Computing All Squares in Compressed Texts

Lesha Khvorost
Ural Federal University
jaamal@mail.ru

Abstract

We consider the problem of computing all squares in a string represented by a straight-line program (SLP). An instance of the problem is an SLP $\mathbb{S}$ that derives some string S and we seek a solution in the form of a table that contains information about all squares in S in a compressed form. We present an algorithm that solves the problem in $O(|\mathbb{S}|^4 \log^2 |S|)$ time and requires $O(|\mathbb{S}|^2)$ space, where $|\mathbb{S}|$ (respectively $|S|$) stands for the size of the SLP $\mathbb{S}$ (respectively the length of the string S).

1 Introduction

Various compressed representations of strings are known: straight-line programs (SLPs) [10–12, 15], collage-systems [9], string representations using antidictionaries [16], etc. Nowadays text compression based on context-free grammars such as SLPs attracts much attention. The reason for this is not only that grammars provide well-structured compression but also that the SLP-based compression is in a sense polynomially equivalent to the compression achieved by the Lempel-Ziv algorithm that is widely used in practice. It means that, given a string S, there is a polynomial relation between the size of an SLP that derives S and the size of the dictionary stored by the Lempel-Ziv algorithm [15].

While compressed representations save storage space, there is a price to pay: some classical problems on strings become computationally hard when one deals with compressed data and measures algorithms' speed in terms of the size of compressed representations. As examples we mention here the problems **Hamming distance** [10] and **Literal shuffle** [3]. On the other hand, there exist problems that admit algorithms working rather well on compressed representations: **Pattern matching** [10, 14], **Longest common substring** [12], **Computing all palindromes** [12]. This dichotomy gives rise to the following research direction: to classify important string problems by their behavior with respect to compressed data.

Computing All Squares (**CAS**) is a natural problem on strings some of whose variants are of importance for molecular biology. (We just mention in passing a typical biological application [4] in which repeats in mouse genome were employed to trace the migration of mouse subspecies through Eurasia.) Up to recently it is was not known whether or not **CAS** admits an algorithm

polynomial in the size of a compressed representation of a given string.[1] The question is rather non-trivial because, in general, a string can have exponentially many squares with respect to the size of its compressed representation. For example, the string a^n has $\Theta(n^2)$ squares, while it is easy to build an SLP of size $O(\log n)$ that derives a^n. Thus, if we look for a polynomial algorithm for **CAS**, we have to develop a suitable data structure to store information about squares in a compressed form. Also, the fact that the number of squares may be quite large implies that a polynomial algorithm cannot search for squares consecutively by moving from one square to the "next" one. Squares should be somehow grouped in relatively large families that are to be discovered at once. The aim of the present paper is to demonstrate that these difficulties can be overcome for the case where strings are represented via SLPs.

The paper is structured as follows. Section 2 gathers some preliminaries concerning strings and SLPs. Section 3 collects brief descriptions and complexity analysis of some basic operations over SLPs that are frequently used in the paper. In Section 4 we present a polynomial algorithm for **CAS**. In Section 5 we discuss our results and their relation to other recent work in the area [2,13].

The main result of the paper has been announced in [8]. It relies on an earlier algorithm by the author [7] which was developed to find all *pure* squares (squares of primitive words) in a text derived from a given SLP.

2 Preliminaries

We consider strings of characters from a fixed finite alphabet Σ. The *length* of a string S is the number of its characters and is denoted by $|S|$. The *concatenation* of strings S_1 and S_2 is denoted by $S_1 \cdot S_2$ or simply by S_1S_2. A *position* in a string S is a point between two consecutive characters. We number positions from left to right by $1, 2, \ldots, |S|-1$. It is convenient to consider also the position 0 preceding the string and the position $|S|$ following it. For an integer i with $0 \le i \le |S|$ we denote by $S[i]$ the character between the positions i and $i+1$ of S. For example, $S[0]$ is the first character of S. A *substring* of S starting at a position ℓ and ending at a position r where $0 \le \ell < r \le |S|$ is denoted by $S[\ell \ldots r]$ (in other words, $S[\ell \ldots r] = S[\ell] \cdot S[\ell+1] \cdot \ldots \cdot S[r-1]$). We say that a substring $S[\ell \ldots r]$ *touches* a position t if $\ell \le t \le r$.

A string is called a *square* if it can be obtained by concatenating two copies of some string called the *root* of the square. A square xx is called *pure* if x occurs exactly two times in xx. If p is a positive integer, a string S is called *p-periodic* if for every position i with $0 \le i < |S-p|$, the equality $S[i] = S[i+p]$ holds. The integer p is then referred to as a *period* of S. By the classic Fine–Wilf theorem [5], each period p of a string S such that $p \le |S|/2$ is a multiple of the least period of S. A p-periodic substring of a string S is said to be *maximal* if it is not contained in any longer p-periodic substring of S.

A *straight-line program* (SLP) $\mathbb{S}$ is a sequence of *rules*, that is assignments

[1] A polynomial algorithm that solves **CAS** for strings represented by Lempel-Ziv encodings was announced in [6]. This representation is slightly more general than that by SLPs. However, to the best of our knowledge, no details of the algorithm have ever been published.

of the form:

$$\mathbb{S}_0 := expr_0,\ \mathbb{S}_1 := expr_1, \ldots, \mathbb{S}_n := expr_n, \tag{1}$$

where each $expr_i$ is either a letter from Σ (in this case the rule $\mathbb{S}_i := expr_i$ is said to be *terminal*) or an expression of the form $\mathbb{S}_\ell \cdot \mathbb{S}_r$ with $0 \le \ell, r < i$ (in this case the rule $\mathbb{S}_i := expr_i$ is called *nonterminal*). Thus, an SLP is a context-free grammar in Chomsky normal form. Every SLP $\mathbb{S}$ derives exactly one string $S \in \Sigma^+$ and we refer to S as the *text* derived from $\mathbb{S}$.

For an illustration, consider the following SLP $\mathbb{F}_6$ that derives the 6-th Fibonacci word $F_6 = abaababaabaab$:

$$\mathbb{F}_0 := a,\ \mathbb{F}_1 := b,\ \mathbb{F}_2 := \mathbb{F}_1 \cdot \mathbb{F}_2,\ \mathbb{F}_3 := \mathbb{F}_3 \cdot \mathbb{F}_1,$$
$$\mathbb{F}_4 := \mathbb{F}_4 \cdot \mathbb{F}_3,\ \mathbb{F}_5 := \mathbb{F}_5 \cdot \mathbb{F}_4,\ \mathbb{F}_6 := \mathbb{F}_6 \cdot \mathbb{F}_5.$$

The parse tree of the derivation is shown in Figure 1. In this example, the SLP derives a text of length 13 and contains 7 rules. In the general case, the n-th Fibonacci word can be derived from the following SLP with $n+1$ rules:

$$\mathbb{F}_0 := b,\ \mathbb{F}_1 := a,\ \mathbb{F}_2 := \mathbb{F}_1 \cdot \mathbb{F}_0,\ \mathbb{F}_3 := \mathbb{F}_2 \cdot \mathbb{F}_1,\ \ldots,\ \mathbb{F}_n := \mathbb{F}_{n-1} \cdot \mathbb{F}_{n-2}.$$

Since the length of the n-th Fibonacci word is equal to the $(n+1)$-th Fibonacci number, i.e. the nearest integer to $\frac{\varphi^{n+1}}{\sqrt{5}}$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio, we see that the rule number of an SLP may be exponentially smaller than the length of the text derived from the SLP.

