Straight-line Programs: A Practical Test (Extended Abstract)*

I. Burmistrov L. Khvorost A. Kozlova E. Kurpilyansky

Abstract

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 is a new approach that constructs grammars using Cartesian trees. Also we compare both algorithms and Rytter's algorithm on various data sets and provide a comparative analysis of compression ratio achieved by these algorithms, by LZ77, and by LZW.

§1. Introduction

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

Obviously algorithms that process compressed representations depend on the way of compression. 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 one is that grammars provide well-structured compressed representation that is 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 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, see [9]. It should also be noted that classical LZ78 [15] and LZW [13] algorithms can be considered as special cases of grammar compression. (At the same time other compression algorithms from the Lempel-Ziv family—such as LZ77 [14] and 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. It means that the execution time of such an algorithm polynomially

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

depends on size of 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 big-O notation for algorithms on SLPs are often very big. 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 good compression ratio in practice. Thus, a major question is whether or not there exist SLP-based compression models suitable to practical usage? This question splits into two sub-questions addressed in the present paper: How difficult is it to compress data to an SLP-representation? How large compression ratio do SLPs provide as compared to classic 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 Section 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 Section 4 we compare efficiency of SLP construction algorithms and also present results of a comparison of compression ratio between all SLP-based algorithms and some classic compression algorithms. In Section 5 we summarize our results.

A part of the results of the present paper related to an improve 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 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 $0 \le i \le |S|$ 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. A *substring* of S starting at the position ℓ and ending at the position r, $0 \le \ell < r \le |S|$, is denoted by $S[\ell \ldots r]$ (in other words $S[\ell \ldots r] = S[\ell] \cdot S[\ell + 1] \cdot \ldots \cdot S[r - 1]$).

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 character 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. 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 following SLP

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

that derives the 6th Fibonacci word *abaababaabaab*.

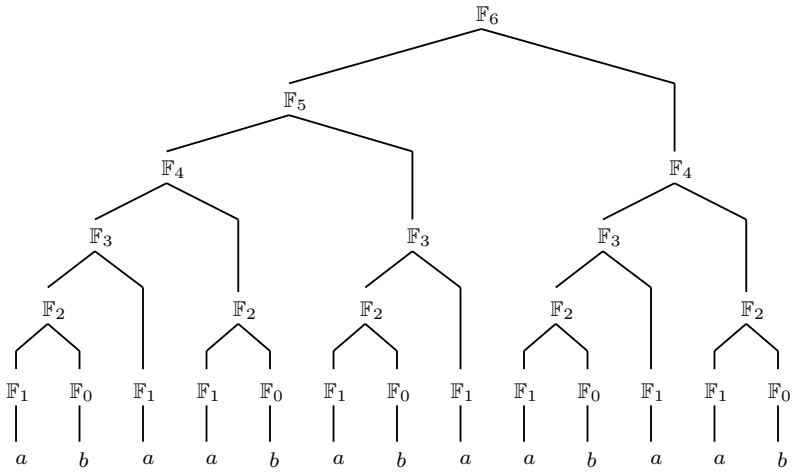


Figure 1: An SLP that derives *abaababaabaab*

In the example the SLP derives a text of length 13 and contains 7 rules. In 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). So for some texts its compressed representation using SLPs may be exponentially smaller than the intial text.

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 (and every internal node in its parse-tree) is denoted by the same letter with indices, for example, $\mathbb{S}_1, \mathbb{S}_2, \ldots$. The *size* of an SLP $\mathbb{S}$ is the number of its rules and 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 the rule $\mathbb{S}_i$ by $h(\mathbb{S}_i)$.

The *concatenation* of SLPs $\mathbb{S}$ and $\mathbb{S}'$ is an SLP that derives $S \cdot S'$ and is denoted by $\mathbb{S} \cdot \mathbb{S}'$. We would like to emphasize that the concatenation of SLPs

is not a rigidly defined operation (unlike 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 concatenation of 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 minimal 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 the factorization. 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 that was built at previous steps with the SLP that derives the particular factor. It is obvious that such an algorithm depends on both size of the text and 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 LZ77 compression algorithm as the main factorization. This choice ensures a polynomial relation between size of an SLP deriving text S and size of the LZ77 dictionary for S. Using LZ-factorization properties we get the following relation: the SLP constructed for a particular factor is contained in the SLP that was built at previous steps. This relation essentially increases construction efficiency.

