Computing All Pure Squares in Compressed Texts

Plenary Conference AutoMathA : from Mathematics to Applications 2009

Lesha 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 $O(\max(|\mathbb{S}|^5 \log |S|, |\mathbb{S}|^3 \log^3 |\mathbb{S}| \log |S|))$ time and requires $O(|\mathbb{S}|^3)$ 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) [8–10, 13], collage-systems [7], string representations using antidictionaries [14], 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 [13].

While compressed representations save storage space, there is a price to pay: some classical problems on strings become computationally hard when

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

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** [8] and **Literal shuffle** [2]. On the other hand, there exist problems that admit algorithms working rather well on compressed representations: **Pattern matching** [8,12], **Longest common substring** [10], **Computing all palindromes** [10]. 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 [3].) 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 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:

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 describe the structure of a PS-table and present an algorithm that, given an

[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.

SLP $\mathbb{S}$, fills out the corresponding PS-table using time and space resources that depend polynomially on the size of $\mathbb{S}$ and logarithmically on the size of the text $\mathbb{S}$ generates. Our algorithm 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. Next we discuss functionality of PS-tables and list several problems that can be easily solved whenever a PS-table is available. In particular, one can easily test square-freeness of a given compressed text since it is obvious that if a text has no pure squares, then it has no squares at all. (A polynomial algorithm for testing square-freeness of a compressed text has been recently developed in [11] but only for the special case of SLPs, in which the parse tree of the text is balanced.) In Section 4 we summarize our results. In Appendix A we present a sketch of the algorithm's logic. Appendix B contains an illustrative example. Finally, in Appendix C we describe nontrivial operations with SLPs that are invoked in the paper.

2 Preliminaries on SLPs

Let Σ be an arbitrary finite alphabet. 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*).

Thus, an SLP is a context-free grammar in Chomsky normal form. It is easy to see that every SLP $\mathbb{S}$ generates exactly one string $S \in \Sigma^+$. We refer to S as the *text* generated by $\mathbb{S}$.

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

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

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

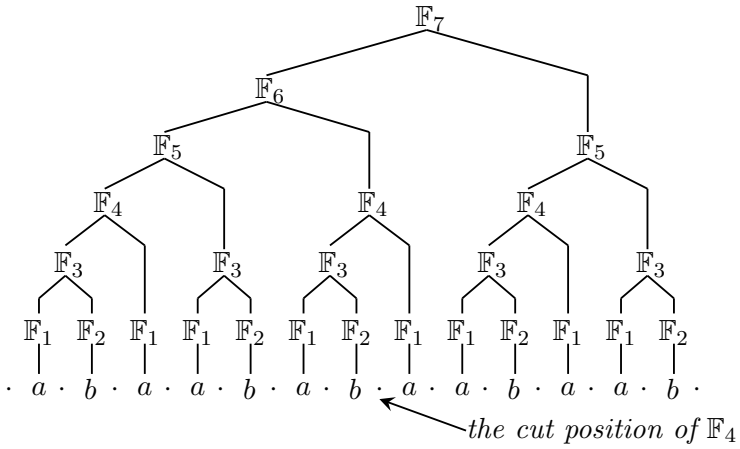


Figure 1. The parse tree of the SLP that generates $F_7 = abaababaabaab$

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 $|\mathbb{S}|$. Similarly, the *length* of a text S is denoted by $|S|$.

A *position* in a text S is a point between consecutive letters. 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. A substring of S starting at a position t_1 and ending at a position t_2, $0 \leq t_1 < t_2 \leq |S|$, is denoted by $S[t_1 \ldots t_2]$. We say that a substring *touches* some position t if t is located inside the substring or on its border. 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. The position $|x|$ of a square xx is called the *center* of xx and x is referred to as the *root* of xx.

3 Computing all pure squares

3.1 Structure of PS-tables

Let us fix an SLP $\mathbb{S}$ that derives a text S. 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 ith row and jth column of $PS(\mathbb{S})$ 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 0th row contain terminal rules followed by rules that derive texts of length 2, etc. The cells $PS(i, 0)$ with $i > 0$ contain the intervals $[2^{i-1}, 2^i)$. In the cell $PS(i, j)$ with $i, j > 0$, we shall store information about the family of all pure squares such that

(1) they touch the cut position of the rule $\mathbb{S}_j$;

(2) they are contained in the text S_j;

(3) lengths of their roots belong to the interval $[2^{i-1}, 2^i)$.

