Computing All Pure Squares in Compressed Texts

L. Khvorost*
Ural State University
jaamal@mail.ru

Abstract

A square xx is called pure if x is a primitive string. We consider the problem of computing all pure 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 pure squares in S in a compressed form. We present an algorithm that solves the problem in $O(|\mathbb{S}|^4 \cdot \log^2 |S|)$ time and requires $O(|\mathbb{S}| \cdot \max\{|\mathbb{S}|, \log |S|\})$ space, where $|\mathbb{S}|$ (respectively $|S|$) is 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) [6, 7, 9, 11], collage-systems [10], string representations using antidictionaries [13], 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 text 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 [9].

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 author acknowledges support from the Federal Education Agency of Russia, grant 2.1.1/3537.

the size of compressed representations. As examples we mention here the problems **Hamming distance** [6] and **Literal shuffle** [2]. On the other hand, there exist problems that admit algorithms working rather well on compressed representations: **Pattern matching** [6, 8], **Longest common substring** [11], **Computing all palindromes** [11]. This dichotomy gives rise to the following research direction: to classify important string problems by their behavior with respect to compressed data.

The **Computing All Squares** (**CAS**) problem is a well-known problem on strings. It is of importance, for example, in molecular biology. (We just mention in passing that the story of origin and migration of mice on the Eurasia continent was restored thanks to information on squares in DNA sequences in mouse genome [4].) It is not yet known whether or not **CAS** admits an algorithm polynomial in the size of a compressed representation of a given text.[1] 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. So we must store information about squares in a compressed form. Also this implies that we 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. So far we are able to overcome these difficulties for pure squares only.

A string x is *primitive* if $x = u^k$ for some integer k implies that $k = 1$ and $u = x$. A *pure* square is a square xx where x is primitive. Otherwise, xx is called a *repetition*. We formulate the **Computing all pure squares** problem in terms of SLPs as follows:

PROBLEM: **CAPS**
INPUT: an SLP $\mathbb{S}$ that derives some text S;
OUTPUT: a data structure (a PS-table) that contains information about all pure squares in S in a compressed form.

Observe that restricting to pure squares by no means makes the problem easy: the difficulties mentioned above persist for pure squares. Indeed, the same string a^n has $n-1$ pure squares, thus exponentially many with respect to the size of its compressed representation.

Let us describe in more detail the content of the present paper and its structure. Section 2 gathers some preliminaries about SLPs. In Section 3 we present some basic operations over SLPs widely used in paper. In Sec-

[1] A polynomial algorithm that solves **CAS** for strings represented by Lempel-Ziv encodings was announced in [5]. This representation is slightly more general than that by SLPs. However no details of the algorithm have ever been appeared.

tion 4 we present two algorithms. The first algorithm, given an SLP $\mathbb{S}$, checks whether or not the text $\mathbb{S}$ generates is square free. (A polynomial algorithm for testing square-freeness of a compressed text has been recently developed in [12] but only for the special case of SLPs, in which the parse tree of the text is balanced.) The second algorithm, given SLP $\mathbb{S}$, fills out the corresponding PS-table. Both algorithms use time and space resources that depend polynomially on the size of $\mathbb{S}$ and logarithmically on the size of the text generated by $\mathbb{S}$. Our algorithms closely follows the approach from [1] where an efficient solution to **CAS** for non-compressed strings has been proposed and we reproduce the key lemmas from [1] for the reader's convenience. Also in Section 4 we discuss functionality of PS-tables and list several problems that can be easily solved whenever a PS-table is available. In Section 5 we summarize our results. Appendix A contains illustrative examples.

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 $S_1 \cdot S_2$. A *position* in a string S is a point between 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 text and the position $|S|$ following it. For a string S and an integer $i (0 \leq i \leq |S|)$ we define $S[i]$ as character between positions i and $i + 1$ of S. For example $S[0]$ is a first character of S. A *substring* of S starting at a position ℓ and ending at a position r, $0 \leq \ell < r \leq |S|$, 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* some position t if $\ell \leq t \leq r$. The position $|x|$ of a square xx is called the *center* of xx and x is referred to as the *root* of xx.

A *straight-line program* (SLP) $\mathbb{S}$ is a sequence of assignments of the form:

$$\mathbb{S}_1 = expr_1,\ \mathbb{S}_2 = expr_2, \ldots, \mathbb{S}_n = expr_n,$$

where $\mathbb{S}_i$ are *rules* and $expr_i$ are expressions of the form:

- $expr_i$ is a symbol of Σ (we call such rules *terminal*), or
- $expr_i = \mathbb{S}_\ell \cdot \mathbb{S}_r$ $(\ell, r < i)$ (we call such rules *nonterminal*).

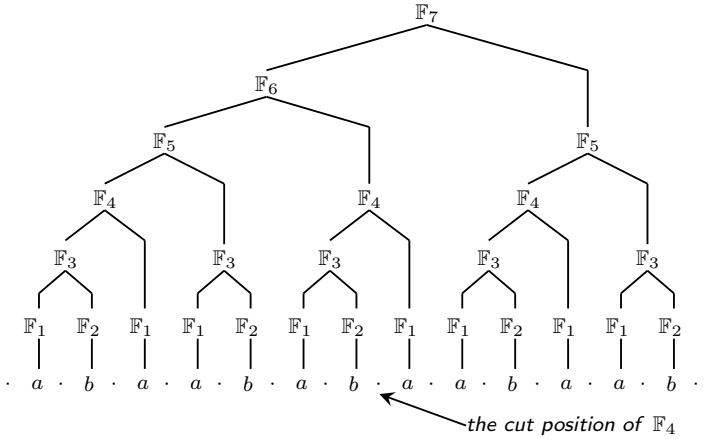

Figure 1. The parse tree of the SLP that generates $F_7 = abaababaabaab$

Thus, an SLP is a context-free grammar in Chomsky normal form. Every SLP $\mathbb{S}$ generates exactly one string $S \in \Sigma^+$ and we refer to it as the *text* generated by $\mathbb{S}$.