Definition 3.1. The LZ-factorization of a text S is given by a 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 $S[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |S|]$ which occurs as a substring in $F_1 \cdot \ldots \cdot F_{i-1}$ or $S[|F_1 \cdot \ldots \cdot F_{i-1}|]$ in case this prefix is empty. The number k is *size of factorization*.

There is only one condition on the structure of SLPs parse-tree: it is a maximal binary tree. It means that every internal node of an SLP has exactly two children (the term is taken from the coding theory: it is clear that a binary prefix code is a maximal with respect to 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 non-terminal node the heights of its children can differ at most by 1.

There is a logarithmic bound on the height of an AVL tree depending on 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 alternative we consider the algorithm that constructs SLPs using Cartesian trees in Section 3.4.

Definition 3.3. *Binary search tree* is a binary tree such that each node holds a number named a *key* and the keys have the following properties:

- the left subtree of a node contains only nodes with keys less than the node's key.
- the right subtree of a node contains only nodes with keys greater than the node's key.
- both the left and the right subtrees must also be binary search trees.

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

A *Cartesian tree* is a binary tree in which each node holds a pair of numbers (key, priority). So a Cartesian tree is a binary search tree with respect to keys and a heap with respect to priorities.

There is a probabilistic logarithmic estimation on the height of a Cartesian tree depending on the number of its nodes ([10], see Section 3.4). At the same time a Cartesian tree construction algorithm spends essentially less time on balancing of nodes. It is interesting to compare how the choice of maintaining the underlying data structure affects 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 of SLP construction. We remind here some key ideas of the algorithm as 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 concatenation of AVL-grammars. The following lemma provides an upper bound for the operation:

Lemma 3.2. *Assume $\mathbb{S}_1, \mathbb{S}_2$ are two nonterminals of AVL-grammars. Then we can construct in $O(|h(\mathbb{S}_1) - h(\mathbb{S}_2)|)$ time 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.

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 the 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 balance of an AVL tree is quite a complex operation. After adding a new node that breaks balance of an AVL-tree the modified tree should be rebalanced using local transformation named *rotation*. There are two types of rotations. Both are presented in Figure 2. Each rotation may generate at most three new nodes (nodes marked by dashes in Figure 2). Also every rotation may generate at most three unused rules.

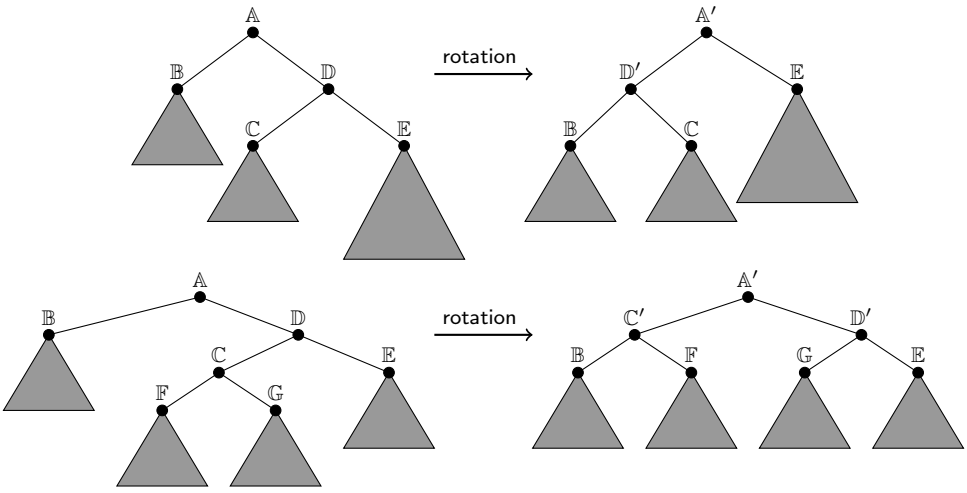


Figure 2: Types of rotations of an AVL-tree

It follows from Lemma 3.2 that concatenation of two AVL-grammars with drastically different heights generates a lot of new nodes. Adding a big number of new nodes to an AVL-grammar generates many rotations. In the main loop of Rytter's algorithm height of current AVL-grammar $\mathbb{S}$ grows permanently. In 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 essentially greater than optimal.

