Straight-line Programs: A Practical Test[*]

Ivan Burmistrov Lesha Khvorost

Abstract

We present an improvement of Rytter's algorithm that constructs a straight-line program for a given text and show that the improved algorithm is optimal in the worst case with respect to the number of AVL-tree rotations. Also we compare Rytter's and ours algorithms on various data sets and provide a comparative analysis of compression ratio achieved by these algorithms, by LZ77 and by LZW.

Keywords: straight-line program, LZ-compression, AVL-tree.

1 Introduction

Nowadays searching algorithms on huge data sets attract much attention. To reduce the input size one needs algorithms that can work directly with a compressed representation of input data.

Various compressed representations of strings are known: straight-line programs (SLPs) [5], collage-systems [2], string representations using antidictionaries [6], etc. Nowadays text compression based on context-free grammars such as SLPs has become a popular research direction. The reason for this is not only that grammars provide well-structured compression but also that the SLP-based compression is, in a sense, polynomially equivalent to the compression achieved by the Lempel-Ziv algorithm that is widely used in practice. It means that, given a text S, there is a polynomial relation between the size of an SLP that derives S and the size of the dictionary stored by the Lempel-Ziv algorithm, see [5]. It should also be noted that classical LZ78 [10] and LZW [8] 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 [9] and run-length compression—do not fit directly into grammar compression model.)

Using the fact that SLPs are nicely structured, several researchers keep developing analogues of classical string algorithms that (at least theoretically) perform quite well on SLP-compressed representations: **Pattern matching** [3], **Longest common substring** [4], **Computing all palindromes** [4], some versions of **Longest common subsequence** [7]. At the same time, constants hidden in big-O notation for algorithms on SLPs are often very big. Also the

[*] The authors acknowledge support from the Ministry for Education and Science of Russia, grant 2.1.1/13995, and from the Russian Foundation for Basic Research, grant 10-01-00524.

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 questions 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 SLPs. In Section 3 we present an improved version of Rytter's algorithm [5] for constructing an SLP-presentation of a given text. In Section 4 we compare the improved algorithm vs. the original algorithm from [5] and also present results of a comparison of compression ratio between the two SLP-based algorithms, LZ77 and LZW. In Section 5 we summarize our results.

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

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 an SLP. For a grammar $\mathbb{S}$ generating a text S, we define *parse-tree* of S as a 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 sons. We accept the following conventions in the paper: every SLP is denoted by a capital blackboard bold letter, for example, $\mathbb{S}$. Every rule of this SLP (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 *concatenation* of SLPs $\mathbb{S}_1$ and $\mathbb{S}_2$ is an SLP that derives $S_1 \cdot S_2$ and denoted by $\mathbb{S}_1 \cdot \mathbb{S}_2$.

The *height* of a node in a binary tree is defined as follows. The height of 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. An *AVL-tree* $\mathbb{T}$ is a binary tree such as for every non-terminal node the heights of its children can differ at most by 1. In the paper we consider only SLPs whose parse-trees are AVL-trees. We denote the height of the rule $\mathbb{S}_i$ by $h(\mathbb{S}_i)$.

3 Algorithms for constructing SLPs

3.1 Rytter's algorithm and its bottleneck

The problem of constructing a minimal size grammar generating a given text is known to be NP-hard [1]. Hence we should look for polynomial-time approximation algorithms. One of the key approaches of such algorithms is to construct a factorization of a given text and to build some binary search tree using the factorization. It is obvious that the size of the resulting grammar depends on factorization. In [5] Rytter considered factorizations defined by the LZ77 encoding algorithm.

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 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 following theorem gives an estimation of the size of an SLP constructed on the basis on a LZ-factorization.

Theorem 3.1 ([5])**.** *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 the above theorem contains an algorithm for constructing SLPs. We remind here some key ideas of the algorithm as they are important for the further discussion. For efficient concatenation of grammars the algorithm uses the following lemma:

Lemma 3.2 ([5])**.** *Assume $\mathbb{S}_1, \mathbb{S}_2$ are two nonterminals of AVL-balanced grammars. Then we can construct in $O(|h(\mathbb{S}_1) - h(\mathbb{S}_2)|)$ time an AVL-balanced 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**
INPUT: a string 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 $S[0]$.

