Journal of Mathematical Sciences, Vol. 192, No. 3, July, 2013

STRAIGHT-LINE PROGRAMS: A PRACTICAL TEST (EXTENDED ABSTRACT)

I. S. Burmistrov,* A. V. Kozlova,* E. B. Kurpilyansky,* and A. A. Khvorost* UDC 519.256

We present two algorithms that construct a context-free grammar for a given text. The first one is an improvement of Rytter's algorithm that constructs grammars using AVL trees. The second one follows a new approach and constructs grammars using Cartesian trees. Also we compare both algorithms and Rytter's algorithm on various data sets and provide a comparative analysis of the compression ratio achieved by these algorithms and by the LZ77 and LZW algorithms. Bibliography: 15 *titles.*

1. Introduction

Nowadays, search algorithms on huge data sets attract much attention. Since compressed representations are convenient for storing and handling huge data sets, one of the possible ways to process huge volumes of data is to work directly with compressed representations.

Obviously, algorithms that process compressed representations depend on the compression mechanism. There are various compressed representations: collage systems [4], string representations using antidictionaries [11], straight-line programs (SLPs) [9], run-length encoding [1], etc. Text compression based on context-free grammars such as SLPs has become a popular research direction by the following reasons. The first reason is that grammars provide a well-structured compressed representation suitable for data searching. The second one is that the SLP-based compression is polynomially equivalent to the compression achieved by the Lempel–Ziv algorithm, which is widely used in practice. This 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, see [9]. It should also be noted that the classical LZ78 [15] and LZW [13] algorithms can be regarded as special cases of grammar compression. (At the same time, other compression algorithms from the Lempel–Ziv family, such as LZ77 [14] and the run-length encoding, do not fit directly into the grammar compression model.)

There is a wide class of string problems that can be solved in terms of SLPs. This means that the execution time of such an algorithm depends polynomially on the size of the SLP. For example, the class contains the following problems: **Pattern matching** [6], **Longest common substring** [7], **Counting all palindromes** [7], some versions of the problem **Longest common subsequence** [12]. At the same time, constants hidden in the big-O notation for algorithms on SLPs are often very large. Also, the aforementioned polynomial relation between the size of an SLP for a given text and the size of the LZ77 dictionary for the same text does not yet guarantee that SLPs provide a good compression ratio in practice. Thus a major question is whether or not there exist SLP-based compression models suitable for practical applications. This question splits into two subquestions addressed in the present paper: How difficult is it to compress data to an SLP-representation? How large a compression ratio do SLPs provide as compared to classical algorithms used in practice?

Let us describe in more detail the content of the paper and its structure. Section 2 gathers some preliminaries about strings and SLPs. In Sec. 3, we present two SLP construction algorithms. The first one is an improved version of Rytter's algorithm [9]. The second one is a new algorithm that constructs SLP using Cartesian trees. In Sec. 4, we compare the efficiency of SLP construction algorithms and also present the results of comparing the compression ratio for all SLP-based algorithms and some classical compression algorithms. In Sec. 5, we summarize our results.

A part of the results of the present paper related to the improved version of Rytter's algorithm was presented at the 1st International Conference on Data Compression, Communication, and Processing held in Palinuro, Italy, in 2011 (`http://ccp2011.dia.unisa.it/CCP_2011/Home.html`) and was announced in [2].

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 it 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 the positions from left to right by $1, 2, \ldots, |S|-1$.

*Institute for Mathematics and Computer Sciences, Ural State University, Ekaterinburg, Russia, e-mail: burmistrov.ivan@gmail.com, voron13e02@gmail.com, DembelZ@yandex.ru, jaamal@mail.ru.

Translated from *Zapiski Nauchnykh Seminarov POMI*, Vol. 402, 2012, pp. 45–68. Original article submitted May 17, 2012.

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 with $0 \le i \le |S|$, we define $S[i]$ as the character between the positions i and $i+1$ of S. For example, $S[0]$ is the first character of S. The *substring* of S starting at a position ℓ and ending at a position r, $0 \le \ell < r \le |S|$, is denoted by $S[\ell \dots r]$ (in other words, $S[\ell \dots r] = S[\ell] \cdot S[\ell+1] \cdot \ldots \cdot S[r-1]$).

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

$$\mathbb{S}_1 = \exp r_1,\ \mathbb{S}_2 = \exp r_2, \dots, \mathbb{S}_n = \exp r_n,$$

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

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

Thus an SLP is a context-free grammar in Chomsky normal form. Obviously, every SLP generates exactly one string over Σ^+. This string is referred to as the *text* generated by the SLP. For a grammar $\mathbb{S}$ generating a text S, we define the *parse tree* of S as the derivation tree of S in $\mathbb{S}$. We identify terminal symbols with their parents in this tree; after this identification, every internal node has exactly two children.

Figure 1 presents the parse tree of the SLP

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

which derives the 6th Fibonacci word *abaababaabaab*.

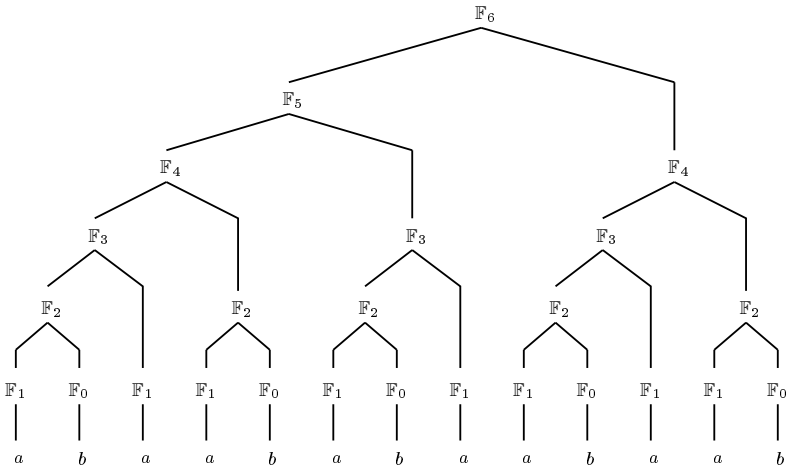


Fig. 1. An SLP that derives *abaababaabaab*.

In this example, the SLP derives a text of length 13 and contains 7 rules. In the general case, the nth Fibonacci word can be derived from the following SLP with $n+1$ rules:

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

Recall that the length of the nth Fibonacci word is equal to the $(n+1)$th Fibonacci number, i.e., the nearest integer to $\frac{\varphi^{n+1}}{\sqrt{5}}$ where $\varphi = \frac{1+\sqrt{5}}{2}$ (the golden ratio). Thus for some texts, their compressed representation using SLPs may be exponentially smaller than the initial text.