Example: The following SLP $\mathbb{F}_7$ generates the 7-th Fibonacci word F_7= *a b a a b a b a a b a a b*:

$$\mathbb{F}_1 = a,\ \mathbb{F}_2 = b,\ \mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2,\ \mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1,$$
$$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3,\ \mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4,\ \mathbb{F}_7 = \mathbb{F}_6 \cdot \mathbb{F}_5;$$

We accept the following conventions in the paper: every SLP is denoted by a capital blackboard bold letter, for example, $\mathbb{S}$. Every rule of this SLP is denoted by the same letter with indices, for example, $\mathbb{S}_1, \mathbb{S}_2, \ldots$. The text that is derived from a rule is denoted by the same indexed capital letter in the standard font, for example, the text that is derived from $\mathbb{S}_i$ is denoted by S_i.

The *size* of an SLP $\mathbb{S}$ is the number of its rules and is denoted by $|\mathbb{S}|$. The *concatenation* of SLPs $\mathbb{S}_1$ and $\mathbb{S}_2$ is an SLP that derives $S_1 \cdot S_2$ and denoted by $\mathbb{S}_1 \cdot \mathbb{S}_2$. 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 the example in Figure 1 is equal to 2. For every terminal rule we define its cut position to be equal to 0.

3 Basic operations

In this section we present some basic operation over SLPs widely used in the paper.

3.1 Subgrammar cutting

PROBLEM: **SubCut**
INPUT: an SLP $\mathbb{S}$ that derives text S and $0 \leq \ell < r \leq |\mathbb{S}|$ are integers;
OUTPUT: an SLP $\mathbb{S}[\ell \dots r]$ that derives text $S[\ell \dots r]$;
ALGORITHM: The algorithm descends from the root of $\mathbb{S}$ to a son that touches both positions ℓ and r while it finds a rule $\mathbb{S}'$ such that $\mathbb{S}'$ is a terminal rule or a nonterminal rule that left son touches only ℓ and right son touches only r. Notice that original values of ℓ and r may change during the process. If $\mathbb{S}'$ is a terminal rule then the algorithm returns $\mathbb{S}'$. Let $\mathbb{S}' = \mathbb{L} \cdot \mathbb{R}$ be a nonterminal rule and γ be a cut position of $\mathbb{S}'$. The algorithm decompose $\mathbb{L}$ into rules $\mathbb{L}_p, \dots, \mathbb{L}_1 \in \mathbb{L}$ such that $\mathbb{L}_p \cdot \dots \cdot \mathbb{L}_1$ derives $S[\ell \dots \gamma]$ and decompose $\mathbb{R}$ into rules $\mathbb{R}_1, \dots, \mathbb{R}_q \in \mathbb{R}$ such that $\mathbb{R}_1 \cdot \dots \cdot \mathbb{R}_q$ derives $S[\gamma \dots r]$. At Figure 2 presented pseudo code that illustrate how the algorithm obtains $\mathbb{L}_p, \dots, \mathbb{L}_1$. The set of rules $\mathbb{R}_1, \dots, \mathbb{R}_q$ obtains similarly.

```
function (𝕃, l) {
  if (ℓ == 0)
    return 𝕃;
  var result = ∅;
  var ℂ = 𝕃;    //we suppose ℂ = ℂ_l · ℂ_r
  while (l != 0) {
    if (l ≥ |ℂ_l|)
      ℂ = ℂ_r;
    else {
      result+ = ℂ_r;
      ℂ = ℂ_l;
    }
  }
  return result;
}
```

Figure 2. Pseudo code for decomposition $\mathbb{L}$ into rules $\mathbb{L}_p \cdot \dots \cdot \mathbb{L}_1$

Finally the algorithm concatenates two set of rules into a single SLP in the following order:

$$\mathbb{L}_p \cdot (\mathbb{L}_{p-1} \cdot \dots \cdot (\mathbb{L}_1 \cdot (\mathbb{R}_1 \cdot \dots \cdot (\mathbb{R}_{q-1} \cdot \mathbb{R}_q)))).$$

COMPLEXITY: To find a rule $\mathbb{S}'$ the algorithm descends along a path between the root of $\mathbb{S}$ and some leaf of $\mathbb{S}$ at worst case, so the algorithm needs $O(|\mathbb{S}|)$ time. To decompose $\mathbb{L}$ into rules $\mathbb{L}_p, \dots, \mathbb{L}_1$ the algorithm descends along a path between the root of $\mathbb{L}$ and some leaf of $\mathbb{L}$ at worst case, so the algorithm needs $O(|\mathbb{L}|)$ time. Similarly the algorithm needs $O(|\mathbb{R}|)$ time to decompose

$\mathbb{R}$ into rules $\mathbb{R}_1, \ldots, \mathbb{R}_q$. Altogether there is algorithm that solves **SubCut** problem using $O(|\mathbb{S}|)$ time and $O(|\mathbb{S}|)$ space.

3.2 Pattern matching

PROBLEM: **PM**
INPUT: SLP's $\mathbb{S}, \mathbb{T}$ such that $|S| \leq |T|$;
OUTPUT: $O(|\mathbb{T}|)$ arithmetic progressions that describe start positions of all occurrences S in T;

Theorem 3.1 ([6]). *There is algorithm that solves **PM** problem using $O(|\mathbb{T}|^2|\mathbb{S}|)$ time and $O(|\mathbb{T}||\mathbb{S}|)$ space.*

3.3 Substrings extending

PROBLEM: **SubsExt**
INPUT: an SLP $\mathbb{S}$ and integers $\ell_1, r_1, \ell_2, r_2 (0 \leq \ell_1 < r_1 \leq |\mathbb{S}|, 0 \leq \ell_2 < r_2 \leq |\mathbb{S}|)$ such that $S[\ell_1 \ldots r_1] = S[\ell_2 \ldots r_2]$;
OUTPUT: integers ℓ_{ex} and r_{ex}, where ℓ_{ex} is the length of the longest common suffix of $S[1 \ldots r_1]$ and $S[1 \ldots r_2]$, r_{ex} is the length of the longest common prefix of $S[\ell_1 \ldots |S|]$ and $S[\ell_2 \ldots |S|]$;

ALGORITHM: The algorithm obtains r_{ex} by induction.