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 order of their occurrence 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 may be generated in the course of the sequence of concatenations $(\dots((\mathbb{F}_1\cdot\mathbb{F}_2)\cdot\mathbb{F}_3)\dots)\cdot\mathbb{F}_{2n+3}$. No rotations are needed to concatenate $\mathbb{F}_1, \mathbb{F}_2, \dots, \mathbb{F}_{n+1}$ since at each step we concatenate full 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 full 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 the AVL-tree of height $n+1$. Obviously each following concatenation breaks balance of the current AVL-tree and generates at least one rotation. So total 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).

Notice that if the algorithm could choose the optimal order of concatenations namely

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

then the algorithm would generate no rotations at all.

One of possible directions for an optimization of Rytter's algorithm is to determine "good" concatenation order. Another direction for an optimization consists of minimizing the number of queries to an AVL-grammar. Minimizing of the number of queries to AVL-grammars becomes important when the size of input text becomes huge and we cannot store an AVL-tree in the memory. Formally it means that costs of a query to an AVL-tree are greater than costs of calculations in 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:

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

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 it is possible to process them together. So we can construct SLPs $\mathbb{S}_2$ that derives $b\cdot a^{2^{n-2}}$, $\mathbb{S}_3$ that derives $b\cdot a^{2^{n-3}}$, etc. Finally we can concatenate the SLPs in the following order: $\mathbb{S}_1\cdot(\dots(\mathbb{S}_{n-3}\cdot(\mathbb{S}_{n-2}\cdot\mathbb{S}_{n-1})))$.

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 the algorithm has already constructed a huge SLP, then most factors occur in the text generated by this SLP and may be processed together.

Modified Rytter's algorithm Using input text S and its LZ-factorization $F_1, F_2, \dots, F_k$ 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 $\ell\in\{1,\dots,k-i\}$ be a maximal integer such that each factor from

the following set $F_{i+1}, \ldots, F_{i+\ell}$ occurs in $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Since LZ-factorization is fixed, the value of ℓ can be obtained by a linear search on factors. The 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 concatenation order using dynamic programming. Let $\varphi(p, q)$ be the function that is calculated by the following recurrent formula:

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

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

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

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

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

```
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;
}
```

Figure 3: Pseudo code that computes value of $\varphi(p, q)$

Finally the algorithm reads the value of r from the cell $(1, \ell)$ and determines the order of concatenations for $\mathbb{F}_{i+1}, \ldots, \mathbb{F}_{i+k}$ using $O(\ell)$ time. Using this order, the algorithm 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 mainly repeats the correspondent 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 case is clear.

Suppose that an SLP $\mathbb{S}$ that derives 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 next factors that occur in $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Let us consider the subgrammars $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \ldots, \mathbb{F}_{i+\ell}$ of $\mathbb{S}$ that derive texts $F_{i+1}, F_{i+2}, \ldots, F_{i+\ell}$ correspondingly. The height of $\mathbb{F}_{i+j}$ is not greater than $1.4404 \log |F_{i+j}|$ [5]. Hence by Lemma 3.2 the number of new rules that the algorithm adds at each step of constructing 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 from 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 concatenation of $\mathbb{S}$ and $\mathbb{F}$ is equal to $O(\log n)$. Hence the size of SLP $\mathbb{S} \cdot \mathbb{F}$ that derives the text $F_1 \cdot F_2 \cdot \ldots \cdot F_{i+\ell}$ is equal to $O((i + \ell) \log n)$. □

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

3.4. SLP construction using Cartesian trees As we already noticed, SLP construction algorithms that use AVL-trees spend a lot of time on balance. We think that following idea maybe useful for solving SLP construction problem: to change data structure that represents SLPs from an AVL-tree to another one that spends less time on balance. In this section we present an algorithm that construct SLPs using Cartesian trees.