Step: Let $i > 1$ be an integer and an SLP $\mathbb{S}$ that derives $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ has already been constructed. Since a 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. From this, the algorithm extracts rules $\mathbb{S}_1, \ldots, \mathbb{S}_\ell$ such that $F_{i+1} = S_1 \cdot S_2 \cdot \ldots \cdot S_\ell$. Since $\mathbb{S}$ is balanced, we have $\ell = O(\log |S|)$. Then, using Lemma 3.2, the algorithm concatenates rules in some specific order (see [5] 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 $\mathbb{S}_1 \cdot \ldots \cdot \mathbb{S}_\ell$.

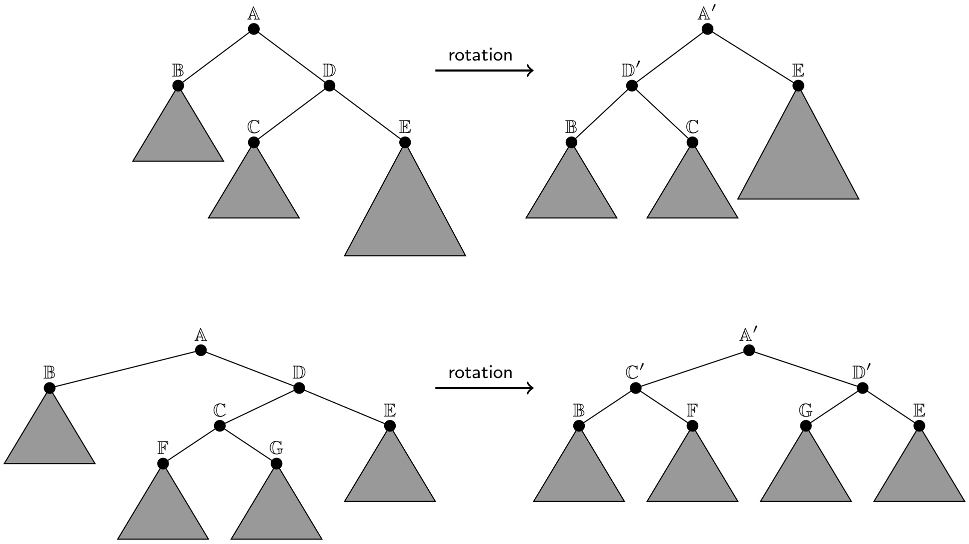

Figure 1. Rotations of an AVL tree

It is well-known that adding a new rule to an AVL-tree is quite a complex operation. Every adding operation generates a sequence of rotations of the tree. There are two types of rotations symbolically presented in Figure 1, see [5] for more details. Every rotation may generate at most three new rules. Also every rotation may generate three unused rules. There are two possible directions for an optimization of the algorithm: to construct more compact grammar and to minimize the number of queries to AVL-trees. Minimizing of the number of queries to AVL-trees 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.

The following example illustrates a bottleneck of the algorithm from [5]:

Example 1: Let n be an integer and S be a text of length 2^n. Suppose the algorithm has already built an SLP $\mathbb{S}$ that derives $S[0 \ldots 2^{n-1}]$. Let $F_1 \cdot F_2 \cdot \ldots \cdot F_{n-1}$ be LZ-factorization of $S[2^{n-1} \ldots 2^n]$ such that $|F_i| = 2^{n-i-1}$, where $i \in \overline{1 \ldots n-1}$.

Let us estimate the number of rotations that may be generated in the course of the sequence of concatenations $(((\mathbb{S} \cdot \mathbb{F}_1) \cdot \mathbb{F}_2) \ldots) \cdot \mathbb{F}_{n-1}$ in the worst case.

In order to get an upper bound, we suppose that the resulting tree grows after every concatenation. So we get the following estimation:

$$\sum_{i=1}^{n-1}(\log 2^{n-1}+(i-1)-\log 2^{n-i-1}) = \frac{(n-1)(n-2)}{2}(1+\log 2)-(n-2) = \Theta(n^2).$$

Notice that if the algorithm could choose another order of concatenations, then the number of rotations generated by the algorithm could be essentially smaller, namely, close to $O(n)$. For instance, in the case of concatenating in the reverse order $\mathbb{S}\cdot(\mathbb{F}_1 \dots (\mathbb{F}_{n-2}\cdot\mathbb{F}_{n-1}))$, we get the following estimation:

$$\sum_{i=1}^{n-1}(\log 2^{n-i} - \log 2^{n-i-1}) = (n-1)\log 2 = \Theta(n).$$

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 $S = b\cdot a^{2^{n-1}}\cdot b\cdot a^{2^{n-2}}\cdot\ldots\cdot b\cdot a$. So the length of S is equal to $2^n + n - 2$. Consider the LZ-factorization of S:

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

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 the algorithm may process them together. So the algorithm may 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 the algorithm may concatenate the SLPs in the following order: $\mathbb{S}_1\cdot(\ldots(\mathbb{S}_{n-3}\cdot(\mathbb{S}_{n-2}\cdot\mathbb{S}_{n-1})))$.