In the paper, we adopt the following conventions: every SLP is denoted by a capital blackboard bold letter, for example, $\mathbb{S}$. Every rule of this SLP (and every internal node in its parse tree) is denoted by the same letter with subscripts, for example, $\mathbb{S}_1, \mathbb{S}_2, \ldots$. The *size* of an SLP $\mathbb{S}$ is the number of its rules, and it is denoted by $|\mathbb{S}|$. The *height* of a node in a binary tree is defined as follows. The height of a terminal node (leaf) is equal to 0 by definition. The height of a nonterminal node is equal to 1 + the maximum of the heights of its children. We denote the height of a rule $\mathbb{S}_i$ by $h(\mathbb{S}_i)$.

A *concatenation* of SLPs $\mathbb{S}$ and $\mathbb{S}'$ is an SLP that derives $S \cdot S'$, and it is denoted by $\mathbb{S} \cdot \mathbb{S}'$. We would like to emphasize that concatenation of SLPs is not a rigidly defined operation (like concatenation of strings), since there are various ways to construct an SLP that derives $S \cdot S'$ from the SLPs $\mathbb{S}$ and $\mathbb{S}'$. So a particular way of concatenating SLPs depends on the context of the problem under consideration.

3. SLP construction algorithms

3.1. SLPs, factorizations, and trees. The SLP construction problem can be stated as follows:

Problem: **SLP construction**.
Input: a text S.
Output: an SLP $\mathbb{S}$ that derives S.

The problem of constructing a minimum-size grammar generating a given text is known to be NP-hard [3]. Hence we should look for polynomial-time approximation algorithms. One of the key approaches to such algorithms is to construct a factorization of a given text and to build some binary search tree using it. If we fix some factorization, then at each step an SLP construction algorithm can construct an SLP that derives a particular factor. Next the algorithm concatenates the SLP built at the previous steps with the SLP that derives the particular factor. It is obvious that such an algorithm depends on both the size of the text and the size of the factorization. Hence the SLP construction problem can be reformulated in the following way.

Problem: **SLP construction using factorization**.
Input: a text S and its LZ-factorization $F_1, F_2, \ldots, F_k$.
Output: an SLP $\mathbb{S}$ that derives S.

Rytter in [9] uses a natural factorization generated by the LZ77 compression algorithm as the main factorization. This choice ensures a polynomial relation between the size of an SLP deriving the text S and the size of the LZ77 dictionary for S. Using the properties of the LZ-factorization, we get the following relation: the SLP constructed for a particular factor is contained in the SLP built at the previous steps. This relation substantially increases the efficiency of the construction.

Definition 3.1. *The LZ-factorization of a text S is the decomposition $S = F_1 \cdot F_2 \cdot \ldots \cdot F_k$ where $F_1 = S[0]$ and F_i is the longest prefix of $S[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |S|]$ that occurs as a substring in $F_1 \cdot \ldots \cdot F_{i-1}$, or $S[|F_1 \cdot \ldots \cdot F_{i-1}|]$ if this prefix is empty. The number k is called the* size of the factorization.

There is only one condition on the structure of the parse tree of an SLP: it is a maximal binary tree. This means that every internal node of an SLP has exactly two children (the term is taken from coding theory: it is clear that a binary prefix code is maximal by inclusion if and only if its binary tree is maximal in the above sense). There exist several types of binary trees. Which type is more suitable for the SLP construction problem? The algorithm proposed in [9] uses balanced trees, namely, AVL trees.

Definition 3.2. *An* AVL tree *is a binary tree such that for every nonterminal node, the heights of its children differ at most by* 1.

There is a bound on the height of an AVL tree logarithmic in the number of its nodes, see [5]. It is the main reason why this type of trees is used in Rytter's algorithm. At the same time, the algorithm is nontrivial and resource-intensive. As an alternative, in Sec. 3.4 we consider an algorithm that constructs SLPs using Cartesian trees.

Definition 3.3. A binary search tree *is a binary tree in which every node is assigned a number called a* key *such that the following properties are satisfied:*

- *the left subtree of a node X contains only nodes with keys less than the key of X;*
- *the right subtree of a node X contains only nodes with keys greater than the key of X;*
- *both left and right subtrees are also binary search trees.*

A heap *is a binary tree in which each node is assigned a number called a* priority *and for every node its priority is greater than the priorities of its children.*

A Cartesian tree *is a binary tree in which each node is assigned a pair of numbers: a key and a priority. Thus a Cartesian tree is a binary search tree with respect to keys and a heap with respect to priorities.*

There is a probabilistic estimate on the height of a Cartesian tree logarithmic in the number of its nodes ([10], see Sec. 3.4 below). At the same time, an algorithm for constructing a Cartesian tree spends substantially less time on balancing nodes. It is interesting to compare how the choice of the underlying data structure affects the properties of the SLP returned by the algorithm.

3.2. Rytter's algorithm and its bottleneck. Rytter [9] proved the following theorem.

Theorem 3.1. *Given a string S of length n and its LZ-factorization of length k, one can construct an SLP for S of size $O(k \log n)$ in time $O(k \log n)$.*

The proof of Theorem 3.1 contains an algorithm for constructing an SLP. We recall some key ideas of the algorithm, since they are important for the further discussion.

An *AVL grammar* is an SLP whose parse tree is an AVL tree. The key operation of the algorithm is the concatenation of AVL grammars. The following lemma provides an upper bound on the complexity of this operation.

Lemma 3.2. *Let $\mathbb{S}_1, \mathbb{S}_2$ be two AVL grammars. Then we can construct in time $O(|h(\mathbb{S}_1) - h(\mathbb{S}_2)|)$ an AVL grammar $\mathbb{S} = \mathbb{S}_1 \cdot \mathbb{S}_2$ that derives the text $S_1 \cdot S_2$ by adding only $O(|h(\mathbb{S}_1) - h(\mathbb{S}_2)|)$ nonterminals.*

PROBLEM: **SLP construction using factorization.**
INPUT: a text S and its LZ-factorization $F_1, F_2, \ldots, F_k$.
OUTPUT: an SLP $\mathbb{S}$ that derives S.
RYTTER'S ALGORITHM: The algorithm constructs an SLP by induction on k.

Base. Initially, $\mathbb{S}$ is equal to the terminal rule that derives $S[0]$.

Main loop. Let $i > 1$ be an integer, and assume that an SLP $\mathbb{S}$ that derives $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ has already been constructed. Since the LZ-factorization of S is fixed, an occurrence of F_{i+1} in $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ is known. The algorithm takes a subgrammar of $\mathbb{S}$ that derives F_{i+1} and obtains rules $\mathbb{S}_1, \ldots, \mathbb{S}_\ell$ such that $F_{i+1} = S_1 \cdot S_2 \cdots S_\ell$. Since $\mathbb{S}$ is balanced, we have $\ell = O(\log |S|)$. Using Lemma 3.2, the algorithm concatenates the rules in some specific order (see [9] for details) and sets the next value of $\mathbb{S}$ to be equal to the result of concatenating the previous value of $\mathbb{S}$ with $\mathbb{S}_1 \cdot \ldots \cdot \mathbb{S}_\ell$.