BASE: The algorithm set r_{ex} is equal to $\min\{|S| - r_1, |S| - r_2\}$. Using **SubCut** problem the algorithm constructs $\mathbb{S}[r_1 \ldots r_1 + r_{ex}]$ and $\mathbb{S}[r_2 \ldots r_2 + r_{ex}]$. Next it runs **PM** problem with the SLPs as input. If the result set is not empty then algorithm stops. Otherwise it sets $step = \lfloor \frac{r_{ex}}{2} \rfloor$ and $r_{ex} = 0$.

STEP: Using **SubCut** problem the algorithm constructs $\mathbb{S}[r_1 \ldots r_1 + r_{ex} + step]$ and $\mathbb{S}[r_2 \ldots r_2 + r_{ex} + step]$. Next it runs **PM** problem with the SLPs as input. If the result set is not empty then the algorithm sets $r_{ex} = r_{ex} + step$ and $step = step + \lceil \frac{step}{2} \rceil$. Otherwise it sets $step = step - \lceil \frac{step}{2} \rceil$.

The algorithm obtains ℓ_{ex} analogously.

COMPLEXITY: There are $O(\log |S|)$ steps. At each step the algorithm runs **SubCut** problem two times and **PM** problem one time. Totally it needs $O(|\mathbb{S}|^3)$ time and $O(|\mathbb{S}|^2)$ space for each step. Altogether there is algorithm that solves **SubsExt** problem $O(|\mathbb{S}|^3 \log |S|)$ time and $O(|\mathbb{S}|^2)$ space.

Using **SubsExt** problem we can easy solve the following problem:
PROBLEM: **Period termination**
INPUT: an SLP $\mathbb{S}$, an integer $p > 0$ and positions $0 \leq \ell < r \leq |\mathbb{S}|$ such that $S[\ell \ldots r]$ is p-periodic substring;

OUTPUT: t_L, t_R are positions in the text S where p-periodicity of $S[\ell \dots r]$ terminates from the left and from the right correspondingly;

ALGORITHM: Using **SubCut** problem the algorithm constructs SLP $\mathbb{P}$ that derives $S[\ell \dots \ell + p]$. Let k be an odd integer that $p^k > |S|$. Using $\Theta(\log k)$ time the algorithm constructs SLP $\mathbb{P}^k$ that derives $S[\ell \dots \ell + p]^k$. Finally it runs **SubsExt** problem with the following parameters: $\mathbb{S} \cdot \mathbb{P}^k$, $\ell_1 = \ell, r_1 = r, \ell_2 = |S| + p \cdot \frac{k-1}{2}, r_2 = |S| + p \cdot \frac{k-1}{2} + (r - \ell)$.

COMPLEXITY: The algorithm runs **SubCut** problem one time. Next it spends $\log k \leq \log |S|$ operations to construct $\mathbb{P}^k$. Finally in runs **SubsExt** problem one time. Altogether there is algorithm that solves **Period termination** problem using $O(|\mathbb{S}|^3 \log |S|)$ time and $O(|\mathbb{S}|^2)$ space.

3.4 Purity check

PROBLEM: **Purity check**

INPUT: an integer $p > 0$ and an SLP $\mathbb{S}$ such that $|S| \geq 2p$ and for every position $c \in \overline{p \dots |S| - p}$ $S[c - p \dots c + p]$ is a repetition;

OUTPUT: **true**, if all the repetitions are pure squares, otherwise **false**;

Lemma 3.2 ([1]). *A family of repetitions contains a pure square if and only if all the repetitions in the family are pure squares.*

If square xx is not pure then x contains at least one pure square. It follows form definition of pure square. To solve **Purity check** problem enough to check whether exists pure square that belongs to $S[\ell - |x| \dots \ell]$.

Sketch:

The algorithm starts from root of $\mathbb{S}$ and recursively goes down in parse tree of $\mathbb{S}$.

4 The algorithm

4.1 Idea of pure squares location

Let us fix a rule $\mathbb{S}_j = \mathbb{S}_l \cdot \mathbb{S}_r$ and its cut position γ. There exists a number i_0 such that $2^{i_0} \geq \max\{|S_l|, |S_r|\}$. Let us fix a number $i \in \{1, \dots, i_0\}$. So for every pair (j, i) the algorithm looks for the following set of pure squares: $\{xx \mid |x| \in [2^{i-1}, 2^i - 1]$ and xx touches $\gamma\}$.

CORRECTNESS: Let us show that the algorithm doesn't miss any pure square by contradiction. Let xx be a pure square that was not found using the search algorithm. Let ℓ, r be left and right position of xx in the text S.

Using $O(|\mathbb{S}|)$ time we can obtain a rule $\mathbb{S}_j$ that S_j fully contains xx and xx touches cut position of $\mathbb{S}_j$ (just walk down from root of $\mathbb{S}$ while not found a first rule that cut position is in $[\ell, r]$). There exist number i that $|x| \in [2^{i-1}, 2^i - 1]$. Since xx touches γ and $|xx| < 2^{i+1}$ then xx belongs to 2^{i+1}-environment of γ. So we have contradiction with condition that for fixed rule $\mathbb{S}_j$ and number i we able to found all pure squares.

4.2 Idea of local search

Let partition the 2^{i+1}-environment of γ into 16 text blocks of equal length. Thus the length of each block is 2^{i-2}. Partition starts from γ so leftmost and rightmost blocks in the partition may have length less than 2^{i-2}. Let us enumerate the blocks from B_1 to B_{16}. Notice centers of pure squares should locate at blocks from B_5 to B_{12} otherwise we get contradiction with length of pure square or touching γ.

Let c be center of a pure square xx and it belongs to one of the eight blocks B_k. Since $|x| \geq 2^{i-1}$ then xx should contain at least three consecutive blocks B_{k-1}, B_k and B_{k+1}. For example, let c belong to block B_7 then xx must contains blocks B_6 and B_8. Let us fix B_{k-1}. Using pattern matching algorithm we able to find all occurrences of B_{k-1} in $B_{k+1} \cdot B_{k+2}$. Next the algorithm take pair of blocks (B_{k-1} and its occurrence in $B_{k+1} \cdot B_{k+2}$) and extend both to check whether or not they form a square.

