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 $O(|\mathbb{S}|^4 \cdot \log^2 |S|)$ 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), collage-systems, string representations using antidictionaries, 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 [5].

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** [4] and **Literal shuffle** [2]. On the other hand, there exist problems that admit algorithms working rather well on compressed representations: **Pattern matching** [4], **Longest common substring** [6], **Computing all palindromes** [6]. 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. 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] 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

*The author acknowledges support from the Russian Foundation for Basic Research, grant 10-01-00793.

[1] A polynomial algorithm that solves **CAS** for strings represented by Lempel-Ziv encodings was announced in [3]. This representation is slightly more general than that by SLPs. However no details of the algorithm have ever been appeared.

compressed form. Also this implies that we cannot search for squares consecutively by moving from one square to the "next" one.

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 where $0 \le i \le |S|$ we define $S[i]$ as the character between the positions i and $i + 1$ 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 $S[\ell \ldots r]$. 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. The position $|x|$ of a square xx is called the *center* of xx and x is referred to as the *root* of xx. A square xx is called *pure* if x occurs exactly two times in xx. A string S is called *p-periodic* if for a given integer p and every position i such that $0 \le i \le |S| - p$ one has $S[i] = S[i + p]$. The integer p is called the *period* of S.

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 $expr_i$ either 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*). Every SLP $\mathbb{S}$ generates exactly one string $S \in \Sigma^+$ and we refer to S as the *text* generated by $\mathbb{S}$.

We adopt 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 *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.

3 Basic operations

In this section we present some basic operations over SLPs widely used in the paper. The operations are well-known and presented here for the reader's convenience.

Problem: **Subgrammar cutting (SubCut)**
Input: an SLP $\mathbb{S}$ that derives a text S, integers ℓ and r such that $0 \le \ell < r \le |\mathbb{S}|$.
Output: an SLP $\mathbb{S}[\ell \ldots r]$ which derives the text $S[\ell \ldots r]$.
Complexity: There is an algorithm that solves **SubCut** using $O(|\mathbb{S}|)$ time and $O(|\mathbb{S}|)$ space.

Problem: **Pattern matching (PM)**
Input: SLPs $\mathbb{S}$ and $\mathbb{T}$ that derives texts S and T respectively, $|S| \le |T|$.
Output: $O(|\mathbb{T}|)$ arithmetic progressions that describe the start positions of all occurrences of S in T.
Complexity: There is an algorithm [4] that solves **PM** using $O(|\mathbb{T}|^2|\mathbb{S}|)$ time and $O(|\mathbb{T}||\mathbb{S}|)$ space.

Problem: **Substrings extending (SubsExt)**
Input: an SLP $\mathbb{S}$, integers ℓ_1, r_1, ℓ_2 and r_2 where $0 \le \ell_1 < r_1 \le |\mathbb{S}|, 0 \le \ell_2 < r_2 \le |\mathbb{S}|$ 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 $S[1 \dots r_1]$ and $S[1 \dots r_2]$, r_{ex} is the length of the longest common prefix of $S[\ell_1 \dots |S|]$ and $S[\ell_2 \dots |S|]$.
COMPLEXITY: There is an algorithm that solves **SubsExt** using $O(|\mathbb{S}|^3 \log |S|)$ time and $O(|\mathbb{S}|^2)$ space.

PROBLEM: **Period termination**
INPUT: an SLP $\mathbb{S}$, an integer $p > 0$ and positions ℓ, r where $0 \le \ell < r \le |\mathbb{S}|$ such that $S[\ell \dots r]$ is p-periodic substring.
OUTPUT: positions t_L, t_R in the text S where p-periodicity of $S[\ell \dots r]$ terminates from the left and from the right correspondingly. It means that $S[t_L \dots r]$ is p-periodic while $S[t_L + 1 \dots r]$ is not p-periodic.
COMPLEXITY: There exists an algorithm that solves **Period termination** using $O(|\mathbb{S}|^3 \log |S|)$ time and $O(|\mathbb{S}|^2)$ space.

4 Square-freeness checking algorithm

PROBLEM: **Square-freeness**
INPUT: an SLP $\mathbb{S}$ that derives a text S;
OUTPUT: true/false, whether or not S is square-free?

Squares location idea. Suppose xx is a square that occurs in S and c is the center of xx. There is a unique rule $\mathbb{S}_j$ such that xx fully occurs in S_j and xx touches the cut position of $\mathbb{S}_j$. So for every integer $j \in \{1, 2, \dots, |\mathbb{S}|\}$ the algorithm looks for squares that touch the cut position of $\mathbb{S}_j$ only.