It is well known that maintaining the balance of an AVL tree is quite a difficult task. After adding a new node that breaks the balance of an AVL tree, the modified tree should be rebalanced using a local transformation called a *rotation*. There are two types of rotations. Both are presented in Fig. 2. Every rotation may generate at most three new nodes (such nodes are marked by primes in Fig. 2). Also, every rotation may generate at most three unused rules.

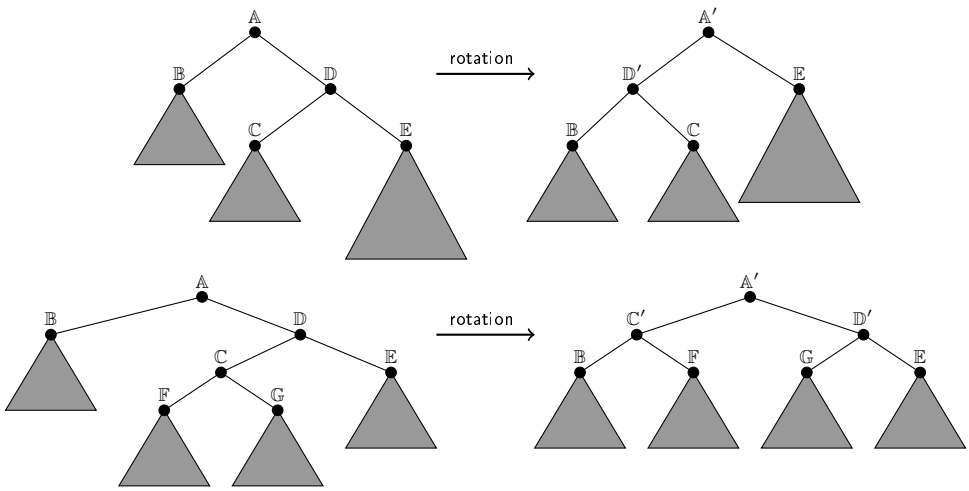


Fig. 2. Two types of rotations of an AVL tree.

It follows from Lemma 3.2 that concatenating two AVL grammars with drastically different heights generates a lot of new nodes. Adding a large number of new nodes to an AVL grammar generates many rotations. In the main loop of Rytter's algorithm, the height of the current AVL grammar $\mathbb{S}$ is constantly growing. At the same time, at each iteration $\mathbb{S}$ concatenates with AVL grammars of relatively small height. The following example shows that the total number of rotations in Rytter's algorithm may be substantially greater than the optimal one.

Example 1. Let $S = a^{2^n} b c^{2^n}$ where n is a fixed integer. Consider the LZ-factorization of S:

$$S = a \cdot a \cdot a^2 \cdot a^4 \cdot \ldots \cdot a^{2^{n-1}-1} \cdot b \cdot c \cdot c \cdot c^2 \cdot c^4 \cdot \ldots \cdot c^{2^{n-1}-1}.$$

Let us denote the factors by $F_1, F_2, \ldots, F_{2n+3}$ in the order they occur in the LZ-factorization. Let $\mathbb{F}_1, \mathbb{F}_2, \ldots, \mathbb{F}_{2n+3}$ be SLPs that correspond to the factors.

Let us estimate the number of rotations that can be generated in the sequence of concatenations $(\ldots((\mathbb{F}_1 \cdot \mathbb{F}_2) \cdot \mathbb{F}_3)\ldots) \cdot \mathbb{F}_{2n+3}$. No rotations are needed to concatenate $\mathbb{F}_1, \mathbb{F}_2, \ldots, \mathbb{F}_{n+1}$, since at each step we concatenate complete binary trees of equal height. So the parse tree of $\mathbb{F}_1 \cdot \mathbb{F}_2 \cdot \ldots \cdot \mathbb{F}_{n+1}$ is a complete binary tree of height n, and the next concatenation $(\mathbb{F}_1 \cdot \mathbb{F}_2 \cdot \ldots \cdot \mathbb{F}_{n+1}) \cdot \mathbb{F}_{n+1}$ generates an AVL tree of height $n+1$. Obviously, each successive concatenation breaks the balance of the current AVL tree and generates at least one rotation. Thus the whole concatenation generates at least $n+1$ and at most $\Theta(n^2)$ rotations (the upper bound follows from the bound on the number of new nodes from Lemma 3.2).

Note that if the algorithm could choose the optimal order of concatenations, namely,

$$((\ldots((\mathbb{F}_1 \cdot \mathbb{F}_2) \cdot \mathbb{F}_3)\ldots) \cdot \mathbb{F}_{n+1}) \cdot ((\ldots((\mathbb{F}_{n+2} \cdot \mathbb{F}_{n+3}) \cdot \mathbb{F}_{n+4})\ldots) \cdot \mathbb{F}_{2n+3}),$$

then it would generate no rotations at all.

One of the possible directions for optimizing Rytter's algorithm is to determine a "good" order of concatenations. Another one is to minimize the number of queries to an AVL grammar. Minimizing the number of queries to AVL grammars becomes important when the size of the input text becomes huge and we cannot store an AVL tree in the memory. Formally, this means that the cost of a query to an AVL tree is greater than the cost of computations using the random access memory. Our next example shows that several factors can be processed together if they occur in a single SLP.

Example 2. Let $n > 0$ be an integer and $S = b \cdot a^{2^{n-1}} \cdot b \cdot a^{2^{n-2}} \cdots b \cdot a$. The length of S is equal to $2^n + n - 2$. Consider the LZ-factorization of S:

$$b \cdot a \cdot a \cdot a^2 \cdot a^4 \cdots a^{2^{n-2}-1} \cdot b a^{2^{n-2}} \cdot b a^{2^{n-3}} \cdots b a.$$

Let $\mathbb{S}_1$ be an SLP that derives $b \cdot a^{2^{n-1}}$. It is obvious that all other factors starting with $b \cdot a^{2^{n-2}}$ occur in S_1. Therefore, one can process them together. So we can construct an SLP $\mathbb{S}_2$ that derives $b \cdot a^{2^{n-2}}$, an SLP $\mathbb{S}_3$ that derives $b \cdot a^{2^{n-3}}$, etc. Finally, we can concatenate the SLPs in the following order: $\mathbb{S}_1 \cdot (\ldots(\mathbb{S}_{n-3} \cdot (\mathbb{S}_{n-2} \cdot \mathbb{S}_{n-1})))$.

3.3. Optimization of Rytter's algorithm. The main ideas of our improved algorithm are to process several factors together and to concatenate each group of factors choosing an optimal order. The intuition behind the algorithm is very simple: if it has already constructed a huge SLP, then most factors occur in the text generated by this SLP and can be processed together.

Modified Rytter's algorithm. Using the input text S and its LZ-factorization $F_1, F_2, \ldots, F_k$, the algorithm constructs an SLP $\mathbb{S}$ that derives S.

Base. Initially, $\mathbb{S}$ is equal to the terminal rule that derives $S[0]$.