The information is represented by an arithmetic progression and some additional integer parameter(s). (An arithmetic progression is stored as a triple $\langle a, p, q\rangle$ where integers a, p, and q are respectively the initial term, the difference and the number of terms of the progression.) The meaning of the progression and the additional parameters corresponding to a given family of all pure squares may vary depending on the nature of the family so this meaning will be explained stepwise in the course of presenting the main loop of our algorithm. There is a simple case when all pure squares in the family have the same length; then the arithmetic progression consists of the positions of square centers while the only additional parameter is the root length. (The reader may wish to look at the example of the PS-table for the SLP $\mathbb{F}_7$ in Appendix B in which all square families are of this sort). However, there is also a hard case in which more parameters are needed and the meaning of the parameters and the progression is less intuitive.

Let us briefly explain why the restrictions (1)–(3) are consistent with our intention to gather information about **all** pure squares. The restrictions (1)–(2) *provide* information about location of all pure squares in the text. Let xx be an arbitrary pure square that occurs in the text S. We can find the rule $\mathbb{S}_j$ that is xx touches the cut position of $\mathbb{S}_j$ and xx fully contains in the text S_j. (We start from the root of the parse tree of $\mathbb{S}$. If xx is fully contained in a left/right son of $\mathbb{S}_j$, then we *go to the descendant*. Otherwise xx touches the cut position of $\mathbb{S}_j$ and fully contains in the text S_j. Obviously the algorithm stops after $O(|\mathbb{S}|)$ steps.) So for each rule $\mathbb{S}_j \in \mathbb{S}$ we need to gather information about all pure squares that touch the cut position of $\mathbb{S}_j$ and fully contains in the text S_j. We have no idea how to store such huge family of pure squares, so we just partition it onto $O(\log(|S|))$ families of pure squares by length of the root.

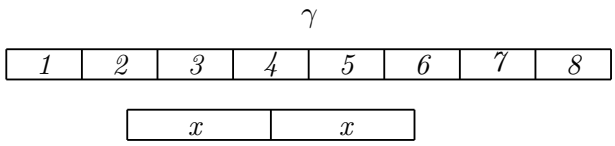


Figure 2. Searching for squares

3.2 Filling out the PS-table

Initialization

First we fill out the cells corresponding to terminal rules with $\varnothing$. Then we check for every rule that derives a text of length 2 whether or not the text is a (necessarily pure) square. We fill out the whole column with $\varnothing$ if the text is not a square; otherwise we fill out the cell in the first row and in the column marked by the rule under inspection with $\langle 1, 0, 1\rangle$, 1.

Main loop

We fix an arbitrary nonterminal rule $\mathbb{S}_j \in \mathbb{S}$. Let γ be a cut position of $\mathbb{S}_j$ and let $[2^{i-1}, 2^i)$ be the current interval of lengths of roots. We consider the 2^i-neighborhood of γ. Let ε be arbitrary (but fixed) symbol such that $\varepsilon \notin \Sigma$. If $|S_j| < 2^{i+1}$ then we mentally complete S_j with ε to length 2^{i+1}. Let S'_j be a text of length 2^{i+1} that contains the 2^i-neighborhood of γ. Next we divide S'_j into 8 blocks of equal length. Thus, the length of each block is 2^{i-2}.

How to find all pure squares in S'_j that touch γ? Here we propose main idea of the algorithm for the problem: let B be a block from the partition. We suppose that B occurs in left part of pure squares. Let BO be a set of start positions of occurrences of B that situated on the right of B. For every position $t_k \in BO$ we reconstruct corresponded block B' and check whether B' and B forms a pure square. If answer is affirmative then we find a pure square.

Why the algorithm find **all** pure squares? Let xx be a required pure square. Clearly left part of xx must fully contains one block from the partition (see Figure 2). The reason is the ratio between $|x|$ and $|B|$. Additionally notice that we need to check only prime five blocks, because other pure squares not touch γ (see Figure 2).