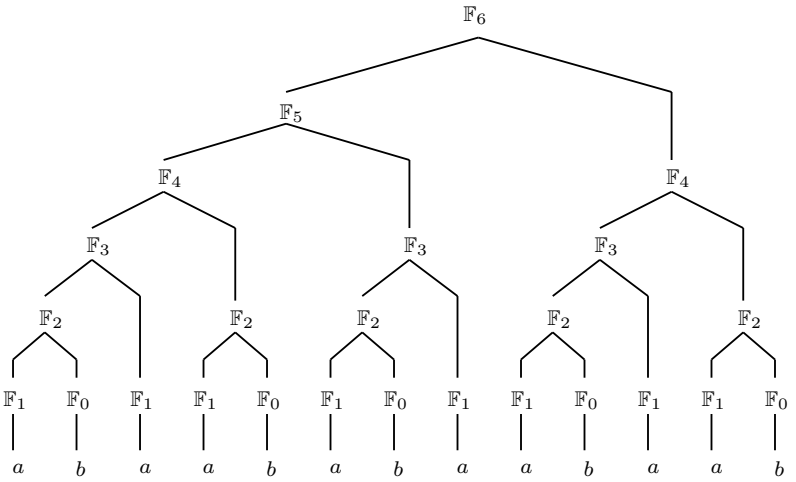

Figure 1: The parse tree of the derivation of the text *abaababaabaab* from $\mathbb{F}_6$

We adopt the following conventions in the paper: every SLP is denoted by a capital blackboard bold letter, for example, $\mathbb{S}$. The left-hand sides of the rules of this SLP are denoted by the same letter with indices, for example, $\mathbb{S}_0, \mathbb{S}_1, \ldots$. If an SLP $\mathbb{S}$ is fixed, its rules are uniquely determined by their left-hand sides and, for brevity, we allow ourselves to refer to $\mathbb{S}_0, \mathbb{S}_1, \ldots$ as rules. For each i, the rule $\mathbb{S}_i$ can be also thought of as an SLP, namely, as the SLP

$$\mathbb{S}_0 := expr_0,\ \mathbb{S}_1 := expr_1, \ldots, \mathbb{S}_i := expr_i,$$

so that one can speak of the text derived from a rule. We denote this text by the same indexed capital letter but in the standard font; for example, the text that is derived from $\mathbb{S}_i$ is denoted by S_i.

The *cut position* of a nonterminal rule $\mathbb{S}_i := \mathbb{S}_\ell \cdot \mathbb{S}_r$ is the position $|S_\ell|$ in the text S_i. For instance, the cut position of $\mathbb{F}_4$ in Figure 1 is equal to 3. For every terminal rule, we define its cut position to be equal to 0.

The *size* of an SLP $\mathbb{S}$ is the number of its rules and is denoted by $|\mathbb{S}|$. The *concatenation* of SLPs $\mathbb{S}$ and $\mathbb{S}'$ is any SLP that derives the text $S \cdot S'$. We denote the concatenation by $\mathbb{S} \cdot \mathbb{S}'$ but we would like to emphasize that, unlike string concatenation, SLP concatenation is not a rigidly defined operation as there are various ways to construct an SLP that derives $S \cdot S'$ starting from given SLPs $\mathbb{S}$ and $\mathbb{S}'$. A rather straightforward way to concatenate $\mathbb{S}$ and $\mathbb{S}'$ is as follows. Let $\mathbb{S}$ be the SLP (1) and let $\mathbb{S}'$ be the SLP

$$\mathbb{S}'_0 := expr'_0,\ \mathbb{S}'_1 := expr'_1, \ldots, \mathbb{S}'_{n'} := expr'_{n'}.$$

We set $m = n + n' + 1$ and consider the SLP $\mathbb{T}$ defined as

$$\mathbb{T}_0 := expr''_0,\ \mathbb{T}_1 := expr''_1, \ldots, \mathbb{T}_m := expr''_m,\ \mathbb{T}_{m+1} := \mathbb{T}_n \cdot \mathbb{T}_m,$$

where for each $i = 0, \ldots, n$,

$$expr''_i = \begin{cases} expr_i & \text{if } expr_i \text{ is a letter from } \Sigma, \\ \mathbb{T}_\ell \cdot \mathbb{T}_r & \text{if } expr_i = \mathbb{S}_\ell \cdot \mathbb{S}_r, \end{cases}$$

and for each $j = n+1, \ldots, m$,

$$expr''_j = \begin{cases} expr'_{j-n-1} & \text{if } expr'_{j-n-1} \text{ is a letter from } \Sigma, \\ \mathbb{T}_\ell \cdot \mathbb{T}_r & \text{if } expr'_{j-n-1} = \mathbb{S}'_\ell \cdot \mathbb{S}'_r. \end{cases}$$

With this straightforward construction, the size of the concatenation of $\mathbb{S}$ and $\mathbb{S}'$ is $|\mathbb{S}| + |\mathbb{S}'| + 1$. Of course, in some special cases one can concatenate CSPs in a much more economic way. For instance, the concatenation $\mathbb{S} \cdot \mathbb{S}$ can be obtained by adding just one extra rule: if $\mathbb{S}$ is the SLP (1), then this new rule is $\mathbb{S}_{n+1} := \mathbb{S}_n \cdot \mathbb{S}_n$. More generally, the concatenation $\underbrace{\mathbb{S} \cdot \mathbb{S} \cdot \ldots \cdot \mathbb{S}}_{k \text{ times}}$ (that we will denote by $\mathbb{S}^k$) can be constructed by adding $\lceil \log k \rceil$ additional rules.

3 Basic operations

Manipulating with SLPs is based on certain "elementary" operations. In this section we list the algorithms for basic operations that we frequently use in the present paper and discuss the space and time complexity of these algorithms. Except the pattern matching algorithm that is taken from [10], the algorithms are folklore and it is hard to provide adequate references for them. Therefore, for the reader's convenience, we present them here in some detail even though we provide no formal correctness proofs.

First we make a general observation: given an SLP $\mathbb{S}$, it is easy to calculate the number $|S|$. Indeed, we can convert the rules of $\mathbb{S}$ into a system of numerical equalities substituting each terminal rule $\mathbb{S}_i := a$, where $a \in \Sigma$, by the equality $|S_i| = 1$ and each nonterminal rule $\mathbb{S}_i := \mathbb{S}_\ell \cdot \mathbb{S}_r$, where $\ell, r < i$, by the equality $|S_i| = |S_\ell| + |S_r|$. Clearly, the resulting system of equalities constitutes a recursion that allows one to calculate $|S|$ (and $|S_i|$ for each i) via $O(|\mathbb{S}|)$ additions. Therefore, in the algorithms below, we may and will assume that $|S|$ is known whenever $\mathbb{S}$ is given. The argument also implies the inequality $\log |S| \le |\mathbb{S}|$ which will be used without reference in several complexity considerations below.

3.1 Subgrammar cutting

An operation that is most frequently invoked in this paper is the one that constructs an SLP presentation for a substring of a text presented by a given SLP. We call this operation *subgrammar cutting* even though we should emphasize that in general an SLP for a substring need not be a subgrammar (in any common sense of the word) of the initial SLP.

Here is a formal description of the subgrammar cutting problem:

PROBLEM: **SubCut**
INPUT: an SLP $\mathbb{S}$, integers ℓ and r such that $0 \le \ell < r \le |S|$;
OUTPUT: an SLP that derives the text $S[\ell \dots r]$.

We denote the output of **SubCut** by $\mathbb{S}[\ell \dots r]$. Our algorithm for **SubCut** consists of three phases. In the first phase we try to locate the least node of the parse tree of the text S with the property that the text $S[\ell' \dots r']$ derived from the rule labelling the node contains the substring $S[\ell \dots r]$, that is, $\ell' \le \ell$ and $r \le r'$. The algorithm uses three variables: a symbol $\mathbb{C}$ for the current node label and integers ℓ and r.

DESCENT: The algorithm starts from the root of the parse tree of S and we initialize $\mathbb{C}$ with the last rule of $\mathbb{S}$ and the variables ℓ and r with the input values of **SubCut**. If $\mathbb{C}$ is a terminal rule then the algorithm stops and returns $\mathbb{C}$ for $\mathbb{S}[\ell \dots r]$. Otherwise the current node is labelled a nonterminal rule $\mathbb{C} := \mathbb{L}\cdot\mathbb{R}$. If $r \le |L|$, then the algorithm descends to node labelled by $\mathbb{L}$; this means that we update $\mathbb{C}$ with $\mathbb{L}$ and keep the values of ℓ and r. If $\ell \ge |L|$, then the algorithm descends to node labelled by $\mathbb{R}$; this means that we update $\mathbb{C}$ with $\mathbb{R}$ and set $\ell := \ell - |L|$, $r := r - |L|$. If $\ell < |L| < r$, the desired node has been found, and the algorithm passes the current values $\mathbb{C}$, ℓ, and r to the next phase.