Main loop. Let $\mathbb{S}$ be an SLP that derives the text $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ where $0 < i < k$. Let $\ell \in \{1, \ldots, k-i\}$ be the largest integer such that each factor from the set $F_{i+1}, \ldots, F_{i+\ell}$ occurs in $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Since the LZ-factorization is fixed, the value of ℓ can be obtained by a linear search on the factors. SLPs $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \ldots, \mathbb{F}_{i+\ell}$ that derive the texts $F_{i+1}, F_{i+2}, \ldots, F_{i+\ell}$ can be computed by an application of the subgrammar cutting algorithm (analogously to [9]).

Next, the algorithm concatenates $\mathbb{F}_{i+1}, \ldots, \mathbb{F}_{i+k}$. It optimizes the order of concatenations using dynamic programming. Let $\varphi(p, q)$ be the function that is calculated by the following recurrence formula:

$$\varphi(p,q) = \begin{cases} 0 & \text{if } p = q, \\ \min_{r=p}^{q}(\varphi(p,r) + \varphi(r+1,q) + |\log(|f_{i+p}| + \cdots + |f_{i+r}|) & \\ \quad - \log(|f_{i+r+1}| + \cdots + |f_{i+q}|)|) & \text{otherwise.} \end{cases}$$

The value $\varphi(p, q)$ is proportional to the upper bound on the number of rotations of a grammar tree that are performed during the concatenation of $\mathbb{F}_p, \mathbb{F}_{p+1}, \ldots, \mathbb{F}_q$. The upper bound follows from Lemma 3.2 and from the estimate on the height of an AVL tree from [5]. Typically, the upper bound is too large. So it is more correct to regard the function $\varphi(p, q)$ as a heuristic using which the algorithm obtains "good" groups of factors.

The algorithm fills an $\ell \times \ell$ table with the values $\varphi(p,q)$, $1 \le p,q \le \ell$. In the case where $p < q$, it additionally stores the integer $r \in \{p, p+1, \ldots, q-1\}$ on which the minimum of the following expression is reached:

$$\varphi(p,r) + \varphi(r+1,q) + \bigl|\log(|F_{i+p}| + \ldots + |F_{i+r}|) - \log(|F_{i+r+1}| + \ldots + |F_{i+q}|)\bigr|.$$

The order of filling out the table is as follows: all cells (p,q) such that $p \ge q$ are set to be equal to 0, next the algorithm fills the cells such that $q-p=1$, next it fills the cells such that $q-p=2$, etc. Thus the algorithm does not recompute recursively the values $\varphi(p,r)$ and $\varphi(r+1,q)$, since they already exist in the table. So every single value $\varphi(p,q)$ can be calculated in time $O(k)$. Figure 3 presents the pseudo-code of the corresponding procedure. Thus the algorithm fills out the table using time $O(\ell^3)$ and space $O(\ell^2)$.

```
result = +∞;
L = 0, R = |F_{i+p}| + ··· + |F_{i+q}|;
for (int r = p; r < q; r++;) {
    L+ = |F_{i+r}|;
    R− = |F_{i+r}|;
    tmp = φ(p,r) + φ(r+1,q) + |log L − log R|;
    if (tmp < result)
        result = tmp;
}
```

Fig. 3. A pseudo-code that computes the value $\varphi(p,q)$.

Finally, the algorithm reads the value of r from the cell $(1,\ell)$ and determines the order of concatenations for $\mathbb{F}_{i+1}, \ldots, \mathbb{F}_{i+k}$ in time $O(\ell)$. Using this order, it constructs an SLP $\mathbb{F}$ that derives $F_{i+1} \cdot F_{i+2} \cdot \ldots \cdot F_{i+l}$. Finally, the algorithm concatenates $\mathbb{S}$ and $\mathbb{F}$ and sets $\mathbb{S}$ to be equal to $\mathbb{S} \cdot \mathbb{F}$.

Theorem 3.3. *Let $f_1, f_2, \cdots, f_k$ be the LZ-factorization of a text w. The above algorithm constructs an SLP for w of size $O(k \log n)$.*

Proof essentially repeats the corresponding part of the proof of Theorem 3.1, but we reproduce it for the sake of completeness.

Let us prove the theorem by induction on the number of factors. The base is clear.

Assume that an SLP $\mathbb{S}$ that derives the text $F_1 \cdot F_2 \cdot \ldots \cdot F_i$, where $0 < i < k$, is already built and has size $O(i \log |F_1 \cdot F_2 \cdot \ldots \cdot F_i|) = O(i \log n)$. Let $F_{i+1}, \ldots, F_{i+\ell}$ be the next factors that occur in $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Let us consider subgrammars $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \ldots, \mathbb{F}_{i+\ell}$ of $\mathbb{S}$ that derive the texts $F_{i+1}, F_{i+2}, \ldots, F_{i+\ell}$, respectively. The height of $\mathbb{F}_{i+j}$ is not greater than $1.4404 \log |F_{i+j}|$, see [5]. Hence, by Lemma 3.2, the number of new rules that the algorithm adds at each step of constructing an SLP $\mathbb{F}$ that derives $F_{i+1} \cdot F_{i+2} \cdot \ldots \cdot F_{i+\ell}$ is at most $O\left(\log |F_{i+1}| + \log |F_{i+2}| + \cdots + \log |F_{i+\ell}|\right) = O(\log n)$. Each rotation of an AVL grammar generates at most three new rules. The total number of rules in the SLP $\mathbb{F}$ that are absent in the SLP $\mathbb{S}$ is at most $O(\ell \log n)$. Analogously, the number of new rules that the algorithm adds during the concatenation of $\mathbb{S}$ and $\mathbb{F}$ is $O(\log n)$. Hence the size of the SLP $\mathbb{S} \cdot \mathbb{F}$ that derives the text $F_1 \cdot F_2 \cdot \ldots \cdot F_{i+\ell}$ is $O((i+\ell) \log n)$.

The time complexity of the modified Rytter's algorithm cannot be less than the complexity of the original algorithm from [9], since the latter is a special case of the modification described above when all groups are of size 1. On the one hand, the new algorithm generates less rotations, but on the other hand, it spends some extra time on calculating the order of concatenations. The cumulative influence of both factors on the execution time is unclear. In the next section, we propose a practical comparison of the algorithms under discussion.

3.4. SLP construction using Cartesian trees. As we have already noticed, SLP construction algorithms that use AVL trees spend a lot of time on balancing. We think that the following idea may be useful for solving the SLP construction problem: to replace the data structure used for representing SLPs with another one that would allow the algorithm to spend less time on balancing. In this section, we present an algorithm that construct SLPs using Cartesian trees.

There is a probabilistic bound on the height of a Cartesian tree that is logarithmic in the total number of nodes (see [10]). Namely, if the priorities of nodes are chosen at random, independently, and with the same distribution, then the expected height of a Cartesian tree with n nodes is $O(\log n)$. Also, for every fixed constant

c with $c > 1$, the probability that the height of a Cartesian tree with n nodes is greater than $2c \ln n$ is bounded by $n \left(\frac{n}{e}\right)^{-c \ln(c/e)}$.