Let us consider four consecutive blocks from the partition that are situated behind B. Now we show that occurrences of B inside the first and the fourth blocks might form squares beyond length limits only. Let t_0 be

start position of B and t_1 – start position of occurrence of B inside the first block. The following estimation of maximum size of blocks extension holds: $2 \cdot (t_1 - (t_0 + |B|)) < 2 \cdot 2^{i-2} = 2^{i-1}$. So length of maximum root is bounded above by 2^{i-1} that contradicts with estimation on length of root. Let t_2 be start position of occurrence of B inside the forth block. The following estimation of minimum size of blocks extension holds: $2 \cdot (t_2 - (t_0 + |B|)) \geq 3 \cdot 2^{i-1}$. So length of minimum root is bounded below by 2^{i+1} that contradicts with estimation on length of root. So we are interested to find start positions of occurrences of B inside the second and the third blocks (see Figure 3). Further the second and the third blocks that corresponds to B we will call as *local search area* of B. We compute BO using the pattern matching algorithm from [8] (for technical details of the algorithm see Appendix C).

The following lemma from [1] shows the structure of occurrences of B. Recall that a string S has a *period* p if S is a prefix of T^k for some k large enough and some string T of length p.

Lemma 3.1 ([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.*

We obtain the following cases by using Lemma 3.1:

(1) If no occurrence of B exists in the 2^i-neighborhood of γ, then the neighborhood is free from squares of fixed length.

(2) If a single occurrence of B exists in the neighborhood, then we can extend two blocks using the pattern matching algorithm from [8] and the binary search idea in both directions (see Subsection 3.3 for details). So we can obtain at most one pure square.

(3) If more than one occurrence exist in the neighborhood, then we divide all occurrences of B into four arithmetical progressions. There is a problem: how to check whether or not B and arithmetical progression of occurrences of B forms any pure squares?

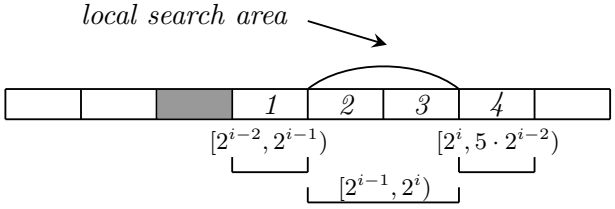


Figure 3. The local search area

Suppose $\langle p_1, p, k\rangle$ is an arbitrary arithmetical progression formed by the start positions of occurrences of B inside some $\frac{|B|}{2}$-block. We extend p-periodicity of B on the right in S, that is we build the string C that starts from t_0, $C[n] = C[n \mod p]$ for every $n \geq t_0$ and $|C| = |S[t_0 \dots |S|]|$. Let α_R be the first position such that $S[\alpha_R] \neq C[\alpha_R]$. Analogously, let α_L be the position where the p-periodicity terminates on the left of B. If the positions α_L and α_R satisfy the following inequalities: $t_0 - (p_k - t_0) + |B| \leq \alpha_L$, $\alpha_R < p_1 + |B|$, then they are called *defined*. Otherwise they are called *undefined*. Since $p_k - t_0$ is the greatest length of a root for fixed B and $\langle p_1, p, k\rangle$, start positions of pure squares can not be further right than $t_0 - (p_k - t_0) + |B|$ and it does not matter where the p-periodicity terminates outside the neighborhood.

Analogously let γ_L and γ_R be the positions where the p-periodicity terminates on the left and on the right of the substring $S[p_1 \dots p_k + |B|]$. If the positions satisfy the following inequalities: $t_0 \leq \gamma_L$ and $\gamma_R < 2p_k - t_0$, then they are called *defined*. Otherwise they are called *undefined*.

We can compute the positions $\alpha_L, \alpha_R, \gamma_L$ and γ_R using first mismatch and period continuation functions by definition (see Subsection 3.3 for details).

For the positions $\alpha_L, \alpha_R, \gamma_L$ and γ_R the next statements are valid:

Lemma 3.2 ([1])**.** *If one of α_R or γ_L is defined, then so is the other one, and $\alpha_R - \gamma_L \leq p$.*

We say that a square xx is *hinged on a block B* if the root x contains B.

Lemma 3.3 ([1])**.** *If both α_R and γ_L are undefined, then none of the repetitions possible hinged on B are a pure square.*

From the previous lemmas it follows that both positions α_R and γ_L are defined.

Lemma 3.4 ([1]). *If both α_R, γ_L are defined then:*

(1) Repetitions that are hinged on B 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 $|x| = p_s - t_0$, provided that there exists some $p_i \in \langle p_1, p, k\rangle$ such that $\gamma_L - \alpha_L = p_s - t_0$.