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.

3.2 An improved algorithm

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 $S[0]$.

Step: Let i be an integer and $\mathbb{S}$ be an SLP that derives $F_1\cdot F_2\cdot\ldots\cdot F_i$. Let $\ell\geq 1$ be a maximal integer such that $F_{i+\ell}$ occurs in $F_1\cdot\ldots\cdot F_i$. Since the LZ-factorization of S is fixed, the value of ℓ can be found by using a linear time search on factors. Let $\mathbb{F}_1$ be an SLP that derives F_{i+1}, let $\mathbb{F}_2$ be an SLP that derives F_{i+2}, etc.

Initially the algorithm concatenates $\mathbb{F}_{i+1},\ldots,\mathbb{F}_{i+k}$ using dynamic programming. Let $\varphi(p,q)$ be the function that calculates an 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 algorithm uses the following recurrent formula to calculate $\varphi(p, q)$:

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

The formula is correct since the concatenation of $\mathbb{F}_p$ and $\mathbb{F}_q$ generates at most $2 \cdot |h(\mathbb{F}_p) - h(\mathbb{F}_q)|$ rotations. In the formula we omit the constant factor 2 since, otherwise, all values of φ will be proportional to the factor.

The algorithm calculates the values of $\varphi(p, q)$ using a φ-table. The φ-table is a $(k-1) \times (k-1)$-table that stores information about the value of $\varphi(p, q)$ and about the optimal value of index r. The cell in the p-th row and the q-th column where $p \leq q$ always contains zero values. Initially 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 φ-table.

Every single value of $\varphi(p, q)$ can be calculated using $O(k)$ time (the pseudocode of the corresponding procedure is presented in Figure 2). Thus, the algorithm fills out the φ-table using $O(k^3)$ time and $O(k^2)$ space.

```
$result = +\infty;$
$l = 0, r = |f_{i+p}| + \cdots + |f_{i+q}|;$
for $(int\ r = p; r \leq q; r++;)$ {
    $l+ = |f_{i+r}|;$
    $r- = |f_{i+r}|;$
    $tmp = \varphi(p, r) + \varphi(r + 1, q) + |\log l - \log r|;$
    if $(tmp < result)$
        $result = tmp;$
}
```

Figure 2. Pseudo code of computing single value of $\varphi(p, q)$

Notice that $\varphi(1, k)$ is equal to the optimal number of rotations of the grammar tree in the worst case (i.e. in the case when each concatenation of $\mathbb{F}_p$ and $\mathbb{F}_q$ generates $O(|h(\mathbb{F}_p) - h(\mathbb{F}_q)|)$ rotations). Since the φ-table contains information about the optimal value of index r in every cell, the algorithm can determine an optimal order of concatenations for $\mathbb{F}_{i+1}, \ldots, \mathbb{F}_{i+k}$. Using the optimal order, the algorithm constructs an SLP $\mathbb{F}$ that derives $F_{i+1}, F_{i+2}, \ldots, F_{i+l}$. 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 $f_1, f_2, \ldots, f_k$ be the LZ-factorization of a text w. The above algorithm constructs an SLP for w of size $O(k \log n)$.*

Proof: The LZ-factorization can be divided into independent groups of factors. Let ℓ be the number of the groups and let $k_1, k_2, \ldots, k_\ell$ be the sizes of the corresponding groups. The number of new rules generated during concatenation of all factors in the i-th group is upper bounded by $2 \cdot (\log(|f_{k_i}| + |f_{k_i+1}| + \ldots + |f_{k_{i+1}-1}|) \cdot (k_{i+1} - k_i))$. So the number of new rules generated during a single step of the algorithm is upper bounded by $2 \cdot (\log(|f_{k_1}| + |f_{k_1+1}| + \ldots + |f_{k_i-1}|) + (k_{i+1} - k_i - 1) \cdot (\log(|f_{k_i}| + \ldots + |f_{k_{i+1}-1}|))$. Finally the number of rotations is upper bounded by $2 \cdot (k_{i+1} - k_i) \cdot \log n$.

Totally the algorithm generates at most $O(k \log n)$ new rules.