To construct an SLP from an LZ-factorization, we need two operations: cutting a subtree with specified positions and concatenating two trees. For a Cartesian tree, it is easy to implement the following operations: *split* is the operation of splitting a tree into two subtrees with a specified position, and *merge* is the operation of merging two trees. But the standard implementation of the *merge* operation requires the following condition: every key of the first tree should be less than any key of the second tree. Hence it is necessary to regenerate the keys of the tree obtained after applying the *split* operation. This situation appears in the main loop of the SLP construction algorithm. After the algorithm has constructed a tree T that derives a prefix of the input text, it cuts a subtree T' of T that derives the next factor and applies the *merge* operation to T and T'. Therefore, the algorithm should completely regenerate the keys of T' before merging T and T'. To make this operation efficient, it is profitable to avoid explicitly storing keys. Next we explain why it is possible.[1]

Let T be an arbitrary Cartesian tree, and assume that the information about its keys has been lost. One can recover the linear order relation on the keys using only the tree structure. The recovering algorithm recursively traverses the tree in the following order: the left subtree, the root, the right subtree. The number of the current node in this order is greater by one than the number of nodes in the subtree that the algorithm has visited before visiting the current node. Therefore, we are able to avoid explicitly storing the keys.

Definition 3.4. *A* Cartesian tree with implicit keys *is a Cartesian tree that does not store the information about keys.*

In what follows, we assume that the key of a node of a Cartesian tree T with implicit keys is equal to the number of the key in the linear order on all keys of T. We denote the subtree of T with the root at a node T_i by $\overline{T_i}$, and the total number of nodes in this subtree, by $\mathrm{count}(T_i)$. If T_ℓ and T_r are the left and right children of T_i, respectively, then we use the following short notation for this fact: $T_i = (T_\ell, T_r)$. It may happen that the nodes T_ℓ and/or T_r are empty. For example, if T_i is a leaf, then both T_ℓ and T_r are empty.

Let us describe an implementation of the *split* and *merge* operations for Cartesian trees with implicit keys.

The *split* operation. The input is a Cartesian tree T with implicit keys and a positive integer k where $k \le |T|+1$. The output is a pair of Cartesian trees L and R with implicit keys such that L contains all nodes of T with keys less than k and R contains all the other nodes of T. By definition, the operation produces two empty trees on the input (empty tree, 1).

The algorithm starts from the root T_0 of T and works recursively. The following cases can occur:

(S1) If $k \le \mathrm{count}(T_\ell) + 1$, then T_0 lies in R and the algorithm splits the subtree $\overline{T}_\ell$. Assume that the *split* operation returns two trees L' and R' on the input $(\overline{T}_\ell, k)$. Then the algorithm returns $L = L'$ and $R = (R', \overline{T}_r)$.

(S2) If $k > \mathrm{count}(T_\ell) + 1$, then T_0 lies in L and the algorithm splits the subtree T_r. Assume that the *split* operation returns two trees L' and R' on the input $(\overline{T}_r, k - \mathrm{count}(T_\ell) - 1)$. Then the algorithm returns $L = (\overline{T}_\ell, L')$ and $R = R'$.

We would like to emphasize that at each node T_i the algorithm stores the number $\mathrm{count}(T_i)$. Since at every step, the algorithm either terminates or recursively calls the *split* operation with a subtree of smaller height, the time complexity of the algorithm is $O(\log |T|)$.

Since the parse tree of an SLP is a maximal binary tree, we should modify the *split* operation to guarantee that the resulting trees are maximal. To achieve this aim, it suffices to delete all nodes that have exactly one child from both output trees. Formally, if a node T_j has a single child T_k, then we delete T_j from the tree. If T_j is the root, then we choose T_k as the new root after deleting T_j. The priorities of nodes do not change.

Obviously, if the input tree T is maximal, then at each step of the algorithm, in each output tree L or R there is at most one node with a single child. Thus the time complexity of "maximizing" both trees L and R is $O(\log |T|)$. In fact, a practical implementation of the maximization procedure does not require a separate pass through the output, since it can be integrated into the algorithm. In what follows, by the *split* operation we mean its modified version that returns maximal trees.

[1] Unfortunately, the elegant idea of a Cartesian tree without explicitly stored keys has not yet been considered in the academic literature. A rather complete account of this idea is presented in the Internet publication [8] in Russian. We know for a certainty that it was first applied at an ACM programming contest in 2002 by N. V. Dourov and A. S. Lopatin (members of the student team of the St. Petersburg State University).

The *merge* operation. The input is two Cartesian trees T' and T'' with implicit keys. The output is a Cartesian tree T with implicit keys that contains all nodes from both T' and T''. By definition, if T' is empty, then the operation returns T'', and vice versa, if T'' is empty, then the operation returns T'.

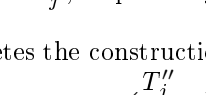
The algorithm starts from the roots T'_0 and T''_0 of the trees T' and T'', respectively, and works recursively. Let $T'_0 = (T'_\ell, T'_r)$ and $T''_0 = (T''_k, T''_q)$. Since the priorities of all nodes were chosen at random and independently, we suppose that they are pairwise distinct. The following two cases can occur:

(M1) If the priority of the node T'_0 is greater than that of the node T''_0, then the algorithm chooses T'_0 as the root of T. The left subtree is equal to $\overline{T}'_\ell$, and the right subtree is equal to the tree returned by the *merge* operation on the input $(\overline{T}'_r, T'')$.

(M2) If the priority of the node T'_0 is less than that of the node T''_0, then the algorithm chooses T''_0 as the root of T. The right subtree is equal to $\overline{T}''_q$, and the left subtree is equal to the tree returned by the *merge* operation on the input $(T', \overline{T}''_k)$.

Since at each step of the recursion, the algorithm walks down either the left subtree or the right subtree, its expected execution time is $O(\log|T'| + \log|T''|)$.

As in the case of the *split* operation, we should modify the *merge* operation in order to use it in the SLP construction. There are two problems. The first one is that we should guarantee that the resulting tree is a maximal binary tree. The second one is that we should guarantee that the array of leaves of T is the concatenation of the array of leaves of T' and the array of leaves of T''. Both problems can be solved using the following simple modification of the algorithm.

Let T'_i be the rightmost leaf of T' and T''_j be the leftmost leaf of T''. Note that the extreme leaves in Cartesian trees with implicit keys are defined unambiguously. Let y' and y'' be the priorities of T'_i and T''_j, respectively. Let $y_* = \min(y', y'')$ and $y^* = \max(y', y'')$. We set the priorities of T'_i and T''_j to be equal to y_*. Applying rules (M1) and (M2), the algorithm eventually reaches a configuration with the current roots T'_0 and T''_0 equal to the leaves T'_i and T''_j, respectively. At this moment, the algorithm adds the new node $U = (T'_i, T''_j)$ with priority y^* and completes the construction of the tree T by adding the three-element subtree  instead of a two-element subtree (T'_i  or  T''_j). Clearly, the modified algorithm takes the same time $O(\log|T'| + \log|T''|)$ as the standard algorithm. It is easy to check that if the trees T' and T'' are maximal, then the output tree T is maximal too. Moreover, T is a concatenation of T' and T'' in the sense of SLPs, see Sec. 2. In what follows, by the *merge* operation we mean its modified version that returns a maximal tree.