(2) Repetitions that are hinged on B 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 $|x| = p_r - t_0$, provided that there exists some $p_j \in \langle p_1, p, k\rangle$ such that $\gamma_L - \alpha_L = p_r - t_0$.

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 $\gamma_R - \alpha_R = \gamma_L - \alpha_L$.

Lemma 3.5 ([1]). *A family of repetitions contains a pure square if and only if all the repetitions in the family are pure squares.*

Using Lemma 3.4, we can simply obtain two families of repetitions that we store as follows:

(1) $\langle t_0 + |B|, 1, \gamma_L - (t_0 + |B|)\rangle$ is set of centers and $|x| = p_s - t_0$ is length of a root;

(2) $\langle \alpha_R, 1, p_1 - \alpha_R\rangle$ is set of centers and $|x| = p_r - t_0$ is length of a root.

If $\alpha_R < \gamma_L$ ($\gamma_R - \alpha_R = \gamma_L - \alpha_L$), then we can obtain two non-overlapping blocks of length $\gamma_R - \alpha_R$. We need to extend the blocks and check if they generate a pure square (we performed similar operation earlier). If the extension is nonempty then we obtain a family of repetitions similar to Lemma 3.4.

We can easily check whether the family of repetitions consists of pure squares by using Lemma 3.5.

Lemma 3.6 ([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 $|x| = p_i - t_0$, 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 $|x| < min(\alpha_R - \alpha_L, \gamma_R - \gamma_L)$.*

Using Lemma 3.4 we can obtain the family of pure squares that we store as follows: $\langle p_1, p, k\rangle, t_0$ is information about the length of the root, $\alpha_L, \alpha_R, \gamma_L, \gamma_R$ is information about the roots length limits and the location of the centers.

Now we can describe all possible variants of information that has to be written into the cell $PS(i, j)$ of the PS-table:

- $\varnothing$ (for a rule free from pure squares of length in the prescribed interval);
- $\langle c, 0, 1\rangle, |x|$ (for a single square that either was obtained using extension of two blocks or was the singular case of Lemma 3.4 or Lemma 3.6);
- $\langle c, 1, q\rangle, |x|$ (for a family of pure squares obtained via Lemma 3.4;
- $\langle p_1, p, k\rangle, t_0, \alpha_L, \alpha_R, \gamma_L, \gamma_R$ (for a family of pure squares obtained via Lemma 3.6).

3.3 Complexity of the algorithm

Let us isolate main operations that we have used in the algorithm and estimate their complexity:

(1) Problem: **Borders computation**

Input: SLP $\mathbb{S}$ that derives text S, l – length of a block, t and p – start positions of two blocks that $p - t > l$;

Output: positions $\alpha_L, \alpha_R, \gamma_L, \gamma_R$;

Here we propose solution that required $O(|\mathbb{S}|^2 \log^3 |\mathbb{S}|)$ time.

• We take three substrings in $O(|\mathbb{S}|)$ time.

• There may appear at most $O(\log |\mathbb{S}|)$ new rules after taking a substring. We can add them to the compressed structure of overlaps in $O(|\mathbb{S}|^2 \log^3 |\mathbb{S}|)$ time according to [6].

• Finally, we run the period continuation algorithm in $O(|\mathbb{S}| \log |\mathbb{S}|)$ three times (see Appendix C for details).

(2) Problem: **Blocks extension** (discussed in cases 2 and 3 listed after Lemma 3.1)

Input: SLP $\mathbb{S}$ that derives text S, l – length of a block, t and p – start positions of two blocks that $p - t > l$;

Output: **yes**, if there exists position $c \in (t + l, p)$ that $S[t + l \dots c] = S[p + l \dots p + c - t]$ and $S[t + c - p \dots t] = S[c \dots p]$; no – otherwise;

Here we propose solution that required $O(|\mathbb{S}^4|)$ time.

• We compute positions $\alpha_L, \alpha_R, \gamma_L, \gamma_R$ in $O(|\mathbb{S}|^2 \log^3 |\mathbb{S}|)$ time;

• We take subprograms $\mathbb{S}[\alpha_L \dots t], \mathbb{S}[t+l \dots p], \mathbb{S}[p+l \dots \gamma_R]$ in $O(|\mathbb{S}|)$ time (see Appendix C for details).