There is a probabilistic logarithmic estimation of the height of a Cartesian tree that depends on the total number of nodes (see [10]). Namely if priorities of nodes were chosen randomly, independently and has the same distribution, then the expected height of a Cartesian tree with n nodes is $O(\log n)$. Also for every fixed constant c where $c > 1$ the probability of the event that height of a Cartesian tree with n nodes is greater than $2c \ln n$ is bounded by $n \left(\frac{n}{e}\right)^{-c \ln(c/e)}$.

We need the two main operations to construct an SLP from an LZ-factorization: the operation of cutting a subtree by specified positions and the operation of concatenation of two trees. For a Cartesian tree it is easy to implement the following operations: *split* is the operation of partition of a tree into two subtrees by a specified position and *merge* is the operation of merging two trees. But a standard implementation of *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 keys of a tree that was obtained after applying *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, the algorithm cuts a subtree T' of T that derives a next factor and applies *merge* operation to T and T'. Therefore the algorithm should completely regenerate keys of T' before merging T and T'. It is profitable to avoid explicit storing of keys for the sake of efficiency. Next we explain why it is possible.[1]

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

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

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

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

***Split* operation.** The input has 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 holds all nodes of T with keys less than k and R holds all other nodes of T. By definition the operation produces two empty trees on the following input: (empty tree, 1).

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

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

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

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

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

Since a parse tree of an SLP is maximal binary tree then we should modify *split* operation to guarantee that result trees are maximal. To achieve this aim it is enough to delete all nodes that have exactly single child from both output trees. Formally if a node T_j has a single child T_k then we delete T_j from a tree. If T_j is the root then we set up T_k as the root after deleting T_j. Priorities of nodes does not change.

Obviously if an input tree T is maximal then there is at most one node with a single child at each step of the algorithm in every output tree L or R. So time complexity of maximizing both trees L and R is equal to $O(\log |T|)$. Indeed practical implementation of the maximization procedure does not require independent pass through the output since it may be integrated into the algorithm. Next in the paper by *split* operation we mean its modified version that returns maximal trees.

***Merge* operation.** The input has two Cartesian tress 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 produces T'' and vise-versa if T'' is empty then the operation produces T'.

The algorithm starts from roots T'_0 and T''_0 of the trees T' and T'' respectively and works recursively. Let T'_0 is equal to (T'_ℓ, T'_r) and T''_0 is equal to (T''_k, T''_q). Since priorities of all nodes were chosen randomly and independently we suppose that they are pairwise different. There are two cases:

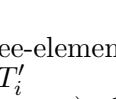
(M1) If the priority of the node T'_0 is greater than priority of the node T''_0, then the algorithm sets T'_0 as the root of T. The left subtree is equal to $\overline{T}'_\ell$ and the right subtree is equal to the tree that *merge* operation produces on the input $(\overline{T}'_r, T'')$.

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

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

As in the case of *split* operation we should modify *merge* operation to be able apply it during SLP construction. There are two problems. The first one is that we should guarantee that a result tree is a maximal binary tree. The second one is that we should guarantee that an array of leaves of T is equal to concatenation of an array of leaves of T' and an 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''. We would like to note that extreme leaves defined unambiguously in Cartesian trees

with implicit keys. Let y' and y'' be priorities of T'_i and T''_j correspondingly. Let y_* be equal to $\min(y', y'')$ and let y^* be equal to $\max(y', y'')$. We set priorities of T'_i and T''_j to be equal to y_*. The algorithm eventually reaches the following configuration applying rules (M1) and (M2): the current roots T'_0 and T''_0 are equal to leaves T'_i and T''_j correspondingly. At this moment the algorithm adds a 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 the two-element subtree ( or ). Clearly the modified algorithm spends 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. Also T is a concatenation of T' and T'' in the sense of SLPs, see Section 2. Next in the paper by *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. Next we introduce a Cartesian SLP construction algorithm.