In the second phase, we work with the nonterminal rule $\mathbb{C} := \mathbb{L}\cdot\mathbb{R}$ such that $\ell < |L| < r$ and aim to decompose the rules $\mathbb{L}$ and $\mathbb{R}$ into smaller "pieces" whose concatenations are SLPs that derive the substrings $S[\ell \dots |L|]$ and respectively $S[|L| \dots r]$. We present the decomposition algorithm for $\mathbb{L}$ only since $\mathbb{R}$ can be handled in a symmetric way. The algorithm operates with a symbol $\mathbb{CL}$ for the current node label and uses a stack for storing factors of the decomposition.

LEFT DECOMPOSITION: We initialize $\mathbb{CL}$ with $\mathbb{L}$. If $\mathbb{CL}$ is a terminal rule then the algorithm adds $\mathbb{CL}$ to the result stack and stops. Otherwise $\mathbb{CL} := \mathbb{LL}\cdot\mathbb{LR}$ is a nonterminal rule. If $\ell > |LL|$, then the algorithm descends to the node

labelled by $\mathbb{LR}$, that is, updates $\mathbb{CL}$ with $\mathbb{LR}$. The stack remains unchanged. If $\ell = |LL|$ then the algorithm adds $\mathbb{LR}$ to the stack and stops. If $\ell < |LL|$, then the algorithm adds $\mathbb{LR}$ to the stack and descends to the node labelled by $\mathbb{LL}$, that is, updates $\mathbb{CL}$ with $\mathbb{LL}$. When the algorithm stops, we have a nonempty stack of rules that is passed to the final phase.

Concatenation: We concatenate the SLPs from the stack produced by the Left Decomposition in the top-to-bottom order (that is, the top element of the stack becomes the leftmost factor and so on). The concatenation produces an SLP $\mathbb{S}[\ell \dots |L|]$ that derives $S[\ell \dots |L|]$. Dually, we concatenate the SLPs from the stack produced by the Right Decomposition in the bottom-to-top order (the top element of the stack becomes the rightmost factor and so on). This produces an SLP $\mathbb{S}[|L| \dots r]$ that derives $S[|L| \dots r]$. Finally, we concatenate $\mathbb{S}[\ell \dots |L|]$ with $\mathbb{S}[|L| \dots r]$ to produce the desired SLP $\mathbb{S}[\ell \dots r]$.

Complexity: The descent phase uses $O(|\mathbb{S}|)$ time because the number of its steps does not exceed the length of the longest path between the root of the parse tree of S and some leaf of this tree. By the same reason, the decomposition phases spend $O(|\mathbb{S}|)$ time and $O(|\mathbb{S}|)$ space for stacks. The concatenation phase uses $O(|\mathbb{S}|)$ time and $O(|\mathbb{S}|)$ space to concatenate the content of the stacks. Altogether the above algorithm solves **SubCut** in $O(|\mathbb{S}|)$ time and $O(|\mathbb{S}|)$ space.

3.2 Pattern matching

Suppose that we are given two SLPs $\mathbb{S}$ and $\mathbb{T}$ and we want to find all occurrences of the text S derived from $\mathbb{S}$ as a substring of the text T derived from $\mathbb{T}$. (Clearly, we may and will assume that $|S| \leq |T|$.) A difficulty here is that in general the number of occurrences of S in T may be exponential as a function of $|\mathbb{T}|$. Thus, any polynomial algorithm for pattern matching with SLPs as input should store information about the occurrences in a suitable compressed form. It turns out that a suitable way to encode the occurrences of S in T is by arithmetic progressions. More precisely, it can be shown that the start positions of all occurrences of S in T can be grouped into $O(|\mathbb{T}|)$ arithmetic progressions. Each such arithmetic progression is completely characterized by 3 numbers: the first start position s, the difference d, and the length n, and we will denote the progression by the triple $\langle s, d, n\rangle$. For example, a progression denoted $\langle 3, 2, 4\rangle$ indicates that T contains 4 occurrences of S that start from the positions 3, 5, 7, and 9.

Here is a formal description of the pattern matching problem:

Problem: **PM**
Input: SLPs $\mathbb{S}$ and $\mathbb{T}$ such that $|S| \leq |T|$;
Output: $O(|\mathbb{T}|)$ arithmetic progressions that describe the start positions of all occurrences of S in T if S occurs in T as a substring; the empty set otherwise.

For an illustration, consider **PM** for the Fibonacci SLPs $\mathbb{F}_2$ and $\mathbb{F}_6$. Then one of the possible outputs consists of the two progressions $\langle 0, 3, 2\rangle$ and $\langle 5, 3, 3\rangle$.

We need the following result from [10]:

Theorem 3.1. *There exists an algorithm that solves **PM** using $O(|\mathbb{T}|^2|\mathbb{S}|)$ time and $O(|\mathbb{T}||\mathbb{S}|)$ space.*

In order to bound the number of arithmetic progressions in terms of the ratio $|T|/|S|$, the following consequence of the Fine–Wilf theorem is useful. We say that a string S *occurs at a position* t in T if $S = T[t \dots t + |S|]$, that is, t is the start position of an occurrence of S in T.

Lemma 3.2 ([1], Lemma 4.8)**.** *Let $p_1 < p_2 < \cdots < p_k$ be a sequence of positions of a text T such that a string S occurs at each of these positions but at no other position preceding p_k. If $p_k - p_1 \le \frac{|S|}{2}$, then the p_i's form an arithmetic progression with the difference $p = p_2 - p_1$ and the string S is p-periodic with p being its least period.*

Lemma 3.2 implies that if we divide T into $2\lceil \frac{|T|}{|S|} \rceil$ consecutive substrings of length $\le \lfloor \frac{|S|}{2} \rfloor$ and for each such substring consider start positions of occurrences of S in T the substring touches, then these positions can be grouped into a single arithmetic progression. Therefore all start positions of occurrences of S in T can be described by at most $2\lceil \frac{|T|}{|S|} \rceil$ arithmetic progressions.

3.3 Substring extending

Many algorithms detecting squares in a string S start with detecting a pair of equal substrings in S. In order to check if such a pair indeed corresponds to a square is S, we should be able to recognize whether or not the substrings forming the pair can be extended to equal substrings which are adjacent in S. This leads to the following problem.

PROBLEM: **SubsExt**
INPUT: an SLP $\mathbb{S}$, integers ℓ_1, r_1, ℓ_2, r_2 with $0 \le \ell_1 < r_1 \le |S|$, $0 \le \ell_2 < r_2 \le |S|$ such that $S[\ell_1 \dots r_1] = S[\ell_2 \dots r_2]$;
OUTPUT: integers ℓ_{ex} and r_{ex} such that ℓ_{ex} is the length of the longest common suffix of the substrings $S[0 \dots \ell_1]$ and $S[0 \dots \ell_2]$ and r_{ex} is the length of the longest common prefix of the substrings of $S[r_1 \dots |S|]$ and $S[r_2 \dots |S|]$.

We describe an algorithm that finds r_{ex}; clearly, ℓ_{ex} can be found in a symmetric way. The algorithm uses two integer variables: r_{ex} and s.

INITIALIZATION: We set $r_{ex} := 0$ and $s := \min\{|S| - r_1, |S| - r_2\}$.

MAIN LOOP: If $s = 0$, we stop and return the current value of r_{ex}. While $s > 0$, we repeat the following. Using **SubCut**, we construct the SLPs $\mathbb{S}[r_1 + r_{ex} \dots r_1 + r_{ex} + s]$ and $\mathbb{S}[r_2 + r_{ex} \dots r_2 + r_{ex} + s]$ and invoke **PM** with these SLPs as input. If the output is not empty, then one of the substrings $S[r_1 + r_{ex} \dots r_1 + r_{ex} + s]$ and $S[r_2 + r_{ex} \dots r_2 + r_{ex} + s]$ occurs as in the other one. This means that the two substrings are equal because they are of the same length s. In this case we update the variables by setting $r_{ex} := r_{ex} + s$ and $s := \min\{|S| - r_1 - r_{ex}, |S| - r_2 - r_{ex}, \lceil \frac{s}{2} \rceil\}$. Otherwise we keep the value of r_{ex} and set $s := s - \lceil \frac{s}{2} \rceil$.