• Let us denote $\mathbb{LCP}$ – longest common prefix for $\mathbb{S}[t+l \dots p], \mathbb{S}[p+l \dots \gamma_R]$ and $\mathbb{LCF}$ – longest common suffix for $\mathbb{S}[\alpha_L \dots t], \mathbb{S}[t+l \dots p]$. We compute $\mathbb{LCP}$ and $\mathbb{LCF}$ in $O(|\mathbb{S}^4|)$ time using the pattern matching algorithm from [8] and the idea of binary search.

So if $|LCP| + |LCF| > p - t$ then we return **yes**, otherwise – no.

(3) It is clear that we can implement the test from Lemma 3.5 using the pattern matching algorithm from [8] in $O(|\mathbb{S}|^3)$ time.

Thus, computation of $PS(i, j)$ required $O(\max(|\mathbb{S}|^4, |\mathbb{S}|^2 \log^3 |\mathbb{S}|))$ time. So we fill out the whole PS-table in $O(\max(|\mathbb{S}|^5 \log |S|, |\mathbb{S}|^3 \log^3 |\mathbb{S}| \log |S|))$ time.

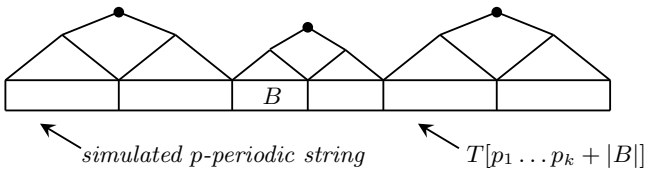


Figure 4. Using the period continuation function

3.4 Functionality of PS-tables

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 the pure square occurs in the text S?

We can compute length of xx in $O(|\mathbb{P}|)$ time. Then we look through the row of the PS-table marked by the interval $[2^{i-1}, 2^i)$ such that $2^{i-1} \le |x| < 2^i$. We need to check whether the pure square belongs to a family encoded in any cell of this row. If the content of some cell has the form $\langle c, 1, q \rangle$, $|x|$,

then we take the subgrammar for $S[c-|x|, c+q+|x|]$ and run the pattern matching algorithm on $\mathbb{S}[c-|x|, c+q+|x|]$ and $\mathbb{P}$. If the content of some cell has form $\langle p_1, p, k\rangle, t_0, \alpha_L, \alpha_R, \gamma_L, \gamma_R$, then we first have to check whether the corresponding family contains a pure square of required length. If it does, then we can collect all these squares into a family with the code of the form $\langle c, 1, q\rangle$, $|x|$ and process it in same way.

In the worst case we need $O(|\mathbb{P}|^3|\mathbb{S}|)$ time. Moreover, we can save information about all occurrences of xx in S and about their number too. 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 longer 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. It is necessary 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 3.7 ([4])**.** *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 it 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 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 $O(\max(|\mathbb{S}|^5 \log |S|, |\mathbb{S}|^3 \log^3 |\mathbb{S}| \log |S|))$ using $O(|\mathbb{S}|^3)$ 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.

The author is grateful to the professor *Mikhail V. Volkov* for positive and calm attitude to the life.

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 Proc. IFIP TCS 2008, pages 87–100, 2008.

[3] *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 8(5):R80 (2007), `http://genomebiology.com/2007/8/5/R80` .

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

[5] *L. Gasieniec, M. Karpinski, W. Plandowski and W. Rytter.* Efficient algorithms for Lempel-Zip encoding. In Proc. SWAT 1996, Lect. Notes Comput. Sci. 1097, pages 392–403 (1996).

[6] *M. Karpinski, W. Rytter and A. Shinohara.* An efficient pattern-matching algorithm for strings with short descriptions. Nordic J. Comput. 4(2): 172–186 (1997).

[7] *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. 298(1):253–272 (2003).

[8] *Y. Lifshits.* Processing compressed texts: a tractability border. In Proc. CPM 2007, Lect. Notes Comput. Sci. 4580, pages 228–240, 2007.

[9] *Y. Lifshits and M. Lohrey.* Querying and embedding compressed texts. In Proc. MFCS 2006, Lect. Notes Comput. Sci. 4162, pages 681–692, 2006.

[10] *W. Matsubara, S. Inenega, A. Ishino, A. Shinohara, T. Nakamura and K. Hashimoto.* Computing longest common substring and all palindromes from compressed strings. In Proc. SOFSEM 2008, Lect. Notes Comput. Sci. 4910, pages 364–375, 2008.