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

The algorithm invokes *split* operation with $(\mathbb{S}, \ell^*)$. Let R be the rightmost tree from the output. Next the algorithm invokes *split* operation with $(R, r^* - \ell^*)$. The leftmost tree from the output is a Cartesian SLP $\mathbb{F}$ that derives F_{i+1}. Finally the algorithm invokes *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 expectation of the execution time of the presented algorithm on a text S of length n and its LZ-factorization of size k is equal to $O(k \log n)$. The expectation of the size of an SLP returned by the algorithm is equal to $O(k \log n)$.*

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

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

§4. Practical results

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

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

These types of strings have been chosen for the following reasons. Fibonacci strings are known to be one of the best inputs to the SLP Construction problem. So we can estimate potential of SLPs as a compression model. Random strings are considered to be incompressible and, potentially, they are the worst input for 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 Section 3 with classic compression algorithms from the Lempel-Ziv family. Our test suite contains two implementations of the Lempel-Ziv algorithm [14]: the algorithm with small (32Kb) searching window and the algorithm with infinite searching window. Also the test suite contains an implementation of the Lempel-Ziv-Welch algorithm [13]. All source code is available under the following link: `http://code.google.com/p/overclocking/`. All algorithms run in the same environment on PC with the following characteristics: Intel Core i7-2600, 3.4GHz, 8Gb operational memory, OS Windows 7 x64.

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

Figures 4–7 present main experimental results on random strings and DNA sequences. For convenience, we accept the following common conventions for tested 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 Ritter's algorithm from Subsection 3.3;
- ▲ – cartesian SLP construction algorithm from Subsection 3.4.

We estimate performance of a compression algorithm in terms of compression ratio and execution time. We calculate compression ratio as the ratio between the size of a compressed presentation and the size of an input text. We measure compression ratio in percents. For example, the formula for SLP compression ratio looks like $\frac{|\mathbb{S}|}{|S|} \cdot 100$. We additionally calculate number of rotations for SLP construction algorithms that use AVL trees.

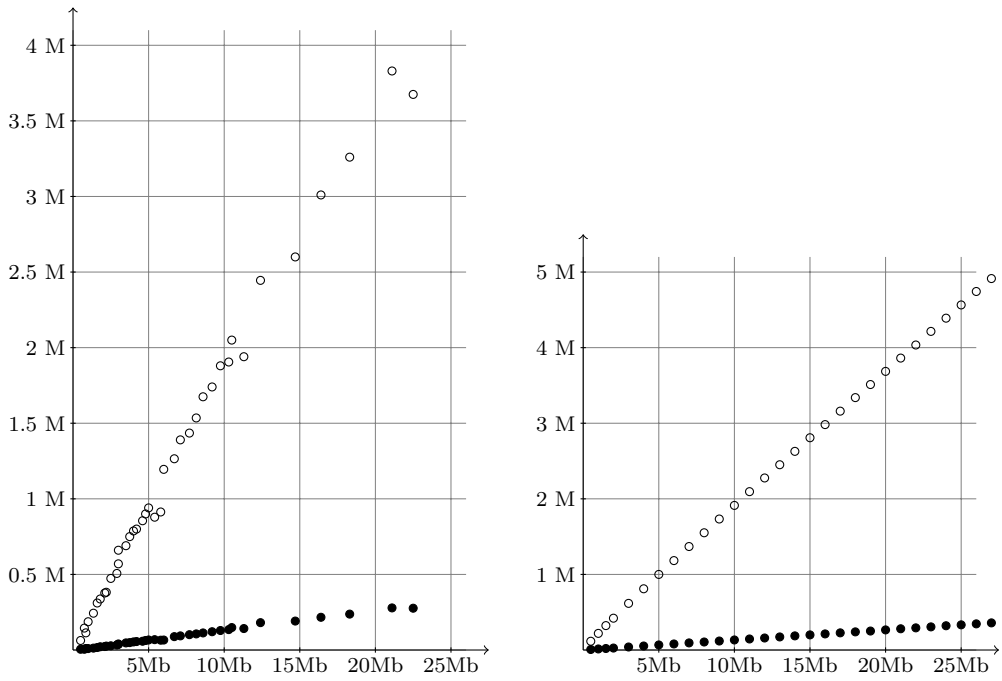


Figure 4: AVL rotations statistics on DNA sequences (from the left) and on random strings (on the right)