We say that an SLP is a *Cartesian SLP* if its parse tree is a Cartesian tree with implicit keys. Now we introduce an algorithm for constructing a Cartesian SLP.

Algorithm for constructing a Cartesian SLP
Input: a text S and its LZ-factorization F_1, F_2, ..., F_k.
Output: a Cartesian SLP that derives S.

Base. Initially, S is equal to the terminal rule that derives $F_1 = S[0]$.

Main loop. Assume that a Cartesian SLP $\mathbb{S}$ that derives the text $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ has already been constructed for a fixed integer i where $i > 1$. The factor F_{i+1} occurs in the text $S = F_1 \cdot F_2 \cdot \ldots \cdot F_i$ by the definition of the LZ-factorization. Let ℓ and r be positions in S such that $F_{i+1} = S[\ell \ldots r]$. Let ℓ^* and r^* be the priorities of the leaves $S[\ell]$ and $S[r]$ in $\mathbb{S}$, respectively. Since the algorithm stores $\mathrm{count}(\mathbb{S}_i)$ in each node $\mathbb{S}_i$, the values of ℓ^* and r^* can easily be computed from ℓ and r.

The algorithm invokes the *split* operation with the input $(\mathbb{S}, \ell^*)$. Let R be the rightmost tree in the output. Next the algorithm invokes the *split* operation with the input $(R, r^* - \ell^*)$. The leftmost tree in the output is a Cartesian SLP $\mathbb{F}$ that derives F_{i+1}. Finally, the algorithm invokes the *merge* operation with $\mathbb{S}$ and $\mathbb{F}$, and the output is a Cartesian SLP that derives $F_1 \cdot F_2 \cdot \ldots \cdot F_{i+1}$.

Theorem 3.4. *The expected execution time of the presented algorithm on a text S of length n and its LZ-factorization of size k is $O(k \log n)$. The expected size of the SLP returned by the algorithm is $O(k \log n)$.*

Proof. At each step, the algorithm applies at most two *split* operations and at most one *merge* operation. It follows that the expected execution time of every step is $O(\log n)$. Since the algorithm consists of exactly k steps, its expected execution time is $O(k \log n)$.

At every step of each operation (*split* or *merge*), the algorithm generates one new nonterminal rule. Since the time complexity of each operation is $O(\log n)$ and the operations are invoked $3k$ times in total, the expected size of the output SLP is $O(k \log n)$. □

4. Practical results

4.1. The setup of the experiments. Obviously, the nature of input strings highly affects the compression time and compression ratio. In this paper, we consider three types of strings:

- DNA sequences (downloaded from the DNA Data Bank of Japan, `http://www.ddbj.nig.ac.jp`);
- Fibonacci strings;
- random strings over a four-letter alphabet.

These types of strings were chosen for the following reasons. Fibonacci strings are known to be one of the best inputs to the SLP construction problem. Thus they allow us to estimate the potential of SLPs as a compression model. Random strings are considered to be incompressible, and, potentially, they are the worst input to the SLP construction problem. DNA sequences form a class of well-compressed strings widely used in practice.

We compare the SLP construction algorithms presented in Sec. 3 with classical compression algorithms from the Lempel–Ziv family. Our test suite contains two implementations of the Lempel–Ziv algorithm [14]: an algorithm with small (32Kb) searching window and an algorithm with infinite searching window. The test suite also contains an implementation of the Lempel–Ziv–Welch algorithm [13]. The source code is available at `http://code.google.com/p/overclocking/`. All algorithms were run in the same environment on a PC with the following characteristics: Intel Core i7-2600, 3.4GHz, 8Gb operational memory, OS Windows 7 x64.

4.2. The experimental results. As expected, all SLP construction algorithms work infinitely fast on Fibonacci strings and construct extremely compact representations. For example, on the 35th Fibonacci word of size 36.9Mb, the algorithms return the answer within 1ms and build SLPs of size 100.

Figures 4–7 present the main experimental results on random strings and DNA sequences. For convenience, we adopt the following notation for algorithms:

□ – Lempel-Ziv algorithm with 32Kb search window;

■ – Lempel-Ziv algorithm with infinite search window;

△ – Lempel-Ziv-Welch algorithm;

○ – Rytter's algorithm from [9];

● – modified version of Rytter's algorithm from Sec. 3.3;

▲ – Cartesian SLP construction algorithm from Sec. 3.4.

The performance of a compression algorithm is estimated in terms of the compression ratio and execution time. We calculate the compression ratio as the ratio of the size of the compressed presentation to the size of the input text, measuring it in per cent. For example, the formula for the SLP compression ratio looks like $\frac{|\mathbb{S}|}{|S|} \cdot 100$. We also calculate the number of rotations for SLP construction algorithms that use AVL trees.

Figure 4 shows how the suggested modification of Rytter's algorithm affects the number of rotations. Obviously, the modified algorithm uses substantially less rotations on texts of length more than 10Mb. Figure 4 shows that the suggested heuristic is efficient. It is very interesting that the number of rotations depends regularly on the size of the input text, while the execution time depends weakly on the nature of the input text for all algorithms. We have no theoretical explanation of these observations.

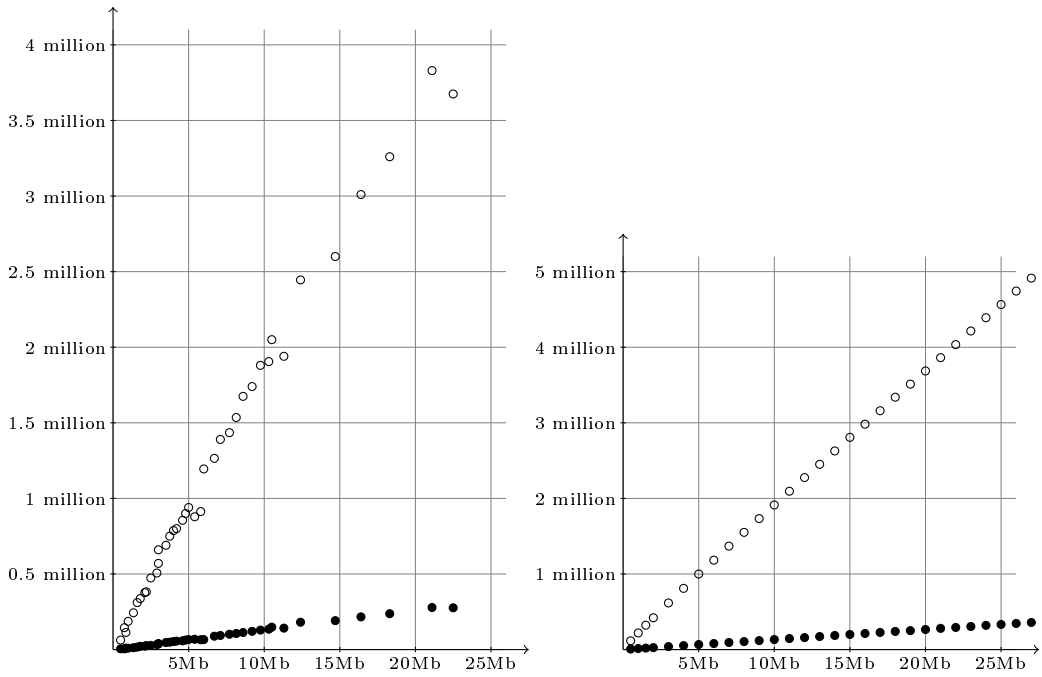


Fig. 4. The statistics of AVL rotations on DNA sequences (left) and on random strings (right).

