA, AND SETSUO ARIKAWA: Faster fully compressed pattern matching algorithm for balanced straight-line programs. In *Proc. 7th International Symp. on String Processing and Information Retrieval (SPIRE'00)*, pp. 132–138. IEEE Computer Society, 2000. 2, 8, 11

[15] COSTAS S. ILIOPOULOS AND LAURENT MOUCHARD: Quasiperiodicity: From detection to normal forms. *Journal of Automata, Languages and Combinatorics*, 4(3):213–228, 1999. 17

[16] SHUNSUKE INENAGA, AYUMI SHINOHARA, AND MASAYUKI TAKEDA: 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*, pp. 225–236. Springer-Verlag, 2004. 10

[17] 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*, pp. 205–214. Springer-Verlag, 1995. 2

[18] M. KARPINSKI, W. RYTTER, AND A. SHINOHARA: An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing*, 4:172–186, 1997. 2

[19] ROMAN KOLPAKOV AND GREGORY KUCHEROV: Finding repeats with fixed gap. In *Proc. 7th International Symposium on String Processing Information Retrieval (SPIRE'00)*, p. 162, 2000. 17

[20] ROMAN KOLPAKOV AND GREGORY KUCHEROV: Finding approximate repetitions under hamming distance. *Theoretical Computer Science*, 303(1):135–156, 2003. 17

[21] ROMAN M. KOLPAKOV AND GREGORY KUCHEROV: Finding maximal repetitions in a word in linear time. In *Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*, pp. 596–604, 1999. 2, 17

[22] GAD M. LANDAU, JEANETTE P. SCHMIDT, AND DINA SOKOL: An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001. 17

[23] YURY LIFSHITS: 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*, pp. 228–240. Springer-Verlag, 2007. 2, 13

[24] YURY LIFSHITS AND MARKUS LOHREY: Querying and embedding compressed texts. In *Proc. 31st International Symposium on Mathematical Foundations of Computer Science (MFCS'06)*, volume 4162 of *Lecture Notes in Computer Science*, pp. 681–692. Springer-Verlag, 2006. 2

[25] M. LOTHAIRE: *Combinatorics on Words*. Addison-Wesley, 1983. 1

[26] M. G. MAIN AND R. J. LORENTZ: Linear time recognition of squarefree strings. In *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pp. 271–278. Springer, 1985. 2

[27] WATARU MATSUBARA, SHUNSUKE INENAGA, AKIRA ISHINO, AYUMI SHINOHARA, TOMOYUKI NAKAMURA, AND KAZUO HASHIMOTO: 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*, pp. 364–375. Springer-Verlag, 2008. 4

[28] MASAMICHI MIYAZAKI, AYUMI SHINOHARA, AND MASAYUKI TAKEDA: 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*, pp. 1–11. Springer-Verlag, 1997. 3, 13

[29] WOJCIECH RYTTER: Grammar compression, LZ-encodings, and string algorithms with implicit input. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP'04)*, volume 3142 of *Lecture Notes in Computer Science*, pp. 15–27. Springer-Verlag, 2004. 2

[30] ALEX THUE: Über unendliche zeichenreihen. *Norske Vid Selsk. Skr. I Mat-Nat Kl. (Christiana)*, 7:1–22, 1906. 2

[31] ALEX THUE: Über die gegenseitige lage gleicher teile gewisser zeichenreihen. *Norske Vid Selsk. Skr. I Mat-Nat Kl. (Christiana)*, 1:1–67, 1912. 2

AUTHORS

Wataru Matsubara
Graduate School of Information Sciences
Tohoku University, Japan

Shunsuke Inenaga
Graduate School of Information Science and Electrical Engineering
Kyushu University, Japan

Ayumi Shinohara
Graduate School of Information Sciences
Tohoku University, Japan