Straight-line Programs: A Practical Test

Ivan Burmistrov [†] and Lesha Khvorost [†]

[†]Department of Mathematics and Mechanics
Ural State University

22th June CCP 2011

Observation

With growing input size of classic string algorithms (**Pattern Matching**, **Longest Common Substring**, etc.) changes the algorithms that efficiently process the input.

There exist the following approaches:

- Processing the input using a file system (**I/O efficient algorithms**);
- Store and process the input as compressed representation (**algorithms on compressed representations**);

Observation

With growing input size of classic string algorithms (**Pattern Matching**, **Longest Common Substring**, etc.) changes the algorithms that efficiently process the input.

There exist the following approaches:

- Processing the input using a file system (**I/O efficient algorithms**);
- Store and process the input as compressed representation (**algorithms on compressed representations**);

Observation

With growing input size of classic string algorithms (**Pattern Matching**, **Longest Common Substring**, etc.) changes the algorithms that efficiently process the input.

There exist the following approaches:

- Processing the input using a file system (**I/O efficient algorithms**);
- Store and process the input as compressed representation (**algorithms on compressed representations**);

Straight-line Program Definition

Definition

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

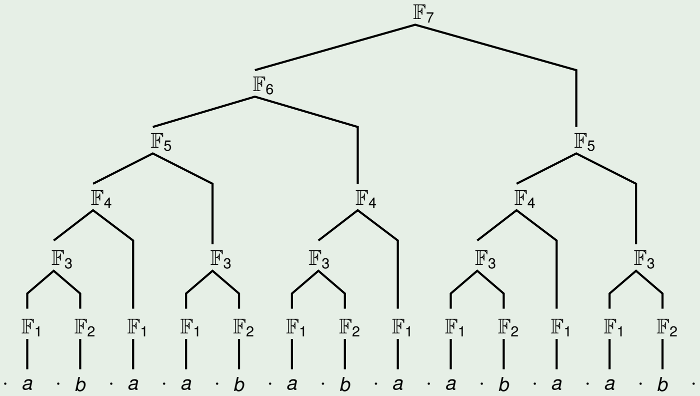
$$\mathbb{S}_1 = expr_1,\ \mathbb{S}_2 = expr_2, \ldots, \mathbb{S}_n = expr_n,$$

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

- $expr_i \in \Sigma$ (**terminal** rules), or
- $expr_i = \mathbb{S}_\ell \cdot \mathbb{S}_r$ $(\ell, r < i)$ (**nonterminal** rules).

The Fibonacci Word SLP

Example (SLP for «*abaababaabaab*»)



Features of SLPs

Positive features

- Well-structured data representation;
- Polynomial relation between the size of SLP for a given text and the size of LZ77-dictionary for the same text;
- There are classic string problems that have polynomial time algorithms depends on size of SLPs (**Pattern Matching**, **Longest Common Substring**, **Computing All Squares**).

Negative features

- Constants hidden in big-O notation for algorithms on SLPs are often very big;
- Polynomial relation between the size of an SLP for a given text and the size of the LZ77-dictionary for the same text doesn't yet guarantee that SLPs provide good compression ratio in practice;

Features of SLPs

Positive features

- Well-structured data representation;
- Polynomial relation between the size of SLP for a given text and the size of LZ77-dictionary for the same text;
- There are classic string problems that have polynomial time algorithms depends on size of SLPs (**Pattern Matching**, **Longest Common Substring**, **Computing All Squares**).

Negative features

- Constants hidden in big-O notation for algorithms on SLPs are often very big;
- Polynomial relation between the size of an SLP for a given text and the size of the LZ77-dictionary for the same text doesn't yet guarantee that SLPs provide good compression ratio in practice;

Features of SLPs

Positive features

- Well-structured data representation;
- Polynomial relation between the size of SLP for a given text and the size of LZ77-dictionary for the same text;
- There are classic string problems that have polynomial time algorithms depends on size of SLPs (**Pattern Matching**, **Longest Common Substring**, **Computing All Squares**).

Negative features

- Constants hidden in big-O notation for algorithms on SLPs are often very big;
- Polynomial relation between the size of an SLP for a given text and the size of the LZ77-dictionary for the same text doesn't yet guarantee that SLPs provide good compression ratio in practice;

Features of SLPs

Positive features

- Well-structured data representation;
- Polynomial relation between the size of SLP for a given text and the size of LZ77-dictionary for the same text;
- There are classic string problems that have polynomial time algorithms depends on size of SLPs (**Pattern Matching**, **Longest Common Substring**, **Computing All Squares**).

Negative features

- Constants hidden in big-O notation for algorithms on SLPs are often very big;
- Polynomial relation between the size of an SLP for a given text and the size of the LZ77-dictionary for the same text doesn't yet guarantee that SLPs provide good compression ratio in practice;

Features of SLPs

Positive features

- Well-structured data representation;
- Polynomial relation between the size of SLP for a given text and the size of LZ77-dictionary for the same text;
- There are classic string problems that have polynomial time algorithms depends on size of SLPs (**Pattern Matching**, **Longest Common Substring**, **Computing All Squares**).