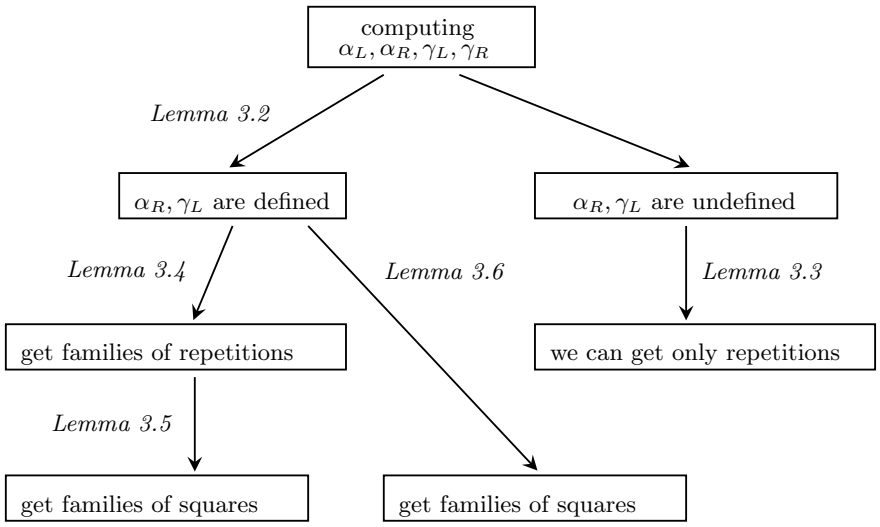
[11] *W. Matsubara, S. Inenaga and A. Shinohara.* Testing square-freeness of strings compressed by Balanced Straight Line Program. In Proc. Fifteenth Computing: The Australasian Theory Symposium (CATS 2009), Wellington, New Zealand. CRPIT, 94, pages 19–28, 2009.

[12] *M. Miyazaki, A. Shinohara and M. Takeda.* An improved pattern matching algorithm for strings in terms of Straight-Line Programs. In Proc. CPM 1997, Lect. Notes Comput. Sci. 1264, pages 1–11, 1997.

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

[14] *Y. Shibata, M. Takeda, A. Shinohara and S. Arikawa.* Pattern matching in text compressed by using antidictionaries. In Proc. CPM 1999, Lect. Notes Comput. Sci. 1645, pages 37–49, 1999.

A Sketch of the algorithm's logic



B An example

We would like to construct the PS-table for $\mathbb{F}_7$ step by step. The PS-table size is equal to $(\lfloor \log |F_7| \rfloor + 1) \times (|\mathbb{F}_7| + 1) = 4 \times 8$.

Step 1. Initialization of the PS-table and filling out cells that correspond to terminal rules with $\varnothing$.

	$\mathbb{F}_1 = a$	$\mathbb{F}_2 = b$	$\mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2$	$\mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1$	$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3$	$\mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4$	$\mathbb{F}_7 = \mathbb{F}_6 \cdot F_5$
$[1,2)$	$\varnothing$	$\varnothing$					
$[2,4)$	$\varnothing$	$\varnothing$					
$[4,8)$	$\varnothing$	$\varnothing$					

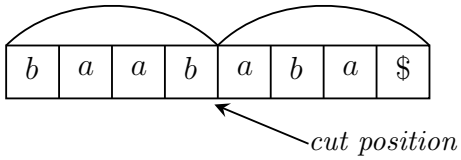
Step 2. Filling out columns that correspond to rules that derive text of length 2 and filling out the row that corresponds to $[1,2)$.

	$\mathbb{F}_1 = a$	$\mathbb{F}_2 = b$	$\mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2$	$\mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1$	$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3$	$\mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4$	$\mathbb{F}_7 = \mathbb{F}_6 \cdot \mathbb{F}_5$
$[1,2)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\langle 3,0,1\rangle, 1$	$\varnothing$	$\langle 8,0,1\rangle, 1$
$[2,4)$	$\varnothing$	$\varnothing$	$\varnothing$				
$[4,8)$	$\varnothing$	$\varnothing$	$\varnothing$				

Step 3. Filling out the row that corresponds to $[2,4)$.