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

As discussed at Subsection 3.3 optimization of the number of rotations does not guarantee optimization of speed of SLP construction since the modified algorithm spends additional time on calculation of the optimal order of concatenation. We compare speed of all SLP construction algorithms using the two following tests. In the first one, the algorithms have stored all SLPs being constructed in operational memory, while in the second one, the SLPs have been stored in an external file so that every rotation of an AVL-tree forces I/O operations with the file. Figures 5 and 6 present results of both tests on DNA sequences and random strings correspondingly. From experimental results it follows that the modified algorithm from Subsection 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 an SLP is stored in operational memory. Also it works five time faster on DNA sequences and three times faster on random strings if an SLP is 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 for this is that heights

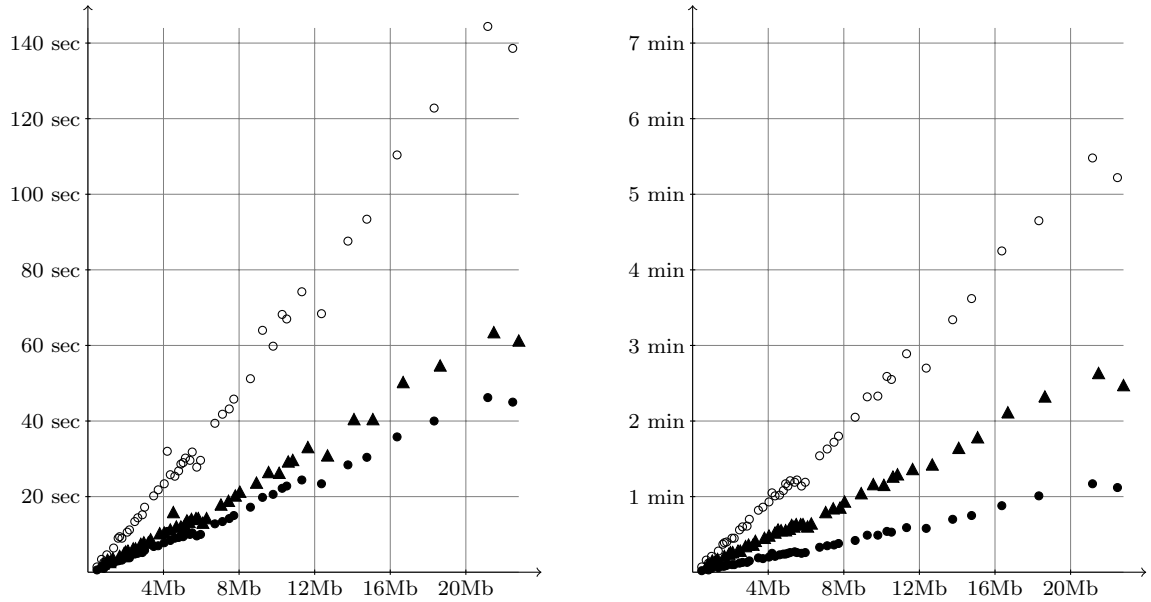


Figure 5: SLP construction time on DNA sequences when an SLP stores in operational memory (from the left) and in external file (on the right)