As discussed in Sec. 3.3, a gain in the number of rotations does not guarantee a gain in the speed of constructing an SLP, since the modified algorithm spends extra time on calculating the optimal order of concatenations. We compare the speed of all SLP construction algorithms using the following two tests. In the first one, the algorithms stored all SLPs being constructed in the random access memory, while in the second one, SLPs were stored in an external file, so that every rotation of an AVL tree forced I/O operations with the file. Figures 5 and 6 present the results of both tests on DNA sequences and random strings, respectively. It follows from the experimental results that the modified algorithm from Sec. 3.3 works several times faster than Rytter's algorithm. The modified algorithm works two times faster on random strings and three times faster on DNA sequences if SLPs are stored in the random access memory. Also, it works five time faster on DNA sequences and three times faster on random strings if SLPs are stored in a file system. The algorithm that use Cartesian trees works faster than Rytter's algorithm, but slower than the modified algorithm. The reason is that the heights of the constructed Cartesian trees are substantially larger than the heights of the corresponding AVL trees. The experimental results show that the average height of an AVL tree is equal to 21.8 and the average height of a Cartesian tree is equal to 47.8. Thus the Cartesian SLP construction algorithm processes more rules than the algorithms using AVL trees. This cancels the gain achieved from the simplicity of maintaining the balance in Cartesian trees.

Figure 7 presents the experimental results for the compression ratio achieved by SLP construction algorithms and by classical compression algorithms from the Lempel–Ziv family. We see that the algorithms using AVL trees achieve similar values of the compression ratio, which are twice less on the average than the compression ratio achieved by the LZW algorithm. It is interesting that the ratio of the compression ratios achieved by the algorithms using AVL trees to the compression ratio achieved by the LZW algorithm does not depend on the type and length of the input text. The compression ratio of the algorithm that uses Cartesian trees is substantially worse than the compression ratio of the other algorithms. In this case, we also observe that the ratio of the compression ratios weakly depends on the type and length of the input text.

5. Conclusion

Our experimental results show that both Rytter's algorithm and the modified algorithm achieve the same compression ratio. But the running time of the second algorithm is substantially smaller. Since using a file system is inevitable with the growth of the input, it is worth noticing that the modified algorithm is more stable with respect to the growth of the input than Rytter's algorithm.

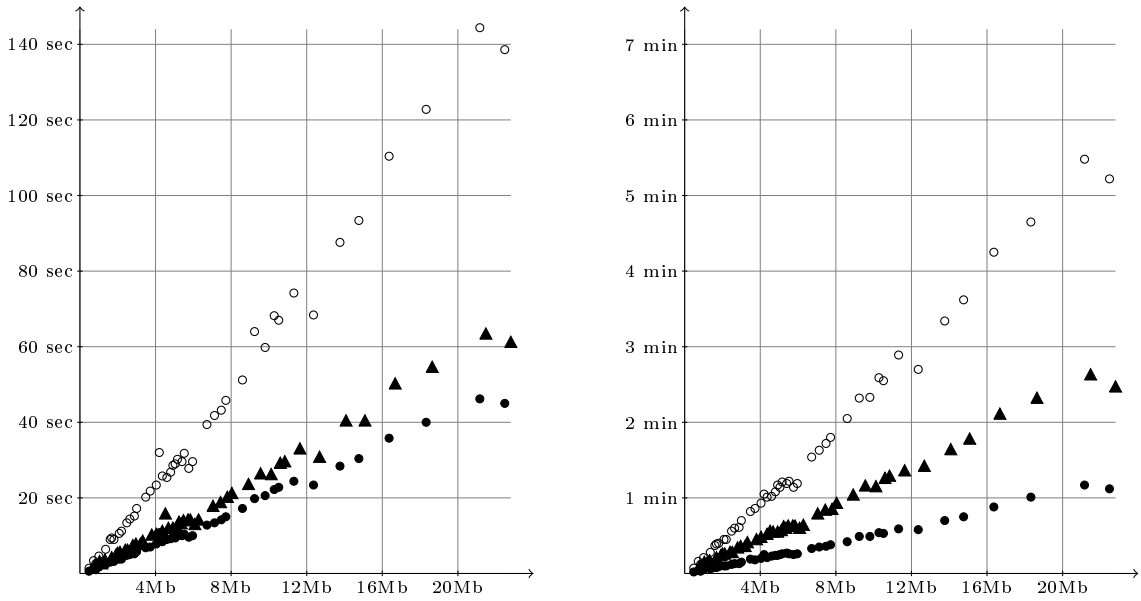


Fig. 5. The SLP construction time on DNA sequences when SLPs stored in the random access memory (left) and in an external file (right).

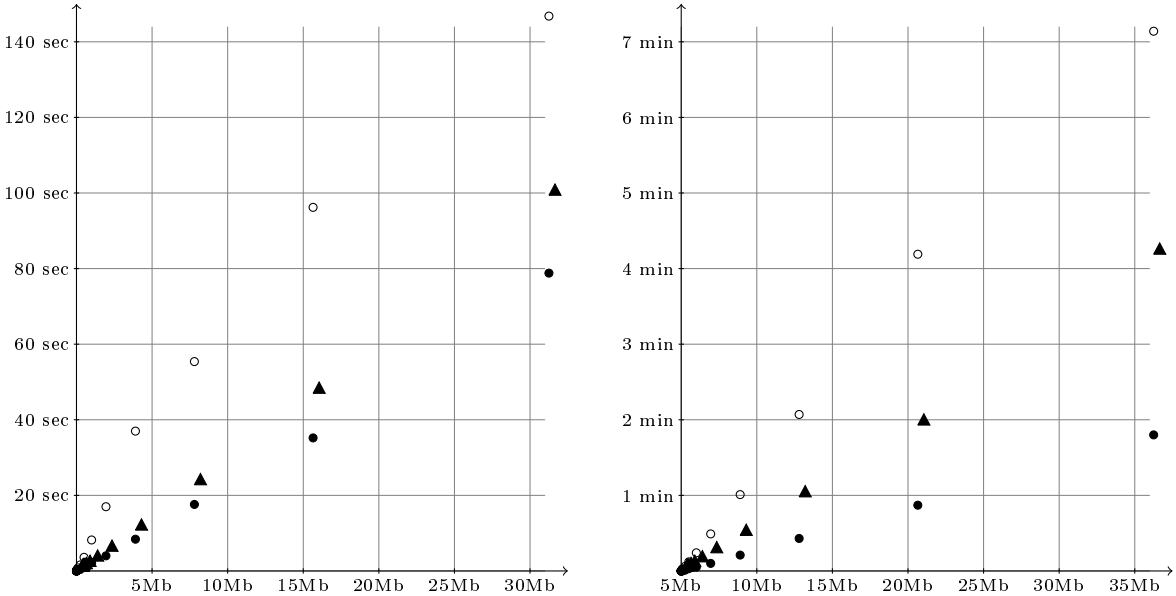


Fig. 6. The SLP construction time on random strings when SLPs are stored in the random access memory (left) and in an external file (right).