Correctness: Let us show correctness of the idea by contradiction. Let xx be a pure square that fully contains in S_j, touches γ and $|x| \in [2^{i-1}, 2^i - 1]$. Suppose xx was not found using the search algorithm. Let c be a center of xx. If c not belong to blocks from B_5 to B_{12} then we have contradiction as shown above. Let c belong to block B_k where $k \in \{5, \ldots, 12\}$. If two adjacent blocks of B not belong to xx then we have contradiction with length of xx. Since $|x| \geq 2^{i-1}$ then xx contains at least three consecutive blocks of the partition. Let B_{k-1} is one of two adjacent blocks. If start position of occurrence of B_{k-1} belong to B_k then $|x| < 2 \cdot 2^{i-2} < 2^{i-1}$ that contradicts with $|x| \in [2^{i-1}, 2^i - 1]$. If start position of occurrence of B_{k-1} belong to B_{k+3} (or even greater) then $|x| > 4 \cdot 2^{i-2} > 2^i - 1$ that contradicts with $|x| \in [2^{i-1}, 2^i - 1]$ also. Otherwise we have contradiction with condition that we able to find all pure squares between a block and its occurrences in some area.

4.3 Square-freeness problem

PROBLEM: **Square-freeness**
INPUT: an SLP $\mathbb{S}$ that derives text S;
OUTPUT: Whether or not S is square-free?

The following lemma is helpful for solving square-freeness problem:

Lemma 4.1 ([1]). *Assume that the period of a string B is p. If B occurs only at positions $p_1 < p_2 < \cdots < p_k$ of a text S and $p_k - p_1 \leq \frac{|B|}{2}$ then the p_i's form an arithmetic progression with difference p.*

```
for each (S_j ∈ S)
    for each (i ∈ {1, ..., i_0})
        partition text S_j into 12 blocks of length 2^{i-2};
        for each (B_k, k ∈ {5, ..., 12}) {
            var progressions = PM(B_{k-1}, B_{k+1} · B_{k+2});
            for each (⟨a, p, t⟩ ∈ progressions) {
                if (t = 2) return false;
                if (t = 1) {
                    l_ex, r_ex = SubsExt(B_k, S[a ... a + 2^{i-2}]);
                    if (l_ex + r_ex > a − (k − 1) · 2^{i-2}) return false;
                }
            }
        }
return true;
```

Figure 3. Pseudo code of algorithm that solves square-freeness problem

So for every block B_{k-1} we obtain all occurrences of B_{k-1} in $B_{k+1} \cdot B_{k+2}$. From the lemma above follows that we can represent all occurrences using four arithmetic progressions. Therefore we need to check whether on not block B_{k-1} and progression $\langle a, p, t\rangle$ of its occurrences form a square? We propose the following algorithm for the check:

- If $t = 0$ then there are no squares. The algorithm moves to the next block;
- If $t = 1$ then we obtain ℓ_{ex}, r_{ex} using **SubsExt** problem for B_{k-1} and $S_j[a \ldots a + 2^{i-2}]$. If $\ell_{ex} + r_{ex} > a - (k-1) \cdot 2^{i-2}$ then there exist at least one square and the algorithm returns **false**. Otherwise there are no squares and the algorithm moves to the next block;

- If $t \geq 2$ there exists at least one square. The algorithm returns **false**.

Pseudo code for square-freeness checking presented at figure 3.

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

4.4 PS-table

The *pure squares table* (PS-table) is a rectangular table $PS(\mathbb{S})$ that stores information about all pure squares in the text in a compressed form. The size of $PS(\mathbb{S})$ is equal to $(\lfloor \log |S| \rfloor + 1) \times (|\mathbb{S}| + 1)$. It is convenient to start numbering of rows and columns of PS-tables with 0. We denote the cell in the i-th row and j-th column of table by $PS(i, j)$. The cell $PS(0, 0)$ is always left blank. The cells $PS(0, j)$ with $j > 0$ contain the rules of the SLP $\mathbb{S}$ ordered such that the lengths of the texts they derive increase. (If some rules derive texts of the same length then the rules are listed in an arbitrary but fixed order). Thus, the first cells of the 0-th row contain terminal rules followed by rules that derive texts of length 2, etc. The cells $PS(i, 0)$ with $i > 0$ contain segments $[2^{i-1}, 2^i - 1]$. In every cell $PS(i, j)$ with $i, j > 0$, we shall store information about families of pure squares. There exist three types of stored families:

- empty family (stored as $\varnothing$);
- family of pure squares with fixed root length (stored using triple $\{|x|, c_l, c_r\}$ where $|x|$ is length of root, c_l is center position of leftmost pure square in $\mathbb{S}_j$, c_r is center position of rightmost pure square in $\mathbb{S}_j$);
- family of pure squares with float root length;

The reader may wish to look at the example of the PS-table for the SLP $\mathbb{F}_7$ in Appendix A in which all square families has fixed root length.

Once a PS-table is constructed, it is easy to solve the following problems:

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

More complicated questions also can be answered. For instance, suppose we are given SLPs $\mathbb{S}$ and $\mathbb{P}$ such that $\mathbb{P}$ derives a pure square xx. The question is whether or not xx occurs in the text S?

Sketch of algorithm:

The algorithm computes length of xx using $O(|\mathbb{P}|)$ time. Next it find a row of the PS-table such that $2^{i-1} < |x| < 2^i$ using $O(\log |S|)$ time. Finally the algorithm checks whether xx belongs to a family encoded in any cell of the row. Obviously the algorithm skip empty families. The following cases are remained to be considered:

- the algorithm process the following family: $\{|y|, c_l, c_r\}$. If $|x| \neq |y|$ then the algorithm moves to the next family. Otherwise it take subgrammar $\mathbb{S}[c_l - |x| \dots c_r + |x|]$ using $O(|\mathbb{S}|)$ time and run pattern matching algorithm on $\mathbb{S}[c_l - |x| \dots c_r + |x|]$ and $\mathbb{P}$ using $O(|\mathbb{P}|^3)$ time.

- the algorithm process the following family: $k, \langle a, p, t\rangle, \alpha_L, \alpha_R, \gamma_L, \gamma_R$. If there is no integer $t' \in \overline{0 \dots t}$ such that $t' = \frac{|x|-a+(k-1)\cdot 2^{i-2}}{p}$ or $|x| \geq \min(\alpha_R - \alpha_L, \gamma_R - \gamma_L)$ then the algorithm moves to the next family. Otherwise the family reduced to family with fixed root length $\{|x|, \max(\alpha_L + |x|, \gamma_L) + 1, \min(\alpha_R, \gamma_R - |x|)\}$ and processed as shown above.