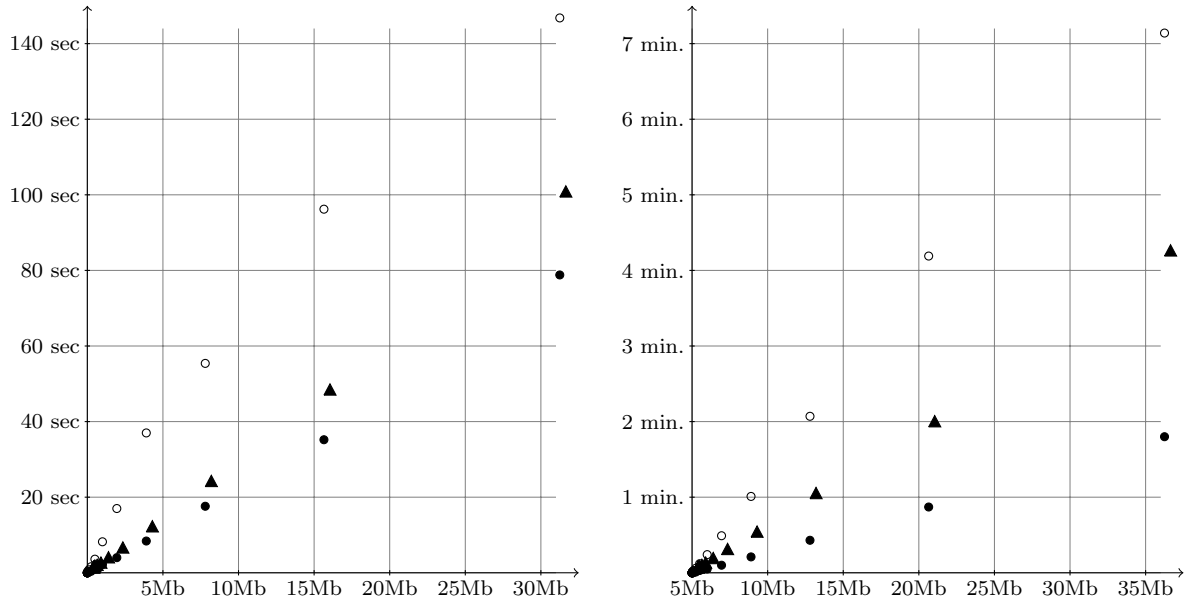


Figure 6: SLP construction time on random strings when n SLP stores in operational memory (from the left) and in external file (on the right)

of constructed Cartesian trees are essentially larger than heights of correspondent 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. So Cartesian SLP construction algorithm processes more rules than algorithms that use AVL-trees during concatenation. This compensates the gain of simplicity of balance supporting in Cartesian trees.

Figure 7 presents experimental results of compression ratio achieved by SLP construction algorithms and by the classic compression algorithms from the Lempel-Ziv family. From the figure it follows that the algorithms that use AVL-trees achieve similar compression ratio that is twice less on average than compression ratio achieved by LZW. It is interesting that ratio between compression ratios achieved by algorithms that use AVL-trees and compression ratio achieved by LZW does not depend on time, type and length of input text. Compression ratio of the algorithm that uses Cartesian trees is essentially worse than compression ratio of other algorithms. In this case we also observe that ratio between compression ratios weakly depends on type and length of input text.

§5. Conclusion

Our experimental results shows that both Rytter's algorithm and the modified algorithm achieve the same compression ratio. But time of SLP construction of the second algorithm is essentially smaller than analogous time of the first one. Since using of a file system is inevitable with growing of the input, it is important to notice that the modified algorithm is more stable to growing of input than Rytter's algorithm.

In the paper we present Cartesian SLP construction algorithm. This algorithm is close to another discussed SLP construction algorithms by execution time but essentially yield them by achieved compression ratio and by height of an output tree. This circumstances is important for searching algorithms that works directly with compressed representations. So our aim to improve performance of SLP construction using efficient data structure was not achieved. Now we think that this aim is hard to reach. It appears that searching of new heuristics on AVL-trees that allow one to construct more compact SLPs is a more productive idea.