In the paper, we present a Cartesian SLP construction algorithm. This algorithm has a similar execution time compared to the other discussed SLP construction algorithms, but provides a substantially worse compression ratio and the height of the output tree. This fact is important for searching algorithms that work directly with compressed representations. Thus our aim to improve the performance of SLP construction using an efficient data structure was not achieved. Now we think that this aim is hard to achieve. It appears that searching for new heuristics based on AVL trees that allow one to construct more compact SLPs is a more productive idea.

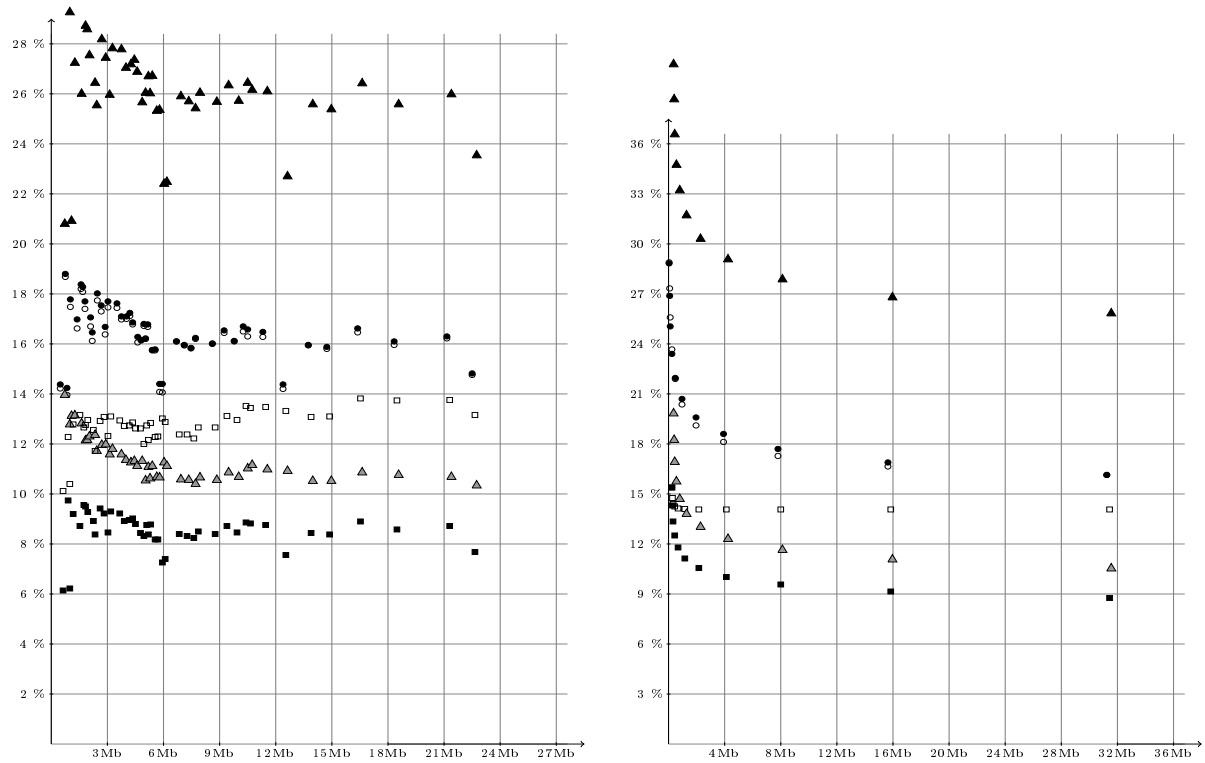


Fig. 7. The compression ratio achieved on DNA sequences (left) and on random strings (right).

All tested SLP construction algorithms are worse than the classical compression algorithms from the Lempel–Ziv family both in the achieved compression ratio and the execution time. SLP construction algorithms are of interest (at least from the theoretical point of view), since they provide a well-structured data representation that allows one to solve some classical searching problems without decompressing. However, the question on what volumes of input data SLP searching algorithms will be more efficient than classical string searching algorithms is still open. We think that it is one of the main research directions in this area.

Acknowledgments

The authors would like to thank Professor Mikhail V. Volkov for his critical notes and continuous support. The authors would like to thank the anonymous referee for his remarks and suggested improvements to the original version of the paper.

The authors acknowledge support from the Russian Foundation for Basic Research, grant 10-01-00793.

Translated by the authors.

REFERENCES

1. A. Apostolico, G. M. Landau, and S. Skiena, "Matching for run-length encoded strings," *J. Complexity*, **15**, 4–16 (1999).
2. I. Burmistrov and L. Khvorost, "Straight-line programs: a practical test," in: *Procedings of the First International Conference on Data Compression, Communications and Processing* (CCP) (2011), pp. 76–81.
3. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," IEEE *Trans. Inform. Theory*, **51**, 2554–2576 (2005).
4. T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa, "Collage system: a unifying framework for compressed pattern matching," *Theoret. Comput. Sci.*, **298**, 253–272 (2003).

5. D. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, 2nd edition, Addison-Wesley (1998).
6. Y. Lifshits, "Processing compressed texts: A tractability border," *Lect. Notes Comput. Sci.*, **4580**, 228–240 (2007).
7. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto, "Computing longest common substring and all palindromes from compressed strings," *Lect. Notes Comput. Sci.*, **4910**, 364–375 (2008).
8. A. Polozov, "Cartesian tree. Part 3: Cartesian tree with implicit keys," blog post, `http://habrahabr.ru/blogs/algorithm/102364/`.
9. W. Rytter, "Application of Lempel–Ziv factorization to the approximation of grammar-based compression," *Theoret. Comput. Sci.*, **302**, 211–222 (2003).
10. R. Seidel and C. Aragon, "Randomized search trees," *Algorithmica*, **16**, 464–497 (1996).
11. Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa, "Pattern matching in text compressed by using anti-dictionaries," *Lect. Notes Comput. Sci.*, **1645**, 37–49 (1999).
12. A. Tiskin, "Faster subsequence recognition in compressed strings," *J. Math. Sci.*, **158**, 759–769 (2009).
13. T. Welch, "A technique for high-performance data compression," *Computer*, **17**, 8–19 (1984).
14. J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, **23**, 337–343 (1977).
15. J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inform. Theory*, **24**, 530–536 (1978).