COMPLEXITY: Since the value of s does not exceed $|S|$ at the initialization phase and is at least halved at each repetition of the main loop, there are $O(\log |S|)$ steps. At each step the algorithm invokes **SubCut** twice and **PM** once. Totally it needs $O(|\mathbb{S}|^3)$ time and $O(|\mathbb{S}|^2)$ space for each step. Altogether

the presented algorithm solves **SubsExt** using $O(|\mathbb{S}|^3 \log |S|)$ time and $O(|\mathbb{S}|^2)$ space.

Using **SubsExt** we can easily solve the following problem:

PROBLEM: **Period termination**
INPUT: an SLP $\mathbb{S}$, a positive integer p, integers ℓ, r with $0 \le \ell < r \le |S|$ such that $S[\ell \dots r]$ is a p-periodic substring;
OUTPUT: integers t_L, t_R such that $0 \le t_L \le \ell$, $r \le t_R \le |S|$ and $S[t_L \dots t_R]$ is a maximal p-periodic substring.

We proceed as follows. First, using **SubCut**, we construct an SLP $\mathbb{P}$ that derives $S[\ell \dots \ell + p]$. Then we let k be the least odd integer such that $p^k > |S|$; clearly, k is of order $O(\log |S|)$. In $O(\log k)$ time we construct an SLP $\mathbb{P}^k$ that derives the string $S[\ell \dots \ell + p]^k$. Finally, let $\mathbb{T} = \mathbb{S} \cdot \mathbb{P}^k$, $\ell_1 = \ell$, $r_1 = r$, $\ell_2 = |S| + p^{\frac{k-1}{2}}$, $r_2 = |S| + p^{\frac{k-1}{2}} + (r - \ell)$. The condition that $S[\ell \dots r]$ is p-periodic then ensures that the substrings $T[\ell_1 \dots r_1] = S[\ell \dots r]$ and $T[\ell_2 \dots r_2]$ are equal. Thus, we can apply our algorithm for **SubsExt** to the SLP $\mathbb{T}$ with the parameters ℓ_1, r_1, ℓ_2, r_2. If ℓ_{ex} and r_{ex} are the output integers for **SubsExt**, we get $t_L = \ell - \ell_{ex}$ and $t_R = r + r_{ex}$.

COMPLEXITY: The algorithm invokes **SubCut** once. Next it spends $\log k$, that is $O(\log \log |S|)$ time to construct the SLP $\mathbb{P}^k$. Finally, it invokes **SubsExt** once. Altogether this algorithm solves **Period termination** using $O(|\mathbb{S}|^3 \log |S|)$ time and $O(|\mathbb{S}|^2)$ space.

4 The algorithm

4.1 Basic strategy

Our algorithm closely follows the logic of [1] where an efficient solution to the problem of finding all squares in a (non-compressed) string has been proposed. (Below we reproduce the key lemmas from [1] for the reader's convenience.) Our contribution is, roughly speaking, twofold. First, we show that the approach from [1] can be implemented on a SLP representing a string in time polynomial of the size of the SLP. Second, we provide a compressed representation for the set of all squares contained in the string. This compressed representation is based on grouping the squares according to two integer parameters i and j that are defined as follows.

Let $\mathbb{S}$ be an SLP. Suppose that xx is a square that occurs in the text S. It is easy to see there is a unique rule $\mathbb{S}_j$ such that the square xx occurs in the text S_j and xx touches the cut position of $\mathbb{S}_j$. This defines the parameter j whose range is therefore the set $\{0, 1, 2, \dots, |\mathbb{S}| - 1\}$. The parameter i is defined as the only integer such that $2^{i-1} \le |x| < 2^i$. The range of this parameter is the set $\{1, 2, \dots, \lfloor \log |S| \rfloor\}$.

We introduce a rectangular $\lfloor \log |S| \rfloor \times |\mathbb{S}|$-table $T(\mathbb{S})$ and store a compressed representation of the group of squares xx that satisfy $2^{i-1} \le |x| < 2^i$, occur in S_j and touch the cut position of $\mathbb{S}_j$ in the cell $T(i, j)$ in the i-th row and the j-th column of this table. Of course, for some i and j, squares with the above properties may not exist; in this case we write $\varnothing$ in the cell $T(i, j)$. Now we

are in a position to precisely describe the form of **Computing All Squares** (**CAS**) solved by our algorithm.

PROBLEM: **CAS**
INPUT: an SLP $\mathbb{S}$;
OUTPUT: a $\lfloor \log |S| \rfloor \times |\mathbb{S}|$-table $T(\mathbb{S})$ such that for each i and j, the cell $T(i,j)$ of $T(\mathbb{S})$ contains either a compressed representation of all squares xx that satisfy $2^{i-1} \le |x| < 2^i$, occur in S_j and touch the cut position of $\mathbb{S}_j$ or $\varnothing$ if no square with the above properties exists.

We have not yet specified what kind of compressed representations is used for non-empty families of squares in cells of $T(\mathbb{S})$. In fact, we use compressed representations of three different forms and the choice of the form depends on several conditions. We will formulate these conditions and describe the corresponding representations in the course of the explanation of our algorithm.

The fact that the output data are structured in a table form may suggest that a sort of dynamic programming is employed to fill out the cells of $T(\mathbb{S})$, that is, the content of $T(i,j)$ is somehow determined by the contents of the cells $T(i',j')$ where $i' < i$ and/or $j' < j$. It is not the case, and our algorithms fills out each cell of $T(\mathbb{S})$ independently of the contents of other cells. On the one hand, this can be seen as a disadvantage as quite similar calculations are to be repeated many times; on the other hand, this opens prospects for efficient parallelization.

4.2 Local search tactic

Now we assume that an index j and a positive integer i are fixed and explain how we search for squares xx to be represented in the cell $T(i,j)$. We may additionally assume that $i > 1$. Indeed, if $i = 1$, then the inequalities $2^{i-1} \le |x| < 2^i$ imply that $|x| = 1$, that is, x is a letter from Σ. To locate squares of the form aa where $a \in \Sigma$ in S_j, we can just invoke **PM** for the SLP $\mathbb{S}_j$ and the SLP $\mathbb{A}_0 := a,\ \mathbb{A}_1 := \mathbb{A}_0 \cdot \mathbb{A}_0$, for each a. We then store the output of **PM** (that is, $O(|\mathbb{S}_j|)$ arithmetic progressions or $\varnothing$) in the cell $T(i,j)$.

Thus, let $i > 1$. If the length $|S_j|$ of text S_j is less than 2^i, no square xx such that $2^{i-1} \le |x| < 2^i$ cannot occur in S_j and we put $\varnothing$ the cell $T(i,j)$. Therefore we assume that $|S_j| \ge 2^i$. Let γ be the cut position of $\mathbb{S}_j$. Consider the 2^{i+1}-neighborhood of γ in S_j, that is, the substring $S_j[\gamma - 2^{i+1} \dots \gamma + 2^{i+1}]$. (Of course, it may happen that $\gamma - 2^{i+1} < 0$ or $\gamma + 2^{i+1} > |S_j|$; in such a case, the borders of the neighborhood are correspondingly adjusted.) We divide the neighborhood into blocks of length $d = 2^{i-2}$ starting from γ in both directions. In the "regular" case when the neighborhood has length $2^{i+2} = 16d$, it gets divided into 16 blocks of equal length which we enumerate from left to right and denote by $B_1, \dots, B_{16}$. In particular, the blocks that touch γ are B_8 (on the left) and B_9 (on the right). In the general case, the neighborhood may be shorter so that we may get less than 16 blocks and/or the leftmost and the rightmost blocks may have length less than d. To simplify notation, we still consider 16 blocks $B_1, \dots, B_{16}$ but allow some blocks to be empty and the extreme non-empty blocks to be of length less than d.