We use the following simple idea to obtain squares. If B is a substring of x then B occurs at least two times in xx. So the algorithm may fix some substring B and look for other occurrences of B. Every pair of occurrences of B indicates that some squares are possible. Hence the algorithm has an additional step to check whether or not a pair of substrings indeed corresponds to squares. We would like to note that such an algorithm is unable to find all squares that touches the cut position of $\mathbb{S}_j$ directly. The algorithm partitions all squares into groups by length of root. There is a unique integer i_0 such that $2^{i_0 - 1} \le |x| < 2^{i_0}$. So for a fixed integer j and every integer $i \in \{1, \dots, \lceil \log_2 |S| \rceil\}$ the algorithm looks for squares xx that touch the cut position of $\mathbb{S}_j$ and $2^{i-1} \le |x| < 2^i$.

Local search idea. Suppose j and i are fixed. Let γ be the cut position of $\mathbb{S}_j$. The algorithm partitions the 2^{i+1}-neighborhood of γ into 16 text blocks of equal length. Thus the length of each block is equal to 2^{i-2}. Let us enumerate the blocks from B_1 to B_{16}. For example, the left block that touches γ is B_8, the block left of B_8 is B_7 and the block right of B_8 is B_9. Depending on the length of S_j the neighborhood may contain less than 16 blocks or extreme blocks may have lengths less than 2^{i-2}. For squares whose centers locate at the blocks $B_1, B_2, \dots, B_4$ and $B_{13}, B_{14}, \dots, B_{16}$ we get a contradiction with restrictions on length of squares or with the condition that squares touch γ. Therefore the algorithm looks for squares whose centers locate at blocks $B_5, B_6, \dots, B_{12}$ only.

Let c be the center of a square xx and lec c belong to one of the eight central blocks B_k. Since $2^{i-1} \le |x|$, we conclude that xx contains at least three consecutive blocks B_{k-1}, B_k and B_{k+1}. All occurrences of B_{k-1} in $B_{k+1} \cdot B_{k+2}$ can be obtained using the **PM** algorithm. So the algorithm takes every pair consisting of a block B_{k-1} and its occurrence in $B_{k+1} \cdot B_{k+2}$ and extends them to check whether or not they form any square.

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

arithmetic progression with difference p.

ALGORITHM: For every block B_{k-1} the algorithm computes $\mathbb{B}_{k-1}$ and $\mathbb{B}_{k+1} \cdot \mathbb{B}_{k+2}$ using **SubCut**. Next it invokes **PM** with $\mathbb{B}_{k-1}$ and $\mathbb{B}_{k+1} \cdot \mathbb{B}_{k+2}$. Lemma 4.1 implies that the occurrences can be represented using at most four arithmetic progressions. The algorithm compresses the occurrences into four arithmetic progressions. Finally it verifies whether or not B_{k-1} and a progression $\langle a, p, t\rangle$ of its occurrences form any square. There are the following cases:

- If $t = 0$ then there are no squares that satisfy the restrictions and the algorithm moves to the next block;
- If $t = 1$ then the algorithm computes ℓ_{ex}, r_{ex} invoking **SubsExt** with 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 that satisfy the restrictions and it moves to the next block;
- If $t \geq 2$ there exists at least one square and the algorithm returns **false**.

COMPLEXITY: For each of the eight central blocks the algorithm invokes **SubCut** two times, **PM** once and **SubsExt** at most four times. Hence the main step requires $O(|\mathbb{S}|^3 \cdot \log |S|)$ time and $O(|\mathbb{S}|^2)$ space. The algorithm makes 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.*

5 S-table and its properties

The *squares table* (shortly S-table) is a rectangular table $S(\mathbb{S})$ that holds information about all squares in the text in a compressed form. The size of $S(\mathbb{S})$ is equal to $(\lfloor \log |S| \rfloor + 1) \times (|\mathbb{S}| + 1)$. It is convenient to start numbering rows and columns of S-tables with 0. We denote the cell in the i-th row and j-th column of table by $S(i, j)$. The cell $S(0, 0)$ is always left blank. The cells $S(0, j)$ with $j > 0$ hold 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 hold terminal rules followed by rules that derive texts of length 2, etc. The cells $S(i, 0)$ with $i > 0$ hold segments $[2^{i-1}, 2^i - 1]$. Every cell $S(i, j)$ with $i, j > 0$ holds information about families of squares.

There exist four types of families: an empty family (stored as $\varnothing$), a simple family of squares (stored using 4-tuple $\{|x|, p, c_\ell, c_r\}$ where $|x|$ is the length of root, c_ℓ is the position of the leftmost square, c_r is the center position of the rightmost square), a dynamic family of pure squares, a dynamic family of squares. Both types of dynamic families contain squares with different root lengths and different centers. We are unable to master an explicit compressed representation for the dynamic families that would be easy to handle with. This fact restricts the range of problems that we can solve by usage of S-tables. But both dynamic families support the following properties: *total* (total number of squares that is contained in the family), *reduction by length of root* (for a fixed value of $|x|$, this property reduces a dynamic family to an array of simple families), *reduction by position* (for a fixed position ℓ of S this property returns range of roots of squares that start from ℓ).