If the algorithm find match at some family then it immediately stops and return **true**. If the algorithm find no match checking all families in the row then it return **false**.

In the worst case we need $O(\max\{|\mathbb{S}|, \log |S|\} \cdot |\mathbb{P}|^3)$ time. Moreover, we can modify the algorithm to save information about all occurrences of xx in S. Clearly, the general pattern matching algorithm also solves these problem, but it needs $O(|\mathbb{P}||\mathbb{S}|^2)$ time. Therefore using information from the PS-table will be more effective if $|\mathbb{S}|$ much larger than $|\mathbb{P}|$.

Another natural problem is the following. Let an SLP $\mathbb{S}$ derive the text S and let a position i be fixed. The problem is to construct an SLP that derives all pure squares starting from the position i in S. (Since by the definition each SLP derives only one string, we mean here an SLP that derives a string consisting of all pure squares in question separated by a new symbol.) Altogether there are $O(\log |\mathbb{S}|)$ rules that contain i.

Lemma 4.3 ([3]). *If there are three squares xx, yy, zz with $|x| < |y| < |z|$ that start at the same position of some string, then $|x| + |y| \leq |z|$.*

The lemma implies that for every interval of lengths there exist at most two squares starting from an arbitrary position. Therefore altogether there are $O(|\mathbb{S}|)$ pure squares starting at a particular position. So we can gather information about all pure squares in an explicit form. We can build an

SLP that derives all pure squares starting in a particular position in $O(|\mathbb{S}|^2)$ time using the substring taking algorithm.

4.5 Searching all pure squares problem

Remind the main problem:
PROBLEM: **Searching all pure squares**
INPUT: an SLP $\mathbb{S}$ that derives text S;
OUTPUT: a compressed presentation of all pure squares in S;

To solve the problem remain to recognize all pure squares between a block B_k and an arithmetic progression $\langle a, p, t\rangle$ of its occurrences. Since t can be exponentially large relative to $|\mathbb{S}|$ the algorithm cannot consecutively check every occurrence of B_k.

Let α_L, α_R be output of **Period termination** problem for $\mathbb{S}_j, B_k = S_j[(k-1)\cdot 2^{i-2} \dots k\cdot 2^{i-2} - 1]$. α_L, α_R called *defined* if they satisfies the following inequalities: $(2k-1)\cdot 2^{i-2} - (a + p\cdot t) \le \alpha_L$, $\alpha_R < a + 2^{i-2}$. Otherwise they are called *undefined*. Since $2^i - 1$ is the greatest length of a root then start positions of pure squares can not be further right than $(2k-1)\cdot 2^{i-2} - (a + p\cdot t)$. So it does not matter where the p-periodicity terminates outside. Analogously γ_L, γ_R defined as output of **Period termination** problem for $\mathbb{S}_j, S_j[a \dots a + p\cdot t]$. γ_L, γ_R called *defined* if they satisfies the following inequalities: $(k-1)2^{i-2} \le \gamma_L$ and $\gamma_R < 2(a + p\cdot t) - (k-1)2^{i-2}$. Otherwise they are called *undefined*.

For the positions $\alpha_L, \alpha_R, \gamma_L$ and γ_R the following lemmas are valid:

Lemma 4.4 ([1]). *If one of α_R or γ_L is defined, then the other one is defined, and $\alpha_R - \gamma_L \le p$.*

Lemma 4.5 ([1]). *If both α_R and γ_L are undefined, then none of the repetitions possible containing B_k are a square.*

Therefore squares exists if and only if α_R and γ_L are defined. Let us consider possible relative positions of α_R and γ_L:

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

Most cases (except $(\gamma_L, \alpha_R]$) can be solved using the following lemma:

Lemma 4.6 ([1]). *If both α_R, γ_L are defined then:*

1. *Repetitions 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 repetitions constitute a family of repetitions that corresponds to the difference $|x| = a + t' \cdot p - (k-1)2^{i-2}$, provided that there exists some $t' \in \overline{0 \ldots t}$ such that $\gamma_L - \alpha_L = a + t' \cdot p - (k-1)2^{i-2}$.*

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

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

At each case of the previous lemma the algorithm may obtain exponentially many repetitions. Therefore it stores in compressed way: $\{|x|, c_l, c_r\}$ where $|x|$ is length of the root, c_l is center of leftmost repetition, c_r is center of rightmost repetition. So if $\alpha_R < \gamma_L$ the algorithm may obtain the following three families of repetitions at worst case: $\{\gamma_L - \alpha_L, k \cdot 2^{i-2}, \alpha_R\}, \{\gamma_R - \alpha_R, \alpha_R + 1, \gamma_L\}, \{\gamma_L - \alpha_L, \gamma_L + 1, \min\{a, (k+1) \cdot 2^{i-2}\}\}$. Else the algorithm may obtain the following two families of repetitions at worst case: $\{\gamma_L - \alpha_L, k \cdot 2^{i-2}, \gamma_L - 1\}, \{\gamma_R - \alpha_R, \alpha_R, \min\{a, (k+1) \cdot 2^{i-2}\}\}$.

The last case can be solved using the following lemma:

Lemma 4.7 ([1]). *If α_R, γ_L are defined and $\gamma_L < \alpha_R$, then there might be a family of repetitions associated with each of the differences $|x| = a + p \cdot t' - (k-1) \cdot 2^{i-2}$ where $t' \in \overline{0 \ldots t}$, with centers at positions h, such that $\gamma_L < h \leq \alpha_R$. The repetitions 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 $|x| < \min(\alpha_R - \alpha_L, \gamma_R - \gamma_L)$.*

Since families of pure squares have float centers and float length of root the algorithm store it exactly: $k, \langle a, p, t \rangle, \alpha_L, \alpha_R, \gamma_L, \gamma_R$.