Consider now a square xx with $2^{i-1} \le |x| < 2^i$ that occurs in S_j and touches γ. Then for some position c (referred to as the *center* of xx), we can write $xx = S_j[c-|x| \dots c+|x|]$ and $c-|x| \le \gamma \le c+|x|$. Combining the latter inequalities with the inequality $|x| < 2^i$, we obtain

$$\gamma - 4d = \gamma - 2^i < \gamma - |x| \le c \le \gamma + |x| < \gamma + 2^i = \gamma + 4d.$$

This means that if a block touches the center c of xx, the block is one of the 8 central blocks $B_4, B_5, \dots, B_{12}$. At most two blocks touch c; let B_k, where $k \in \{4, 5, \dots, 12\}$, be the leftmost of these blocks. We can write this block as $B_k = S_j[\ell \dots r]$ for some ℓ and r with $r - \ell = d$. (It is easy to express ℓ and r via γ, k, and i but we do not need to explicitly write down the corresponding expressions.) Then $\ell < c \le r$ and the block $B_{k-1} = S_j[\ell - d \dots \ell]$ occurs as a substring in $x = S_j[c-|x| \dots c]$ since

$$c - |x| \le c - 2^{i-1} \le r - 2^{i-1} = \ell + d - 2d = \ell - d.$$

The string x repeats in S_j as $x = S_j[c \dots c+|x|]$ whence a copy of the block B_{k-1} should also occur as a substring in $S_j[c \dots c+|x|]$; more precisely, B_{k-1} is equal to the substring $S_j[\ell - d + |x| \dots \ell + |x|]$ because the latter substring is just the right translate by $|x|$ positions of the substring $S_j[\ell - d \dots \ell]$. Observe that the inequalities $2^{i-1} \le |x| < 2^i$ imply that

$$\ell - d + |x| \ge \ell - d + 2d = \ell + d = r \quad \text{and} \quad \ell + |x| < \ell + 2^i = \ell + 4 = r + 3d.$$

This means that the start position of the substring $S_j[\ell - d + |x| \dots \ell + |x|]$ occurs to the right of or coincides with the start position of the block $B_{k+1} = S_j[r \dots r+d]$ while the end position of $S_j[\ell - d + |x| \dots \ell + |x|]$ occurs to the left of the end position the block $B_{k+3} = S_j[r + 2d \dots r + 3d]$. We conclude that a copy of the block B_{k-1} should occur in the concatenation $B_{k+1} \cdot B_{k+2} \cdot B_{k+3}$, see Figure 2. Let us register this conclusion in the following statement.

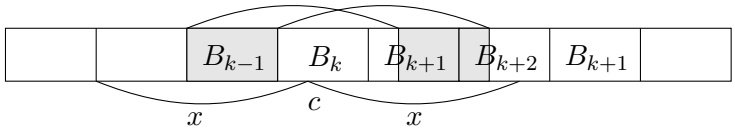

Figure 2: A copy of B_{k-1} in $B_{k+1} \cdot B_{k+2} \cdot B_{k+3}$

Lemma 4.1. *Suppose that for $i > 1$ and j, a square xx with $2^{i-1} \le |x| < 2^i$ occurs in the text S_j and touches the cut position γ of the rule $\mathbb{S}_j$. If the 2^{i+1}-neighborhood of γ in S_j is divided into blocks $B_1, \dots, B_{16}$ as described above, then there exists $k \in \{4, 5, \dots, 12\}$ such that the block B_{k-1} occurs as a substring in the concatenation $B_{k+1} \cdot B_{k+2} \cdot B_{k+3}$.*

It is Lemma 4.1 that underlies the tactic of our algorithm. We proceed as follows. For each $k \in \{4, 5, \dots, 12\}$, we invoke **SubCut** to extract from the SLP $\mathbb{S}_j$ an SLP $\mathbb{B}$ that derives the block B_{k-1} and an SLP $\mathbb{C}$ that derives the concatenation $B_{k+1} \cdot B_{k+2} \cdot B_{k+3}$. Then we run **PM** on the SLPs $\mathbb{B}$ and $\mathbb{C}$.

If **PM** returns the empty set, Lemma 4.1 ensures that no square xx satisfying its conditions and having the center within the block B_k may exist. Then we update the value of k and proceed with the next block in the role of B_{k-1}. Otherwise **PM** returns a bunch of arithmetic progressions that describe the start positions of all occurrences of B_{k-1} in $B_{k+1} \cdot B_{k+2} \cdot B_{k+3}$. Observe that since the length of $B_{k+1} \cdot B_{k+2} \cdot B_{k+3}$ does not exceed $3|B_{k-1}|$, Lemma 3.2 implies that the start positions can be grouped into at most six progressions (see the argument at the end of Subsection 3.2). Now for each of these progressions, we check (using **Period termination**) whether or not the block B_{k-1} and its occurrences in $B_{k+1} \cdot B_{k+2} \cdot B_{k+3}$ corresponding to the chosen progression can be extended to a square.

4.3 Checking square-freeness

Algorithm: For every block B_{k-1} the algorithm invokes **SubCut** two times with parameters B_{k-1} and $B_{k+1} \cdot B_{k+2} \cdot B_{k+3}$. Next the algorithm invokes **PM** with parameters $\mathbb{B}_{k-1}$. From Lemma 3.2 it follows that the occurrences can be represented using at most six arithmetic progressions. So the algorithm compress the occurrences into six arithmetic progressions. Next it verifies whether on not the block B_{k-1} and a progression $\langle a, p, t\rangle$ of its occurrences form any square. Let us consider the following cases:

- If $t = 0$ then there are no squares of expected length that touch γ and fully contain B_{k-1}. The algorithm moves to the next block;
- If $t = 1$ then the algorithm obtains ℓ_{ex}, r_{ex} using **SubsExt** for B_{k-1} and $S_j[a \dots a + 2^{i-2}]$. If $\ell_{ex} + r_{ex} > a - (k-1) \cdot 2^{i-2}$ then there exists at least one square and the algorithm returns **false**. Otherwise there are no squares of expected length that touch γ and fully contain B_{k-1}. The algorithm moves to the next block;
- If $t \geq 2$ there exists at least one square. The algorithm returns **false**.

Complexity: For each of the eight central blocks the algorithm invokes **SubCut** two times, **PM** at once and **SubsExt** at most six times. So the main step required $O(|\mathbb{S}|^3 \cdot \log|S|)$ time and $O(|\mathbb{S}|^2)$ space. The algorithm contains at most $|\mathbb{S}| \cdot \log|S|$ steps. Altogether we get the following theorem:

Theorem 4.2. *There is an algorithm that solves square-freeness problem using $O(|\mathbb{S}|^4 \cdot \log^2|S|)$ time and $O(|\mathbb{S}|^2)$ space.*

4.4 Computing all squares algorithm

In this section we present an algorithm that fills out a table $T(\mathbb{S})$. Remind the main problem:

Problem: **Computing all squares**
Input: an SLP $\mathbb{S}$ that derives a text S;
Output: a table $T(\mathbb{S})$.

ALGORITHM: It remains to recognize all squares between a block B_k and the arithmetic progression $\langle a, p, t\rangle$ of its occurrences. Since t can be exponentially large relative to $|\mathbb{S}|$ there is no polynomial algorithm that can consecutively check every occurrence of B_k.

Let α_L, α_R be output of a call of **Period termination** with the following parameters: $\mathbb{S}_j, p, (k-1)\cdot 2^{i-2}, k\cdot 2^{i-2}-1$. α_L, α_R called *defined* if they satisfies the following inequalities: $(2k-1)\cdot 2^{i-2} - (a+p\cdot t) \leq \alpha_L$, $\alpha_R < a + 2^{i-2}$. Otherwise they are called *undefined*. Since $2^i - 1$ is the greatest length of a root, start positions of squares can not be further right than $(2k-1)\cdot 2^{i-2} - (a+p\cdot t)$. In other words it does not matter where the p-periodicity terminates outside. Analogously let γ_L, γ_R be output of a call of **Period termination** with the following parameters: $\mathbb{S}_j, p, a, a+p\cdot t$. γ_L, γ_R called *defined* if they satisfies the following inequalities: $(k-1)2^{i-2} \leq \gamma_L$ and $\gamma_R < 2(a+p\cdot t) - (k-1)2^{i-2}$. Otherwise they are called *undefined*.