Since concatenation of groups of factors is optimal in the worst case, it is difficult to determine the precise time complexity of the improved algorithm as a function of input size. Notice that if the size of every group of factors is equal to 1 then we have the algorithm from [5]. Otherwise the new algorithm generates less rotations than the standard algorithm but the new algorithm spends some extra time for calculating order of concatenation. In the next section we propose a practical comparison between the two algorithms.

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 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 one of the best inputs to the SLP Construction problem. So we can estimate potential of SLPs as compression model. Random strings are considered to be incompressible and, potentially, they are the worst input for the SLP Construction problem. DNA sequences are real data used in practice.

Let us introduce algorithms that have been tested. Our test suite contains a classic version of the Lempel-Ziv algorithm proposed in [9]. We use the following brief notation for it: **lz77**. The size of scanning window is equal to 32Kb. The reader may think that 32Kb is too short but we choose this value to study how the window size influences compression ratio.

Notice that the size of scanning window affects the length of factors and the total size of a LZ-factorization. So if we want to compress longer factors, then we need to spend more space. In other words, the size of scanning window directly affects the size of compressed presentation. In the same time there is no direct relation between the structure of an SLP and the size of the text derived from it. Therefore using shorter LZ-factorizations leads to constructing more compact SLPs. From this argument it follows that there is a reason to consider a version of the Lempel-Ziv algorithm with an infinite scanning window. We use the following brief notation for it: **lz77inf**.

Additionally we test yet another classic algorithm from the Lempel-Ziv family: the Lempel-Ziv-Welch algorithm (briefly **lzw**). Finally our test suite contains the classic algorithm for SLP Construction problem (briefly **SLPclassic**) proposed in [5] and the algorithm presented in Section 3 (briefly **SLPnew**).

For convenience, we accept the following common notation for the tested algorithms in all following graphs:

- □ – **lz77** (Lempel-Ziv algorithm with 32Kb scanning window)
- ■ – **lz77inf** (Lempel-Ziv algorithm with infinite scanning window)
- ▲ – **lzw** (Lempel-Ziv-Welch algorithm)
- ○ – **SLPclassic** (the algorithm proposed at [5])
- ● – **SLPnew** (the algorithm from Section 3)

We estimate the performance of a compression algorithm in terms of compression ratio and execution time. For SLP compression algorithms we additionally calculate the number of rotations. We calculate compression ratio as the ratio between the size of compressed presentation and the size of input string. We measure compression ratio in percents. For example, the formula for SLP compression ratio looks like $\frac{|\mathbb{S}|}{|S|} \cdot 100$. Since ratio between serialized SLP rule representation and serialized LZ-factor representation is greater than 1 (but constant) we may encounter the following situation: SLP compression ratio is less than LZ compression ratio but the file containing an SLP is larger than the file containing the corresponding LZ-dictionary. We should note that the situation changes with input growing.

All algorithms have been implemented using Java SE and tested on Intel Core i-5(661), 4Gb RAM (Microsoft Windows 7 x64 OS).

4.2 SLP construction results

In this subsection we discuss performance of both SLP construction algorithms.

As expected, both algorithms work infinitely fast on Fibonacci strings and construct extremely compact representations. For example, on the 36-th Fibonacci word of size 36.9Mb both algorithms work within 1ms and build SLPs of size 100. This fact shows that there is a class of strings whereon the SLP compression model is very efficient.

A diagram at Figure 3 shows how the suggested optimization of concatenation affects the number of rotations of AVL trees on DNA sequences.

To illustrate how the optimization affects construction time we present two 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. We expect that there is no big difference between the algorithms in the first setting since the new algorithm spends extra time when computing the most efficient order of concatenation. But what happens when costs of accessing AVL-trees are greater than costs of computing the order? This situation naturally appears when the size of input is huge and we cannot store all the trees in operational memory. Figures 4 and 5 present the execution time statistic for both ways of SLPs storing.