Pseudo code of the algorithm presented at figure 4.

```
for each (𝕊_j ∈ 𝕊)
    for each (i ∈ {1, ..., i_0})
        partition text S_j into 12 blocks of length 2^{i-2};
        for each (B_k, k ∈ {5, ..., 12}) {
            var progressions = PM(B_{k-1}, B_{k+1} · B_{k+2});
            for each (⟨a, p, t⟩ ∈ progressions) {
                α_R, α_L = PeriodTermination(𝕊_j, B_k);
                γ_L, γ_R = PeriodTermination(𝕊_j, S_j[a ... a + p · t]);
                if (α_R is undefined or γ_L is undefined)
                    break;
                if (α_L is defined and exists t' | γ_L − α_L = a + t' · p − (k − 1)2^{i−2}) {
                    var repetitions = α_R < γ_L ? {γ_L − α_L, k2^{i−2}, α_R} : {γ_L − α_L, k2^{i−2}, γ_L − 1};
                    PurityCheck(repetitions);
                }
                if (γ_R is defined and exists t'' | γ_R − α_R = a + t'' · p − (k − 1)2^{i−2}) {
                    var repetitions = α_R < γ_L ? {γ_L − α_L, γ_L + 1, min{a, (k + 1)2^{i−2}}} : {γ_R − α_R, α_R, min{a, (k + 1)2^{i−2}}};
                    PurityCheck(repetitions);
                }
                if (α_L, γ_R are defined and t' = t'') {
                    if (α_R < γ_L) {
                        var repetitions = {γ_R − α_R, α_R + 1, γ_L};
                        PurityCheck(repetitions);
                    }
                    var pureSquares = k, ⟨a, p, t⟩, α_L, α_R, γ_L, γ_R;
                }
            }
        }
```

Figure 4. Pseudo code of algorithm that solves searching all pure squares problem

Theorem 4.8. *There is algorithm that solves Searching all pure squares problem using* $O(|\mathbb{S}|^4 \cdot \log^2 |S|)$ *time and* $O(|\mathbb{S}| \cdot \max(|\mathbb{S}|, \log |S|))$ *space.*

5 Conclusion

We have presented an algorithm that, given an SLP $\mathbb{S}$ deriving a text S, fills out a table containing information about all pure 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.
- The algorithm is quite difficult from the viewpoint of practical implementation. Also it is not excluded that the constants hidden in the "O" notation are actually very big.

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

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

[4] A. Orth G. R. Grant A. J. Jeffreys F. Bonhomme, E. Rivals 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] W. Plandowski L. Gasieniec, M. Karpinski 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.

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

[7] 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.

[8] A. Shinohara M. Miyazaki 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.

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

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

[11] A. Ishino A. Shinohara T. Nakamura W. Matsubara, S. Inenaga 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.

[12] S. Inenaga W. Matsubara and A. Shinohara. Testing square-freeness of strings compressed by balanced straight line program. *Fifteenth Computing: The Australasian Theory Symposium (CATS 2009)*, (94):19–28, 2009.

[13] A. Shinohara Y. Shibata, M. Takeda 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.

A Examples

A.1 Leech word square-freeness

Let us consider the SLP $\mathbb{L}$ that derives substring of Leech square free word:

$$\mathbb{L}_1 = a, \mathbb{L}_2 = b, \mathbb{L}_3 = c, \mathbb{L}_4 = \mathbb{L}_2 \cdot \mathbb{L}_3, \mathbb{L}_5 = \mathbb{L}_2 \cdot \mathbb{L}_1, \mathbb{L}_6 = \mathbb{L}_1 \cdot \mathbb{L}_4$$
$$\mathbb{L}_7 = \mathbb{L}_3 \cdot \mathbb{L}_4, \mathbb{L}_8 = \mathbb{L}_6 \cdot \mathbb{L}_5, \mathbb{L}_9 = \mathbb{L}_8 \cdot \mathbb{L}_4, \mathbb{L}_{10} = \mathbb{L}_8 \cdot \mathbb{L}_7, \mathbb{L}_{11} = \mathbb{L}_{10} \cdot \mathbb{L}_9$$

Parse tree for $\mathbb{L}$ presented at figure 4.

Let us check square freeness of $\mathbb{L}$. Since $|L| = 15$ then the algorithm check the following segments of root length: [1, 1], [2, 3], [4, 7]. Notice that rules of $\mathbb{L}$ already ordered by length of derived text.

- **checking square freeness of squares that root length equals to 1**

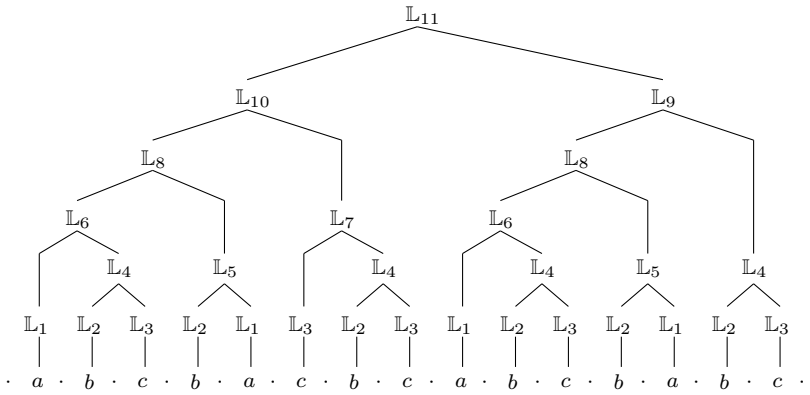

Figure 5. The parse tree of $\mathbb{L}_{11}$ that generates *a b c b a c b c a b c b b a b c*

Firstly the algorithm skip terminal rules. Next it consecutively look for squares exactly in other rules. For instance, let us consider how it checks $\mathbb{L}_{10} = \mathbb{L}_8 \cdot \mathbb{L}_7$. The algorithm find $L_8[5] = a$ and $L_7[1] = c$ using $\mathbb{L}_{10}$. Since $a \neq c$ it moves to $\mathbb{L}_{11}$.

It is clear that $\mathbb{L}_4, \ldots, \mathbb{L}_{11}$ have no squares of length 2 around its cut positions.