The following lemmas present useful relations between $\alpha_L, \alpha_R, \gamma_L$ and γ_R.

Lemma 4.3 ([1]). *If one of α_R or γ_L is defined, then the other one is defined, and $\alpha_R - \gamma_L \leq p$.*

Lemma 4.4 ([1]). *If both α_R and γ_L are undefined, then none of the squares possible containing B_k are pure squares.*

There are two main cases:
Case 1: both $\boldsymbol{\alpha_R}$ and $\boldsymbol{\gamma_L}$ are defined. Let us consider possible relative positions of α_R and γ_L:

- If $\alpha_R \geq \gamma_L$ then centers of squares may be located at $[k\cdot 2^{i-2}, \gamma_L]$, $(\gamma_L, \alpha_R]$, $(\alpha_R, (k+1)\cdot 2^{i-2})$.
- If $\alpha_R < \gamma_L$ then centers of squares may be located at $[k\cdot 2^{i-2}, \alpha_R]$, $(\alpha_R, \gamma_L]$, $(\gamma_L, (k+1)\cdot 2^{i-2})$;

Lemma 4.5 ([1]). *If both α_R, γ_L are defined then:*

1. *Squares that are contain B_k and centered at positions h, such that $h \leq \gamma_L$, may exist only if α_L is defined. These squares constitute a family of squares that corresponds to the difference $|x| = a + t'\cdot p - (k-1)2^{i-2}$, provided that there exists some $t' \in \{0\ldots t\}$ such that $\gamma_L - \alpha_L = a + t'\cdot p - (k-1)2^{i-2}$.*
2. *Squares that are contain B_k and centered at positions h, such that $\alpha_R < h$, may exist only if γ_R is defined. These squares constitute a family of squares that corresponds to the difference $|x| = a + t''\cdot p - (k-1)2^{i-2}$, provided that there exists some $t'' \in \{0\ldots t\}$ such that $\gamma_L - \alpha_L = a + t''\cdot p - (k-1)2^{i-2}$.*

Notice that if $\alpha_R < \gamma_L$, then squares whose center h satisfies $\alpha_R < h \leq \gamma_L$ may exist only if both α_L and γ_R are defined and $\gamma_R - \alpha_R = \gamma_L - \alpha_L$.

Using Lemma 4.5 the algorithm finds simple families of squares and stores them in the following compressed way: $\{|x|, c_l, c_r\}$ where $|x|$ is length of the root, c_l is the center of the leftmost square, c_r is the center of the rightmost square.

In the case $\alpha_R < \gamma_L$ the algorithm additionally calls **SubsExt** to guarantee that every family consists of squares. Let ℓ_{ex}, r_{ex} be output of a call **SubsExt** with the following parameters: $\mathbb{S}_j, \alpha_L, \alpha_R, \gamma_L, \gamma_R$. If $\alpha_R + r_{ex} < \gamma_L - \ell_{ex}$ then there are no families of squares. Otherwise the algorithm intersects sets of centers obtained using Lemma 4.5 with $[\gamma_L - \ell_{ex}, \alpha_R + r_{ex}]$.

If $\alpha_R \leq \gamma_L$ then the algorithm may find at most three simple families of squares: $\{\gamma_L - \alpha_L, \max\{k \cdot 2^{i-2}, \gamma_L - \ell_{ex}\}, \alpha_R\}, \{\gamma_R - \alpha_R, \min\{\alpha_R + 1, \gamma_L - \ell_{ex}\}, \max\{\gamma_L, \alpha_R + r_{ex}\}\}, \{\gamma_R - \alpha_R, \gamma_L + 1, \min\{a, \alpha_R + r_{ex}, (k+1) \cdot 2^{i-2} - 1\}\}$. Otherwise there are no simple families of squares. Consider the following case $\alpha_R > \gamma_L$, $|x| = \gamma_L - \alpha_L$. For an arbitrary but fixed $c \leq \gamma_L$ we have $S[\alpha_L + 1 \dots \alpha_R - 1]$ should be equal to $S[\gamma_L + 1 \dots c + \gamma_L - \alpha_L - 1]$. But $S[\alpha_L + 1 \dots \alpha_R - 1]$ is p-periodic substring and $S[\gamma_L + 1 \dots c + \gamma_L - \alpha_L - 1]$ is not p-periodic substring since it contains α_R position. Notice that every simple family of squares is unique and can not be obtained at another steps of the algorithm.

For example, let S_j be equal to $c^4 \cdot (ab)^5 \cdot c^4 \cdot (ab)^5 \cdot c^4$ and the algorithm processes $B_7 = cc$. It finds all occurrences of $B_6 = ab$ in $B_8 \cdot B_9 \cdot B_{10} = ccabab$ that form the single arithmetic progression $\langle a, p, t\rangle = \langle 18, 2, 2\rangle$. Next the algorithm calculates $\alpha_L = 4, \alpha_R = 14, \gamma_L = 18$ and $\gamma_R = 28$. Since $\gamma_R < \gamma_L$ it calculates $\ell_{ex} = 4$ and $r_{ex} = 4$. Since the algorithm looks for squares that centered at B_7 it finds the single family of squares: $\{12, 14, 15\}$. In other words it finds two squares of length 12 $c^2 \cdot (ab)^5 \cdot c^2 \cdot c^2 \cdot (ab)^5 \cdot c^2$ and $c \cdot (ab)^5 \cdot c^3 \cdot c \cdot (ab)^5 \cdot c^3$ that centered at positions 14 and 15 correspondingly.

Lemma 4.6 ([1]). *If α_R, γ_L are defined and $\gamma_L < \alpha_R$, then there might be a family of squares associated with each of the differences $|x| = a + p \cdot t' - (k-1) \cdot 2^{i-2}$ where $t' \in \{0 \dots t\}$, with centers at positions h, such that $\gamma_L < h \leq \alpha_R$. The squares in each such family are all pure squares, and they are centered at positions h, such that $\max(\alpha_L + |x|, \gamma_L) < h \leq \min(\alpha_R, \gamma_R - |x|)$. Notice that such a family is not empty only if $|x| < \min(\alpha_R - \alpha_L, \gamma_R - \gamma_L)$.*

Using Lemma 4.6 the algorithm finds at most one dynamic family of pure squares and stores it in the following compressed way: $\{k, \langle a, p, t\rangle, \alpha_L, \alpha_R, \gamma_L, \gamma_R\}$. Notice that every dynamic family of pure squares is unique and can not be obtained at another steps of the algorithm.