Using information from an S-table it is easy 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 start from a fixed position, to find the longest square in S, to check whether on not a text S is square-free. At the same time there are problems that are not easy to decide via S-tables. As an example one can take the following problem: for a given $\mathbb{S}$ that derives a text S to construct an SLP that derives the concatenation of all squares that are contained in S. The example shows that an S-table accumulates a quantitative information about squares rather than information appropriate for searching some fixed squares. It is a common feature of algorithms over SLPs that accumulate information about all objects of a specified type. The similar situation appears in **Computing all palindromes** [6].

6 Computing all squares algorithm

PROBLEM: **CAS**
INPUT: an SLP $\mathbb{S}$ that derives a text S;
OUTPUT: a data structure (an S-table) that contains information about all squares in S in a compressed form.
ALGORITHM: To solve the problem it remains to recognize all squares among a block B_k and an arithmetic progression $\langle a, p, t\rangle$ of its occurrences. Since t can be exponentially large relative to size of $\mathbb{S}$, the algorithm cannot consecutively check every occurrence of B_k.

Let α_L, α_R be the output of **Period termination** for $\mathbb{S}_j, B_k = S_j[(k-1)\cdot 2^{i-2} \dots k \cdot 2^{i-2} - 1]$. The positions α_L, α_R are called *defined* if they satisfy 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, the start positions of pure squares cannot be further right than $(2k-1)\cdot 2^{i-2} - (a+p\cdot t)$. Analogously let γ_L, γ_R be the output of **Period termination** for $\mathbb{S}_j, S_j[a \dots a+p\cdot t]$. The positions γ_L, γ_R are called *defined* if they satisfy 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*. The following lemmas present crucial relations between $\alpha_L, \alpha_R, \gamma_L$ and γ_R.

Lemma 6.1 ([1]). *If one of α_R or γ_L is defined then the other one is defined as well and $\alpha_R - \gamma_L \le p$.*

Case 1: both $\boldsymbol{\alpha_R}$ and $\boldsymbol{\gamma_L}$ are defined.

Lemma 6.2 ([1]). *If both α_R, γ_L are defined then:*

1. *Squares that contain B_k and are centered at positions h such that $h \le \gamma_L$ 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 $t' \in \{0 \dots t\}$ such that $\gamma_L - \alpha_L = a + t'\cdot p - (k-1)2^{i-2}$.*

2. *Squares that contain B_k and are 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 $t'' \in \{0 \dots t\}$ 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 $\alpha_R < h \le \gamma_L$ may exist only if both α_L and γ_R are defined and $\gamma_R - \alpha_R = \gamma_L - \alpha_L$.

Using Lemma 6.2 the algorithm finds simple families of squares. If $\alpha_R < \gamma_L$ the algorithm may find at most three simple families of squares: $\{\gamma_L - \alpha_L, p, k\cdot 2^{i-2}, \alpha_R\}, \{\gamma_R - \alpha_R, p, \alpha_R + 1, \gamma_L\}, \{\gamma_L - \alpha_L, p, \gamma_L + 1, \min\{a, (k+1)\cdot 2^{i-2}\}\}$. Otherwise it may find at most two simple families of squares: $\{\gamma_L - \alpha_L, p, k\cdot 2^{i-2}, \gamma_L - 1\}, \{\gamma_R - \alpha_R, p, \alpha_R, \min\{a, (k+1)\cdot 2^{i-2}\}\}$.

Lemma 6.3 ([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 6.3 the algorithm finds at most one dynamic family of pure squares and stores it in the following way: $\{k, \langle a, p, t\rangle, \alpha_L, \alpha_R, \gamma_L, \gamma_R\}$.

Case 2: both α_R and γ_L are undefined. $S[\alpha_L \dots \gamma_R]$ is p-periodic and contains squares with different centers and different lengths of roots. The algorithm accumulates all squares into a single dynamic family of squares that is stored in the following way: $\{k, \langle a, p, t\rangle, \alpha_L, \gamma_R\}$.

Complexity: For each central block the algorithm computes four arithmetic progressions that describe all occurrences of the block. So the algorithm invokes **SubCut** two times and **PM** once. For each arithmetic progression and the block, the algorithm calculates $\alpha_L, \alpha_R, \gamma_L$ and γ_R. So the algorithm invokes **Period termination** two times. After that it extracts families of squares and adds them to the S-table. Totally at the main step the algorithm invokes **SubCut** at most 16 times, **PM** at most 8 times and **Period termination** at most 64 times. The main step requires $O(|\mathbb{S}|^3 \cdot \log |S|)$ time and $O(|\mathbb{S}|^2)$ space. The algorithm makes $|\mathbb{S}| \cdot \log |S|$ steps. Altogether we get the following theorem:

Theorem 6.4. *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.*

7 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 emphasize main features of the algorithm. The algorithm presents a new technique for SLPs processing. It is divided into independent steps in contrast to classical algorithms in this area which consecutively accumulate information. As a result it can be parallelized. 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 known lower bound. The question whether the upper bound can be lowered to cubic in $|\mathbb{S}|$ remains open.

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

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

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

[6] 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.