- We fill out $ST(2,4)$ with $\varnothing$ because we can take 2^0-neighborhood of cut position of $\mathbb{F}_4$ only.
- It is easy to see that $\mathbb{F}_5$ has no squares of length 2 and 3, so we set $ST(2,5) = \varnothing$.
- Now we describe filling out $ST(2,6)$ in detail:

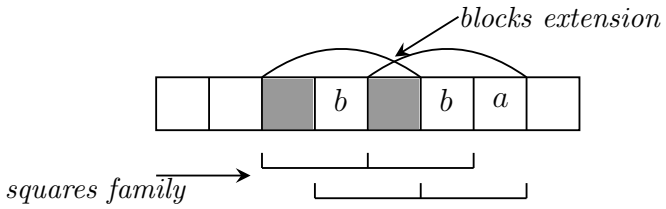
  The neighborhood partition. We take 2^2-neighborhood of the cut position of $\mathbb{F}_6$ and present it as follows:



  Searching for pure squares that contain the first block. We partition the neighborhood into 8 blocks of length 1 and fix the first block that corresponds to the letter b. Next we find an occurrence of the block starting at position 3 of the neighborhood. Finally, we try to extend the block and its occurrence but fail.

  Searching for pure squares that contain the second block. We fix the second block that corresponds to the letter a and find an occurrence of the block starting at position 4 of the neighborhood. We try to extend the block and its occurrence but fail.

  Searching for pure squares that contain the third block. We fix the third block that corresponds to the letter a and find an occurrence of the block starting at position 4 of the neighborhood. We extend the block and its occurrence and find two pure squares:



  Searching for pure squares that contain the fourth block. We fix the fourth block that corresponds to the letter b and find an occurrence of the block starting at position 5 of the neighborhood.

We extend the block and its occurrence and find some family of pure squares.

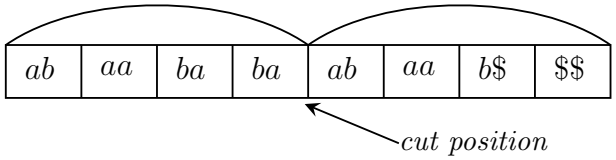
Searching for pure squares that contain the fifth block. We fix the fifth block that corresponds to the letter a and find an occurrence of the block starting at position 6 of the neighborhood. We extend the block and its occurrence and find some family of pure squares.

- We can find the pure square $aabaab$ that starts from position 10 of $\mathbb{F}_7$ similarly.

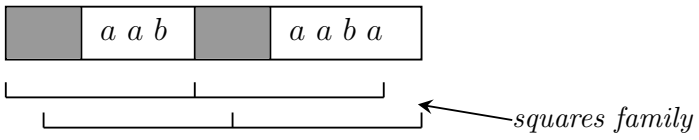
	$\mathbb{F}_1 = a$	$\mathbb{F}_2 = b$	$\mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2$	$\mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1$	$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3$	$\mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4$	$\mathbb{F}_7 = \mathbb{F}_6 \cdot \mathbb{F}_5$
$[1,2)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\langle 3,0,1\rangle, 1$	$\varnothing$	$\langle 8,0,1\rangle, 1$
$[2,4)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\langle 5,1,2\rangle, 2$	$\langle 10,0,1\rangle, 3$
$[4,8)$	$\varnothing$	$\varnothing$	$\varnothing$				

Step 4. Filling out the row that corresponds to $[4,8)$. We fill out $ST(4,4)$, $ST(4,5)$ and $ST(4,6)$ with $\varnothing$ because of the restriction to square length. Now we would like to fill out $ST(4,7)$:

The neighborhood partition. We take the 2^3-neighborhood of cut position of $\mathbb{F}_7$ and partition it onto 8 blocks of length 2:



Searching for pure squares that contain the first block. We fix the first block that corresponds to the word ab and find an occurrence of the block starting at position 3 of the neighborhood. We extend the block and its occurrence and find two pure squares:



Similar cases. If we fix one of the second or the third or the fourth blocks then we will find already known family of pure squares.

Last case. We fix the fifth block that corresponds to the word ab and find an occurrence of the block starting at position 10 of F_7. We extend the block and its occurrence and find the pure square $aabaab$ that already has been found.