- **checking square freeness of squares that root length belong to [2, 3]**

  The algorithm skip all rules that have length less 4 (i.e. $\mathbb{L}_1, \ldots, \mathbb{L}_7$). For instance, let us consider how it checks $\mathbb{L}_8 = \mathbb{L}_6 \cdot \mathbb{L}_5$. Since $|L_8| = 5$ and block length is equal to 1 then the algorithm construct an SLP $\mathbb{L}'_8$ that derives text *$ $ $ $ $ a b c b a $ $ $ $ $ $*, where *$* is special symbol not from Σ. At figure 5 presented partition of L'_8 into blocks of length 1.

$	$	$	$	$	a	b	c	b	a	$	$	$	$	$	$
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 5. Partition L'_8 into blocks

Remind that the algorithm necessary to check blocks $B_5, \ldots, B_{12}$. It no need to check B_5 since it contains *$*. Also it no need to check B_9

and B_{10} since the search area is out of L_8. For instance, let us consider how it checks B_7. The algorithm build $\mathbb{L}'_8[6\dots7]$ that derives B_7 and $\mathbb{L}'_8[8\dots11]$ that derives search area $B_9 \cdot B_{10}$. Next it runs pattern matching algorithm on $\mathbb{L}'_8[6\dots7]$, $\mathbb{L}'_8[8\dots11]$ and obtain occurrence of B_7 at position 9 of L'_8. Finally it runs **SubsExt** problem with the following parameters: $\mathbb{L}'_8$ and positions 6, 7, 8, 9. The algorithm obtains $\ell_{ex} = r_{ex} = 0$. Since $\ell_{ex} + r_{ex} = 0 = a - (k-1) \cdot 2^{i-2} = 8 - (9-1) \cdot 1$ then algorithm moves to L_9.

- **checking square freeness of squares that root length belongs to [4, 7]**

  The algorithm skip all rules that have length less 8 (i.e. $\mathbb{L}_1, \dots, \mathbb{L}_9$). For instance, let us consider how it checks $\mathbb{L}_{11} = \mathbb{L}_{10} \cdot \mathbb{L}_9$. Firstly it construct an SLP $\mathbb{L}'_{11}$ that derives L surrounded with $\$$ and $|L'_{11}| = 32$. At figure 6 presented partition of L'_{11} into blocks of length 2.

$\$\$$	$\$\$$	$\$\$$	$\$\$$	ab	cb	ac	bc	ab	cb	ab	$c\$$	$\$\$$	$\$\$$	$\$\$$	$\$\$$
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 6. Partition L'_{11} into blocks

Next the algorithm find no squares for $B_5, \dots, B_8$ since pattern matching algorithm returns empty set of results. Next the algorithm find occurrence of B_9 at position 22 of L'_{11} and run **SubsExt** problem with the following parameters: $\mathbb{L}'_{11}$ and positions 16, 18, 20, 22. Since $\ell_{ex} = r_{ex} = 0$ the algorithm moves to B_{10}. Finally the algorithm find no squares for B_{10}, B_{11}, B_{12} since correspond search areas contains $\$$.

A.2 Construction PS-table for $\mathbb{F}_7$

Let us consider how the algorithm construct PS-table for $\mathbb{F}_7$ that derives text $a\ b\ a\ a\ b\ a\ b\ a\ a\ b\ a\ a\ b$. The PS-table size is equal to $(\lfloor \log |F_7| \rfloor + 1) \times (|\mathbb{F}_7| + 1) = 4 \times 8$.

- **looking for squares that root length equals to 1**

  Firstly the algorithm mark cells with $\varnothing$ for rules with length less than 2. Next it consecutively look for squares exactly in other rules. PS(1,

3) = ∅ since $F_1[1] = a \neq b = F_2[1]$. Analogously PS(1, 4) = PS(1, 6) = ∅. PS(1, 5) = {1, 3, 3} since $F_4[3] = a = F_3[1]$. PS(1, 7) = {1, 8, 8} since $F_6[8] = a = F_5[1]$. After first step PS-table has the following view:

	$\mathbb{F}_1 = a$	$\mathbb{F}_2 = b$	$\mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2$	$\mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1$	$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3$	$\mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4$	$\mathbb{F}_7 = \mathbb{F}_6 \cdot \mathbb{F}_5$
[1, 1]	∅	∅	∅	∅	{1, 3, 3}	∅	{1, 8, 8}
[2, 3]							
[4, 7]							

- **looking for squares that root length belongs to [2, 3]**

  The algorithm mark cells with ∅ for rules with length less than 4. For instance, let us consider how the algorithm fill PS(2, 6). The algorithm construct SLP $\mathbb{F}'_6$ that derives text *$ $ $ a b a a b a b a $ $ $ $ $ $* of length 16. At figure 7 presented partition of F'_6 into blocks.

$	$	$	a	b	a	a	b	a	b	a	$	$	$	$	$
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 7. Partition F'_6 into blocks

B_5: using **PM** problem the algorithm obtain occurrence of B_5 at position 8 of F'_6 on $\mathbb{F}'_6[5 \dots 6], \mathbb{F}'_6[7 \dots 9]$; the substring extending algorithm obtain $\ell_{ex} = r_{ex} = 1$ on $\mathbb{F}'_6$ with parameters 5, 6, 8, 9; so family of repetitions {3, 7, 7} was found;

B_6: using **PM** problem the algorithm obtain occurrence of B_6 at position 9 of F'_6 on $\mathbb{F}'_6[6 \dots 7], \mathbb{F}'_6[8 \dots 10]$; the substring extending algorithm obtain $\ell_{ex} = 2, r_{ex} = 0$ on $\mathbb{F}'_6$ with parameters 6, 7, 9, 10; so family of repetitions {3, 7, 7} was found;

B_7: using **PM** problem the algorithm obtain occurrence of B_7 at position 9 of F'_6 on $\mathbb{F}'_6[7 \dots 8], \mathbb{F}'_6[9 \dots 11]$; the substring extending algorithm obtain $\ell_{ex} = 0, r_{ex} = 1$ on $\mathbb{F}'_6$ with parameters 7, 8, 9, 10; so no family of repetitions was found;

B_8: using **PM** problem the algorithm obtain occurrence of B_8 at position 10 of F'_6 on $\mathbb{F}'_6[8 \dots 9], \mathbb{F}'_6[10 \dots 12]$; the substring extending algorithm obtain $\ell_{ex} = r_{ex} = 1$ on $\mathbb{F}'_6$ with parameters 8, 9, 10, 11; so family of repetitions {2, 8, 9} was found;

B_9: using **PM** problem the algorithm obtain occurrence of B_9 at position 11 of F'_6 on $\mathbb{F}'_6[9 \dots 10], \mathbb{F}'_6[11 \dots 13]$; the substring extending algorithm obtain $\ell_{ex} = 1, r_{ex} = 0$ on $\mathbb{F}'_6$ with parameters 9, 10, 11, 12; so family of repetitions {2, 9, 9} was found;

The algorithm skip blocks $B_{10}, \dots, B_{12}$ since the search area consist of *$*. After merging families of repetitions the algorithm have the following result: {3, 7, 7}, {2, 8, 9}. Finally the algorithm check purity of families and shift them form F'_6 to F_6.