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

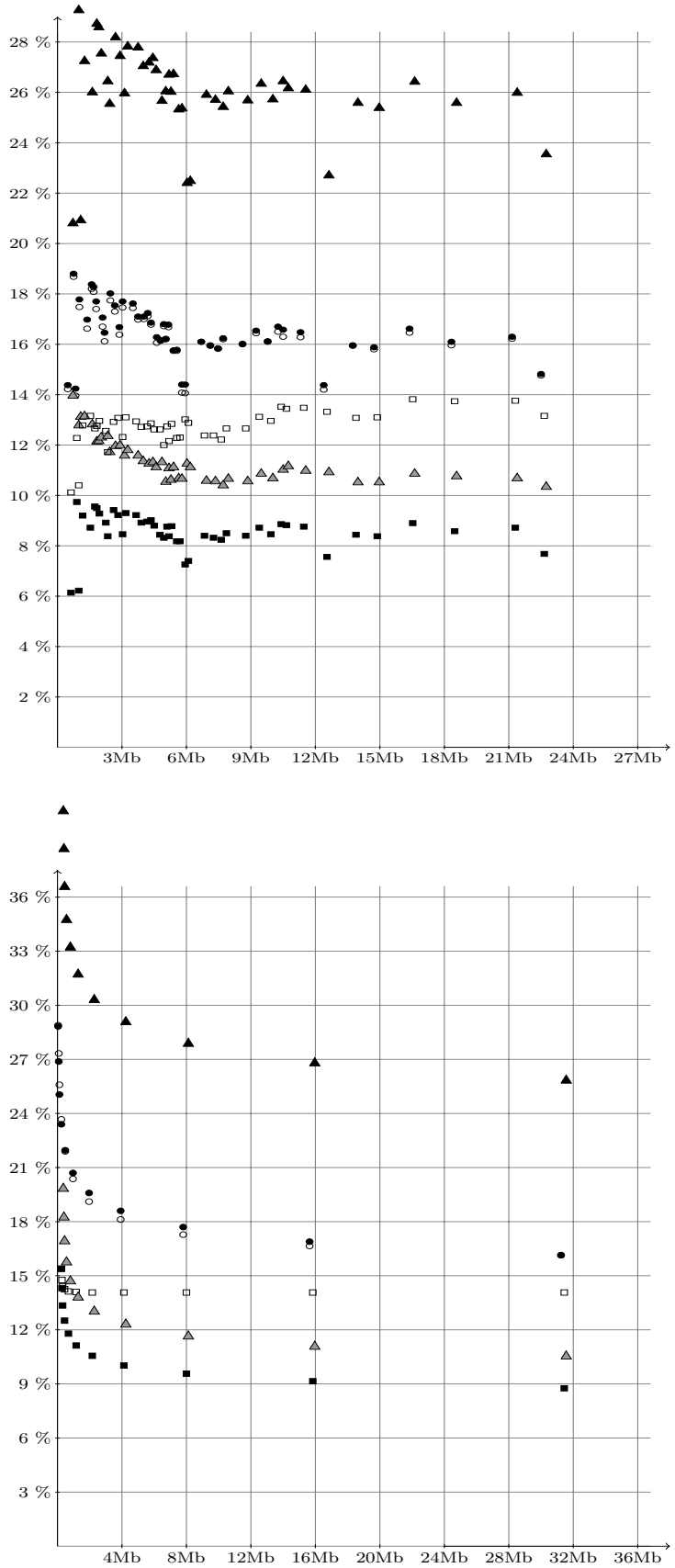


Figure 7: Compression ratio achieved on DNA sequences (at the top) and on random strings (at the bottom)

Acknowledgment

The authors would like to thank professor Mikhail V. Volkov for his critical notes and absolute supporting of their activity. The authors would like to thank anonymous referee for his notes and improvements of the initial text of the paper.

References

[1] *A. Apostolico, G. M. Landau, S. Skiena*, Matching for Run-Length Encoded Strings, J. Complexity, 15 (1999), 4–16.

[2] *I. Burmistrov, L. Khvorost*, Straight-line programs: a practical test, Proc. Int. Conf. Data Compression, Commun., Process., CCP (2011), 76–81.

[3] *M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat*, The smallest grammar problem, IEEE Trans. Information Theory, 51 (2005), 2554–2576.

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

[5] *D. Knuth*, The Art of Computer Programming, vol. 3. Sorting and Searching, Second Edition: Addison-Wesley, 1998.

[6] *Y. Lifshits*, Processing compressed texts: A tractability border, Lect. Notes Comput. Sci., 4580 (2007), 228–240.

[7] *W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, K. Hashimoto*, Computing longest common substring and all palindromes from compressed strings, Lect. Notes Comput. Sci., 4910 (2008), 364–375.

[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, Theor. Comput. Sci., 302 (2003), 211–222.

[10] *R. Seidel, C. Aragon*, Randomized search trees, Algorithmica 16 (1996), 464–497.

[11] *Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa*, Pattern matching in text compressed by using antidictionaries, Lect. Notes Comput. Sci., 1645 (1999), 37–49.

[12] *A. Tiskin*, Faster subsequence recognition in compressed strings, Journal of Mathematical Sciences, 158 (2009), 759—769.

[13] *T. Welch*, A technique for high-performance data compression, IEEE Computer, 17 (1984), 8–19.

[14] *J. Ziv, A. Lempel*, A universal algorithm for sequential data compression, IEEE Trans. Information Theory, 23 (1977), 337–343.

[15] *J. Ziv, A. Lempel*, Compression of individual sequences via variable-rate coding, IEEE Trans. Information Theory, 24 (1978), 530–536.