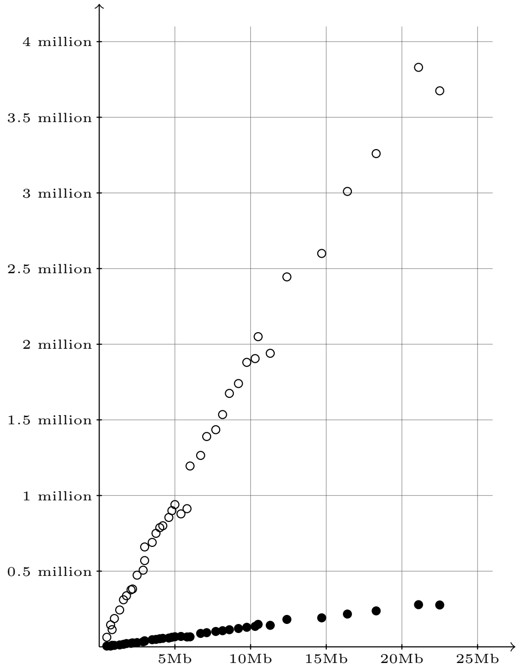

Figure 3. Number of rotations of an AVL tree on DNA sequences

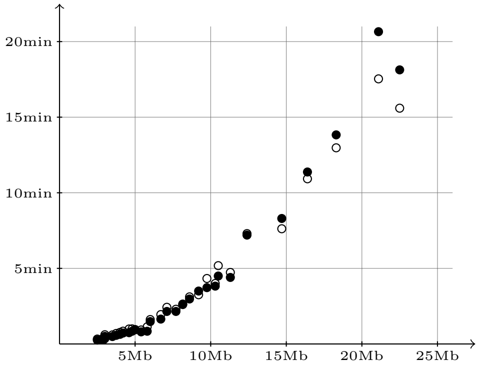

Figure 4. Time statistic of an SLP construction on DNA sequences
(an AVL tree stored in operational memory)

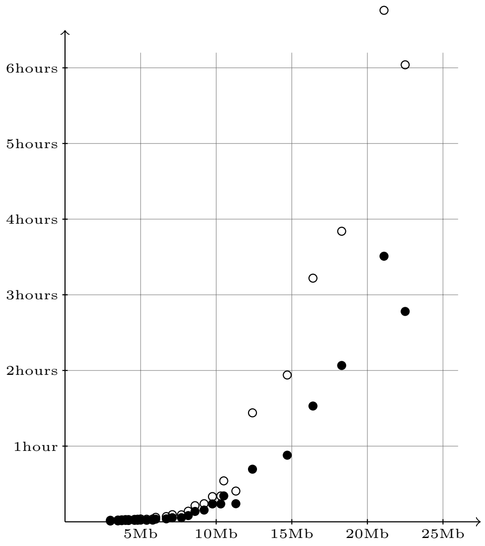

Figure 5. Time statistic of an SLP construction on DNA sequences (an AVL tree stored in file)

Figures 6, 7 and 8 present the results of testing both algorithms on random strings.

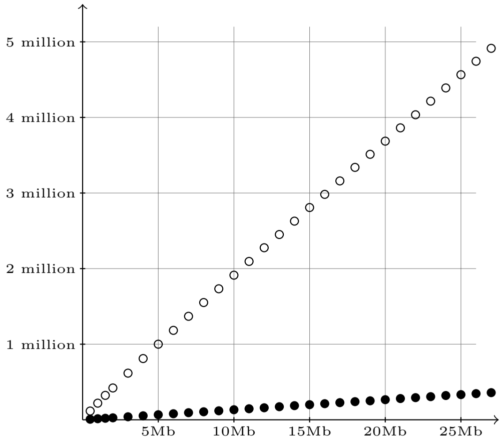

Figure 6. Number of rotations of an AVL tree on random strings

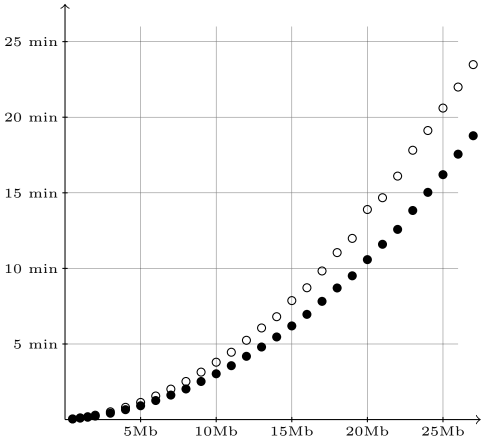

Figure 7. Time statistic of an SLP construction on random strings (an AVL tree stored in operational memory)

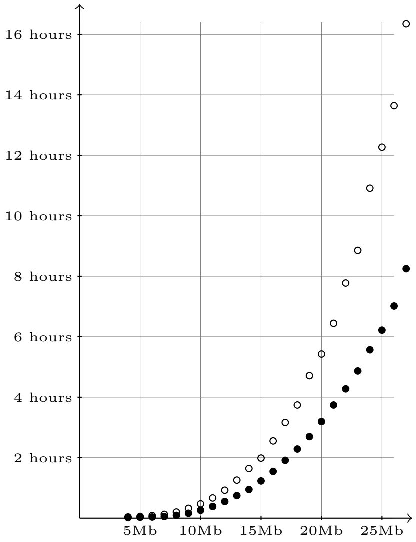

Figure 8. Time statistic of an SLP construction on random strings (an AVL tree stored in file)

The Figures 3-8 shows that the new algorithm is more stable since it is on average as fast as the classic algorithm in operational memory and on average two times faster when a file system is used.