For example, let S_j be equal to $c^{12} \cdot (abba)^4 \cdot (bbaa)^3 \cdot bba \cdot c^{21}$ and the algorithm processes $B_7 = abba$. It finds all occurrences of $B_6 = abba$ in $B_8 \cdot B_9 \cdot B_{10} = (bbaa)^3$ that form the single arithmetic progression $\langle a, p, t\rangle = \langle 30, 4, 3\rangle$. Next the algorithm calculates $\alpha_L = 12, \alpha_R = 28, \gamma_L = 27$ and $\gamma_R = 43$. It finds the dynamic family of pure squares $\{6, \langle 30, 4, 3\rangle, 12, 28, 27, 43\}$ that represents single pure squares centered at position 27: $(abba)^3 abb \cdot (abba)^3 abb$ of length 30. Notice that the pure square $(abba)^2 \cdot abb \cdot (abba)^2 \cdot abb$ of length 22 centered at position 27 was rejected since in this case we are looking for squares with

$16 \le |x| \le 31$.
Case 2: both α_R and γ_L are undefined. So $S[\alpha_L \dots \gamma_R]$ is p-periodic string and it contains squares with float centers and float lengths of roots. The algorithm gather all squares into the single dynamic family of squares that stored in the following compressed way: $\{k, p, \alpha_L, \gamma_R\}$. Notice that every dynamic family of squares is not unique and may contains squares of unexpected length. Hence the algorithm may obtain a similar family at another step and we should cleanup the $T(\mathbb{S})$ table after the algorithm fills it out.
Complexity: Let us estimate the upper bound of the main step of the algorithm. For each of the eight central blocks the algorithm calculates at most six arithmetic progressions that describes all occurrences of the block. So the algorithm invokes **SubCut** at most 16 times and **PM** at most 8 times. For each central block and each arithmetic progression the algorithm calculates $\alpha_L, \alpha_R, \gamma_L$ and γ_R positions and at most one time extends $S[\alpha_L \dots \alpha_R]$ and $S[\gamma_L \dots \gamma_R]$. So the algorithm invokes **Period termination** and **SubsExt** at most 64 times. After that it extracts families of squares and add them to $T(\mathbb{S})$ using constant time. If $T(\mathbb{S})$ contains an adding family in a current cell (or in a current column for dynamic families of squares) then the algorithm skips the family. Totally the main step required $O(|\mathbb{S}|^3 \cdot \log |S|)$ time and $O(|\mathbb{S}|^2)$ space. The algorithm has $|\mathbb{S}| \cdot \log |S|$ steps. Altogether we get the following theorem:

Theorem 4.7. *There is an algorithm that solves Computing all squares problem using $O(|\mathbb{S}|^4 \cdot \log^2 |S|)$ time and $O(|\mathbb{S}| \cdot \max(|\mathbb{S}|, \log |S|))$ space.*

4.5 Squares table performance capabilities

Remind that a $T(\mathbb{S})$ table is a rectangular table that stores at most constant number of families of squares in each cell. There are four types of families of squares:

- An empty family that stored as $\varnothing$;
- A simple family of squares that stored as $\{|x|, c_\ell, c_r\}$;
- A dynamic family of pure squares that stored as $\{k, \langle a, p, t\rangle, \alpha_L, \alpha_R, \gamma_L, \gamma_R\}$;
- A dynamic family of squares that stored as $\{k, p, \alpha_L, \gamma_R\}$;

The structure of both types of dynamic families is nontrivial. This fact restricts a number of problems that are solvable by usage of $T(\mathbb{S})$. In the same time there are common properties supported by all dynamic families: *total* (total number of squares that contains in the family), *reduction by length of root* (for a fixed value of $|x|$ reduce a dynamic family to an array of simple families), *reduction by position* (for a fixed position ℓ of S returns range of roots of squares that start from ℓ and contain in the family). Using this properties it is easy to solve the following problems:

- to find information about all squares of fixed length;
- to compute total number of squares that are contained in S;

- to find information about all squares that starts from a fixed position i;
- to find a maximal by length square that occurs in S;
- to check whether on not a text S is square-free;

In the same time it is hard to solve the following problems:

- for a given SLPs $\mathbb{S}$ and $\mathbb{P}$ such that $\mathbb{P}$ derives a square xx whether or not xx occurs in the text S (it is easy to run the pattern matching algorithm than use information from $T(\mathbb{S})$);
- for a given $\mathbb{S}$ that derives a text S to construct an SLP that derives all squares that occur in S (we belief that this problem is NP-hard);

The last two problems shows us that $T(\mathbb{S})$ accumulates quantitative information about squares rather than information appropriate for searching for some fixed squares. It is common restrictions of an algorithms over SLPs that gather some information about all objects of the specified type. We have the similar situation with **Computing all palindromes** problem [12].

It remains to show how to implement the interface of a dynamic family for both types of families. Consider an implementation of the interface for a dynamic family of pure squares.

- *total* Since $\alpha_R - \gamma_L < p$ there is at most one $t_c \in \{0 \dots t\}$ such that $\alpha_L + |x_c| > \gamma_L$ or/and $\alpha_R \geq \gamma_R - |x_c|$ where $|x_c| = a + p \cdot t_c - (k-1) \cdot 2^{i-2}$. If there is no such t_c then the implementation returns $t \cdot (\alpha_R - \gamma_L)$. Otherwise it computes value of $|x_c|$ that corresponds to t_c and returns $(t_c - 1) \cdot (\alpha_R - \gamma_L) + (\min(\alpha_R, \gamma_R - |x_c|) - \max(\alpha_L + |x|, \gamma_L))$. If the length of a root is equal to $a + p \cdot (t_c + 1) - (k-1) \cdot 2^{i-2}$ then the family contains no squares.
- *reduction by length of root* For a fixed length of a root $|x|$ the implementation returns the following family: $\{|x|, p, \max(\alpha_L + |x|, \gamma_L), \min(\alpha_R, \gamma_R - |x_c|)\}$.
- *reduction by position* Since $\alpha_R - \gamma_L < p$ there is at most one square that belong to the family and starts from ℓ. The implementation returns $\varnothing$ or the length of the root of the square.

Consider an implementation of the interface for a dynamic family of squares. Let $\{k, p, \alpha_L, \gamma_R\}$ be a a dynamic family of squares. For a particular position c we can represent $S[\alpha_L \dots \alpha_R]$ as $S[x \dots c+p] \cdot S[c \dots c+p]^d \cdot S[c \dots y]$. What kind of squares from the family centered at c? There are the following squares: $S[c \dots c+p]^4, S[c \dots c+p]^6 \dots, S[c \dots c+p]^{2 \cdot \lfloor \frac{d}{2} \rfloor}$. Also there are squares that are generated by a set of overlaps of $S[c \dots c+p]$. For example, if $S[c \dots c+p] = aba$ then the set of overlaps is equal to $\{a, aba\}$ and it generates the following squares: $\{aa, abaaba\}$. Notice that all squares generated by overlaps of $S[c \dots c+p]$ are pure squares and out of our interest at this moment. Suppose

that there is another square xx that centered at c. Hence it has the following structure: $u \cdot S[c \dots c+p]^{t_\ell} \cdot S[c \dots c+p]^{t_r} \cdot v$, where u and v are suffix/prefix of $S[c \dots c+p]$ correspondingly and $c = |u| + t_\ell \cdot p$. Obviously $|x|$ is the period of $u \cdot S[c \dots c+p]^{t_\ell} \cdot S[c \dots c+p]^{t_r} \cdot v$. Since p is minimal period by Fine-Wilf's theorem [5] we get that $LCD(|x|, p) = p$. If $|u| + |v| \neq p$ then we get the contradiction since $p||x|$. If $|u| + |v| = p, |u| \neq |v|$ then xx is not centered at c. Otherwise we get the contradiction with the condition that p is minimal.

Now we know all about structure of squares from the family that are centered at a particular position. But we get the following problem: how to aggregate information about squares since the family may contains exponentially many positions? The intuition behind this problem is simple: there are constant number of restrictions: the borders α_L, γ_R and the cut position γ, this restrictions generates a constant number of segments with predictable growth of squares (a particular number of segments depends on relative position of $\alpha_L, \gamma_R, \gamma$). For each segment we can calculate the required property.

Let us show how to compute the count property. The positions α_L, γ_R affect on maximal valid length of square and position γ affects on minimal valid length of square. The main idea is to calculate all squares in the family excluding condition of touching γ and calculate all squares than does not touch γ.

The leftmost square centered at position $\alpha_L + 2p$ and its length equals to p^4. Analogously the rightmost square centered at position $\gamma_R - 2p$ and its length equals to p^4. There is a single square of length p^4 centered at each position from $\alpha_L + 2p, \dots, \alpha_L + 3p - 1$. Symmetrically is a single square of length p^4 centered at each position from $\gamma_R - 2p, \dots, \gamma_R - 3p + 1$. There are two squares of length p^4 and p^6 centered at each position from $\alpha_L + 3p, \dots, \alpha_L + 4p - 1$. Symmetrically are two squares of length p^4 and p^6 centered at each position from $\gamma_R - 3p, \dots, \gamma_R - 4p + 1$ and so on. Totally there are $\lfloor \frac{\gamma_R - \alpha_L}{2p} \rfloor - 2$ such steps and there are $2p \cdot \sum_{i=1}^{\lfloor \frac{\gamma_R - \alpha_L}{2p} \rfloor - 2} i$ squares centered at presented positions. Also there is a top step that contains $\gamma_R - \alpha_L - \lfloor \frac{\gamma_R - \alpha_L}{p} \rfloor \cdot p$ positions with $\lfloor \frac{\gamma_R - \alpha_L}{2p} \rfloor - 1$ squares centered at each position. So we count all squares in the family excluding condition of touching γ.