Finally we compute the following PS-table:

	$\mathbb{F}_1 = a$	$\mathbb{F}_2 = b$	$\mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2$	$\mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1$	$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3$	$\mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4$	$\mathbb{F}_7 = \mathbb{F}_6 \cdot \mathbb{F}_5$
$[1,2)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\langle 3,0,1\rangle, 1$	$\varnothing$	$\langle 8,0,1\rangle, 1$
$[2,4)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\langle 5,1,2\rangle, 2$	$\langle 10,0,1\rangle, 3$
$[4,8)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\langle 5,1,2\rangle, 5$

C Some operations over SLPs

In Appendix C we describe some nontrivial operations over SLPs that are used in the paper.

Substring taking.

INPUT: an SLP $\mathbb{S}$ that derives a text S and positions $0 < k_1 < k_2 < |S|$;

OUTPUT: an SLP $\mathbb{P}$ that derives the substring $S[k_1, k_2]$;

It is easy to find the shortest string S_i that contains the substring $S[k_1 \dots k_2]$. We can construct the grammar $\mathbb{S}_i$ that generates S_i in $O(|\mathbb{S}|)$ time using hash-tables. Now we need to correct the right and the left borders of the parse-tree $\mathbb{S}_i$. We go down from the root of $\mathbb{S}_i$ along the path that leads to the position k_1 and mark the path. If we obtain a rule that contains the position k_1 in the left part, then we put label 0. If a rule contains k_1 in the right part, then we put label 1. Now we go back to the root and add new rules along the way. We search for the first rule in the path $\mathbb{S}_{i_1} = \mathbb{S}_{i_L} \cdot \mathbb{S}_{i_R}$ that was labelled with 0 and take last rule in the path $\mathbb{S}_{j_1}$ (terminal rule). We add the new rule $\mathbb{S}'_{i_1} = \mathbb{S}_{j_1} \cdot \mathbb{S}_{i_R}$. Next we take an arbitrary rule $\mathbb{S}_{i_k} = \mathbb{S}_{i_L} \cdot \mathbb{S}_{i_R}$ marked by 0 and find the nearest new rule $\mathbb{S}'_{i_{k-1}}$. In the result we add at most $O(\log |\mathbb{S}_i|)$ new rules and we need $O(|\mathbb{S}|)$ time.

Pattern matching

INPUT: SLPs $\mathbb{P}$, $\mathbb{S}$ that derive texts P and S respectively and such that $|\mathbb{P}| \le |\mathbb{S}|$;

OUTPUT: all occurrences P in S in a compressed form;

Theorem C.1 ([8]). *There exists an algorithm that solves* **Pattern matching** *problem in $O(|\mathbb{P}| \cdot |\mathbb{S}|^2)$ time and requires $O(|\mathbb{P}| \cdot |\mathbb{S}|)$ space.*

First mismatch and period continuation functions.
For strings S_i, S_j we define the set of overlaps in the following way:

$$Overlaps(S_i, S_j) = \{k > 0 : S_i[|S_i| - k + 1 \dots |S_i|] = S_j[1 \dots k]\}.$$

For an SLP $\mathbb{S}$ an *overlap table* is a square table that contains a compressed representation of $Overlaps(S_i, S_j)$ in a cell (i, j).

Theorem C.2 ([6]). *There is an algorithm that for every SLP $\mathbb{S}$ computes an overlap table in $O(|\mathbb{S}|^4 \log |\mathbb{S}|)$ time and require $O(|\mathbb{S}|^3)$ space.*

Let us fix some integer k and strings S_i, S_j ($|S_j| > k$). We define the first mismatch function in the following way:

$$FM(S_i, S_j, k) = \min\{q > 0 : S_i[|S_i| - k + q] \neq S_j[q]\}.$$

If no such q exists, then $FM(S_i, S_j, k) = 0$.

Theorem C.3 ([6]). *Let rules S_i, S_j from $\mathbb{S}$ and integer k be fixed. There is an algorithm that computes $FM(S_i, S_j, k)$ with $O(|\mathbb{S}|)$ overlap queries between rules that precede $\mathbb{S}_i$ or $\mathbb{S}_j$.*

Theorem C.4 ([6]). *Let $\mathbb{S}_i = \mathbb{S}_{i_l} \cdot \mathbb{S}_{i_r}$ from $\mathbb{S}$ be nonterminal rules. Assume that the overlap table for all rules preceding $\mathbb{S}_i$ is already computed and there is a period p in S_{i_l}. Then the first mismatch of the period continuation in S_{i_r} can be computed in $O(|\mathbb{S}| \log |\mathbb{S}|)$ time.*