4.3 Compression ratio results

Here we present a comparative analysis of compression ratios for DNA sequences and random strings.

Figures 9 and 10 shows that compression ratio provided by the new algorithm is close to compression ratio provided by the classic algorithm. In case of DNA sequences the new algorithm generates many rotations of small AVL trees and as a result it constructs a wider SLP than the classic algorithm. Also tests shows that SLP compression ratio directly depends on the size of LZ-factorization (fluctuations in the results obtained by both classic and new algorithms are similar to fluctuations in the results obtained by **lz77inf** and the size of compressed presentation obtained by **lz77inf** is equal to the LZ-factorization size).

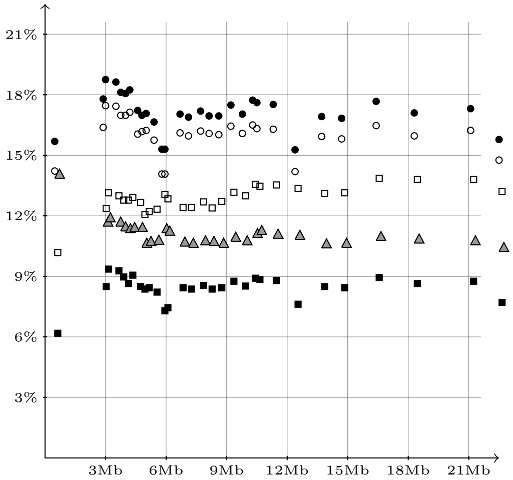

Figure 9. Compression results on DNA sequences

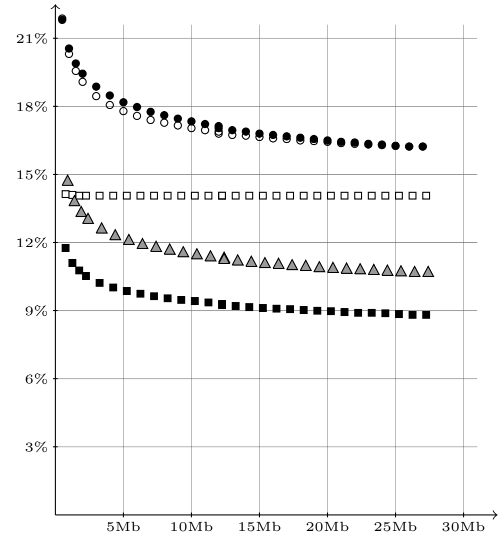

Figure 10. Compression results on random strings

5 Conclusion

In this paper we present an improved algorithm for SLP construction. The algorithm is similar to Rytter's algorithm proposed in [5] but it seems to be more efficient on large inputs. However, there is an open issue related to the new algorithm: how to manage groups of factors optimally? For example, suppose that we already have built an SLP $\mathbb{L}$ of height h and suppose that the next group of factors generates an SLP $\mathbb{R}$ of height $2h$. It looks inefficient to construct the whole $\mathbb{R}$ and then concatenate it with $\mathbb{L}$. A possible solution is to let the allowed size of the group gradually grow as the algorithm progresses on the input string.

In the paper we also present practical tests of SLPs as compression model. As a result we should conclude that the existing ways of SLP construction are quite slow. On the other hand, the tests confirm that compression ratio provided by SLPs is close to compression ratio provided by the family of Lempel-Ziv algorithms (and may be even closer than we initially expected).

Acknowledgment

The authors would like to thank *Mikhail V. Volkov* for his encouragement and critical notes and *Anna Kozlova* for her help in gathering statistics.

References

[1] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

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

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

[4] W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Computing longest common substring and all palindromes from compressed strings. In *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 2008.

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

[6] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *CPM*, volume 1645 of *Lecture Notes in Computer Science*, pages 37–49. Springer, 1999.

[7] A. Tiskin. Faster subsequence recognition in compressed strings. *CoRR*, abs/0707.3407, 2007.

[8] T. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[9] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[10] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.