After second step PS-table has the following view:

	$\mathbb{F}_1 = a$	$\mathbb{F}_2 = b$	$\mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2$	$\mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1$	$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3$	$\mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4$	$\mathbb{F}_7 = \mathbb{F}_6 \cdot \mathbb{F}_5$
[1, 1]	∅	∅	∅	∅	{1, 3, 3}	∅	{1, 8, 8}
[2, 3]	∅	∅	∅	∅	∅	{3, 4, 4}, {2, 5, 6}	{3, 10, 10}
[4, 7]							

- **looking for squares that root length belongs to [4, 7]**

  The algorithm mark cells with ∅ for rules with length less than 8. The algorithm construct SLP $\mathbb{F}'_7$ that surrounds with *$* and has length 32. At figure 8 presented partition of F'_7 into blocks. Let us consider how the algorithm fill PS(3, 7). It skips B_5 since B_5 contains *$*.

$$	*$$*	*$$*	*$$*	*$a*	*ba*	*ab*	*ab*	*ab*	*aa*	*b$*	*$$*	*$$*	*$$*	*$$*	*$$*
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 8. Partition F'_7 into blocks

B_6: using **PM** problem the algorithm obtain occurrence of B_6 at position 15; the substring extending algorithm obtain $\ell_{ex} = 1, r_{ex} = 3$ on $\mathbb{F}'_7$ with parameters 10, 12, 14, 18; so family of repetitions {5, 14, 15} was found;

B_7: using **PM** problem the algorithm obtain occurrence of B_7 at position 17; the substring extending algorithm obtain $\ell_{ex} = 3, r_{ex} = 1$ on $\mathbb{F}'_7$ with parameters 12, 14, 16, 20; so family of repetitions {5, 14, 15} was found;

B_8: using **PM** problem the algorithm obtain occurrence of B_8 at position 20; the substring extending algorithm obtain $\ell_{ex} = r_{ex} =$

0 on $\mathbb{F}'_7$ with parameters 14, 16, 18, 22; so there are no families of repetitions;

B_9: using **PM** problem the algorithm obtain no occurrence of B_9; so there are no families of repetitions;

The algorithm skips B_{10} and B_{11} since search areas consist of \$. It skips B_{12} since B_{12} contains \$. After merging families of repetitions the algorithm have the following result: {5, 14, 15}. Finally the algorithm check purity of families and shift them form F'_7 to F_7.

Finally PS-table has the following view:

	$\mathbb{F}_1 = a$	$\mathbb{F}_2 = b$	$\mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2$	$\mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1$	$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3$	$\mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4$	$\mathbb{F}_7 = \mathbb{F}_6 \cdot \mathbb{F}_5$
[1, 1]	∅	∅	∅	∅	{1, 3, 3}	∅	{1, 8, 8}
[2, 3]	∅	∅	∅	∅	∅	{3, 4, 4}, {2, 5, 6}	{3, 10, 10}
[4, 7]	∅	∅	∅	∅	∅	∅	{5, 5, 6}

A.3 Text with complex family of pure squares

Let us consider an SLP $\mathbb{E} = \mathbb{E}_l \cdot \mathbb{E}_r$ such that $\mathbb{E}_l$ derives text $(a\ b\ a\ b\ a)^6\ a\ b\ a$ and $\mathbb{E}_r$ derives text $b\ a\ a\ b\ a\ (a\ b\ a\ b\ a)^7$. So $\mathbb{E}$ derives text $(a\ b\ a\ b\ a)^7\ a\ b\ a\ (a\ b\ a\ b\ a)^7$ and $|E| = 73$. Let us consider how the algorithm looking for pure squares for rule $\mathbb{E}$ and segment [16, 31]. The algorithm construct SLP $\mathbb{E}'$ that surrounds with \$ and has length 128. The partition of E' into blocks presented at figure 9.

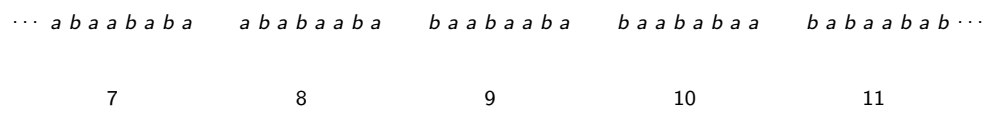

Figure 9. Partition E' into blocks

Let us consider how the algorithm process block B_8. It find occurrence of B_8 in $B_9 \cdot B_{10}$ at positions $\langle 72, 2, 5\rangle$. Since the algorithm find more than one occurrence it extends periodicity to calculate parameters $\alpha_L, \alpha_R, \gamma_L, \gamma_R$. So $\alpha_L = \infty, \alpha_R = 69, \gamma_L = 66, \gamma_R = \infty$. According to lemma !!! the algorithm find complex family of pure squares associated with each of root length {18, 23} and centred at positions {66, 67, 68, 69}. Next we write every root from the family exactly:

- **roots of length 18**

  $a\ b\ a(a\ b\ a\ b\ a)^3$, $b\ a(a\ b\ a\ b\ a)^3a$, $a(a\ b\ a\ b\ a)^3a\ b$, $(a\ b\ a\ b\ a)^3a\ b\ a$;

- **roots of length 23**

  $a\ b\ a(a\ b\ a\ b\ a)^4$, $b\ a(a\ b\ a\ b\ a)^4a$, $a(a\ b\ a\ b\ a)^4a\ b$, $(a\ b\ a\ b\ a)^4a\ b\ a$;