All squares from the family centered at positions $\gamma - 2p, \dots, \gamma + 2p$ touch γ. At each position from $\gamma - 2p - 1, \dots, \gamma - 3p$ and $\gamma + 2p + 1, \dots, \gamma + 3p$ there is a single square of length p^4 that does not touches γ. At each position from $\gamma - 3p - 1, \dots, \gamma - 4p$ and $\gamma + 3p + 1, \dots, \gamma + 4p$ there are two squares of length p^4 and p^6 that does not touch γ and so on. Totally we have the following sum: $p \cdot \sum_{i=1}^{\lfloor \frac{\gamma - \alpha_L}{p} - 4 \rfloor} i + (\gamma - \alpha_L - \lfloor \frac{\gamma - \alpha_L}{p} \rfloor p)(\lfloor \frac{\gamma - \alpha_L}{p} \rfloor - 3) + p \sum_{i=1}^{\lfloor \frac{\gamma_R - \gamma}{p} - 4 \rfloor} i + (\gamma_R - \gamma - \lfloor \frac{\gamma_R - \gamma}{p} \rfloor p)(\lfloor \frac{\gamma_R - \gamma}{p} \rfloor - 3)$.

But there may exist positions greater than $\alpha_L + 2p$ and less than $\gamma_R - 2p$ such that there are no squares that belongs to the family and touches γ. Let s_ℓ be the rightmost position less than γ and s_r be the leftmost position greater than γ. Formally $s_\ell = \alpha_L + \lceil \frac{\gamma - \alpha_L}{2} \rceil$ and $s_r = \gamma + \lceil \frac{\gamma_R - \gamma}{2} \rceil$. Finally we should reduce our sums by positions s_ℓ and s_r.

For example, let $\mathbb{S}$ be an SLP that derives $(aba)^{15}ab$ and $\gamma = 16$. By the formulas above we get $s_\ell = 0 + \lceil \frac{16-0}{2} \rceil = 8$ and $s_r = 16 + \lceil \frac{47-16}{2} \rceil = 32$.

Let us compute all squares that occur at S and centered at $s_\ell, \ldots, s_r$. S has the following distributions of squares: the first two p-blocks from the left and from the right located at positions $0, \ldots, 5$ and $42, \ldots, 47$ correspondingly contains no searching squares. Next S has the "stable" grows of squares during the following $\lfloor \frac{\gamma_R - \alpha_L}{2p} \rfloor - 2 = \lfloor \frac{47-0}{6} \rfloor - 2 = 5$ p-blocks from position 6. Next S has the top block of length $\gamma_R - \alpha_L - \lfloor \frac{\gamma_R - \alpha_L}{2p} \rfloor \cdot 2p + 1 = 47 - \lfloor \frac{47-0}{3} \rfloor \cdot 3 = 47 - 42 = 5$ from position $6 + 5 \cdot 3 = 21$ with 6 squares at each position. Next S has the "stable" falls of squares during the following 5 p-blocks from position $21 + 6 = 27$. To localize squares in $[s_\ell, s_r]$ we find p-blocks that contains s_ℓ and s_r. s_ℓ is contained at $\lfloor \frac{s_\ell}{p} \rfloor + 1 = 3$ p-block from the left. s_r is contained at $\lfloor \frac{\gamma_R - s_r}{p} \rfloor + 1 = 6$ p-block from the right. So we get the following sum: $\sum_+ = 1 + 3 \cdot \sum_{i=2}^{5} + 5 \cdot 6 + 3 \cdot \sum_{i=4}^{5} = 91$. In other words there are 91 squares that occur at S and centered at positions $8, \ldots, 32$.

Let us compute all squares that occur at S, centered at $s_\ell, \ldots, s_r$ and does not touch γ. The first squares that does not touch γ appears at positions $\gamma - 2p - 1 = 8$ and $\gamma + 2p + 1 = 23$. There are 2 squares of length p^4 that centered at $9, \ldots, 8$ and does not touch γ. There are $3 \cdot \sum_{i=1}^{3} i = 18$ squares that centered at $23, \ldots 32$ and does not touch γ. There are $\sum_- = 19$ squares that centered at $s_\ell, \ldots, s_r$ and does not touch γ.

Finally there are $\sum = \sum_+ - \sum_- = 91 - 19 = 82$ squares in S that touch γ.

The implementation of the remain properties are similar to the implementation of the *total* property.

5 Conclusion

We have presented an algorithm that for a given SLP $\mathbb{S}$ deriving a text S fills out a table containing information about all squares that occur in S in time $O(|\mathbb{S}|^4 \cdot \log^2 |S|)$ using $O(|\mathbb{S}| \cdot \max\{|\mathbb{S}|, \log |S|\})$ space. We would like to emphasize some features of the algorithm:

- This algorithm is divided into independent steps in contrast to classical algorithms in this area which consecutively accumulate information about required objects. As a result it can be parallelized.
- This algorithm presents a new technique for SLPs processing.
- The algorithm is quite difficult for practical implementation. It is not excluded that constants hidden in the "O" notation are actually very big.
- The present upper bound for the time complexity is rather high and is not matched by any lower bound. The question whether the upper bound can be lowered to say, cubic in $|\mathbb{S}|$ remains open.

There is an open problem: is there any better way to compress dynamic families of squares?

References

[1] A. Apostolico and D. 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] H. Bannai, T. Gagie, T. I, S. Inenaga, G. M. Landau, and M. Lewenstein. An efficient algorithm to test square-freeness of strings compressed by straight-line programs. *Inf. Process. Lett.*, 112(19):711–714, 2012.

[3] A. Bertoni, C. Choffrut, and R. Radicioni. Literal shuffle of compressed words. In *IFIP TCS*, volume 273 of *IFIP*, pages 87–100. Springer, 2008.

[4] F. Bonhomme, E. Rivals, A. Orth, G. R. Grant, A. J. Jeffreys, and P. J. Bois. Species-wide distribution of highly polymorphic minisatellite markers suggests past and present genetic exchanges among house mouse subspecies. *Genome Biology*, 5(8), 2007.

[5] N. J. Fine and H. S. Wilf. Uniqueness theorem for periodic functions. *Proc. Amer. Math. Soc.*, 16:109–114, 1965.

[6] L. Gasieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for lempel-zip encoding (extended abstract). In *SWAT*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403. Springer, 1996.

[7] L. Khvorost. Searching all pure squares in compressed texts. In *AutoMathA: from Mathematics to Applications, University of Liege, Proceedings*, pages 41–59, 2009.

[8] L. Khvorost. Searching all pure squares in compressed texts. In *RuFiDiM II. Proceedings of the 2nd Russian Finnish Symposium on Discrete Mathemtics.*, volume 17 of *TUCS Lect. Notes.*, pages 116–122. Turku, Juvenes Print, 2012.

[9] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theor. Comput. Sci.*, 1(298):253–272, 2003.

[10] Y. Lifshits. Processing compressed texts: A tractability border. In *CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2007.

[11] Y. Lifshits and M. Lohrey. Querying and embedding compressed texts. In *MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 681–692. Springer, 2006.

[12] W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Computing longest common substring and all palindromes from compressed strings. In *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 2008.

[13] W. Matsubara, S. Inenaga, and A. Shinohara. An efficient algorithm to test square-freeness of strings compressed by balanced straight line programs. *Chicago J. Theor. Comput. Sci.*, 2010, 2010.

[14] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *CPM*, volume 1264 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1997.

[15] W. Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[16] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *CPM*, volume 1645 of *Lecture Notes in Computer Science*, pages 37–49. Springer, 1999.