Negative features

- Constants hidden in big-O notation for algorithms on SLPs are often very big;
- Polynomial relation between the size of an SLP for a given text and the size of the LZ77-dictionary for the same text doesn't yet guarantee that SLPs provide good compression ratio in practice;

Main Question

Whether or not there exist SLP-based compression models suitable to practical usage?

- 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?

Main Question

Whether or not there exist SLP-based compression models suitable to practical usage?

- 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?

Main Question

Whether or not there exist SLP-based compression models suitable to practical usage?

- 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?

SLP Construction Problem

SLP Construction Problem

INPUT: A text S;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Theorem

The problem of constructing a minimal size grammar generating a given text is NP-hard.

SLP Construction Problem

SLP Construction Problem

INPUT: A text S;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Theorem

The problem of constructing a minimal size grammar generating a given text is NP-hard.

Idea of Factorization

Definition

A **factorization** of a text S is a set of strings $F_1, F_2, \ldots F_k$ such that $S = F_1 \cdot F_2 \cdot \ldots \cdot F_k$.

Definition

The **LZ-factorization** of a text S is set of strings: $F_1 \cdot F_2 \cdot \ldots \cdot F_k$, where

- $F_1 = S[0]$;
- F_i is the longest prefix of $S[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |S|]$ which occurs earlier in the text **or** $S[|F_1 \cdot \ldots \cdot F_{i-1}|]$ in case this prefix is empty.

Example (Factorizations of «*abaababaabaab*»)

- $a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot a \cdot b$;
- $a \cdot b \cdot a \cdot aba \cdot baaba \cdot ab$;

Idea of Factorization

Definition

A **factorization** of a text S is a set of strings $F_1, F_2, \ldots F_k$ such that $S = F_1 \cdot F_2 \cdot \ldots \cdot F_k$.

Definition

The **LZ-factorization** of a text S is set of strings: $F_1 \cdot F_2 \cdot \ldots \cdot F_k$, where

- $F_1 = S[0]$;
- F_i is the longest prefix of $S[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |S|]$ which occurs earlier in the text **or** $S[|F_1 \cdot \ldots \cdot F_{i-1}|]$ in case this prefix is empty.

Example (Factorizations of «*abaababaabaab*»)

- $a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot a \cdot b$;
- $a \cdot b \cdot a \cdot aba \cdot baaba \cdot ab$;

Idea of Factorization

Definition

A **factorization** of a text S is a set of strings $F_1, F_2, \ldots F_k$ such that $S = F_1 \cdot F_2 \cdot \ldots \cdot F_k$.

Definition

The **LZ-factorization** of a text S is set of strings: $F_1 \cdot F_2 \cdot \ldots \cdot F_k$, where

- $F_1 = S[0]$;
- F_i is the longest prefix of $S[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |S|]$ which occurs earlier in the text **or** $S[|F_1 \cdot \ldots \cdot F_{i-1}|]$ in case this prefix is empty.

Example (Factorizations of «*abaababaabaab*»)

- $a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot a \cdot b$;
- $a \cdot b \cdot a \cdot aba \cdot baaba \cdot ab$;

Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;
OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example

Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example

$\mathbb{F}_1$

$\cdot\ a\ \cdot\ b\ \cdot\ a\ \cdot\ a\ \ b\ \ a\ \cdot\ b\ \ a\ \ a\ \ b\ \ a\ \cdot\ a\ \ b\ \cdot$

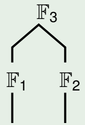
Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example



$\cdot\ a\ \cdot\ b\ \cdot\ a\ \cdot\ a\ b\ a\ \cdot\ b\ a\ a\ b\ a\ \cdot\ a\ b\ \cdot$

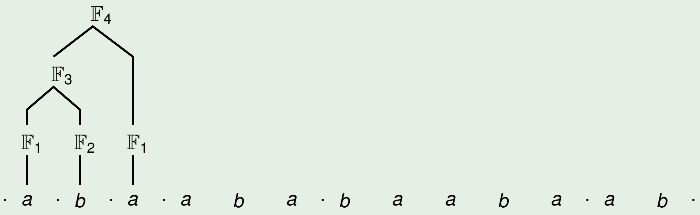
Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example



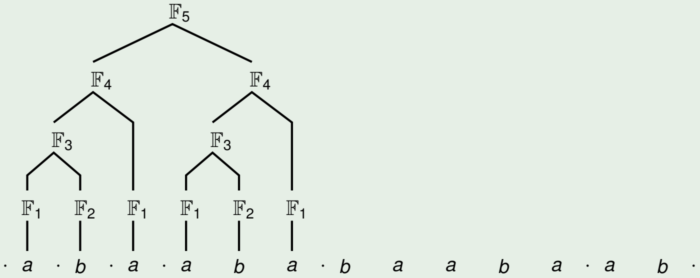
Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example



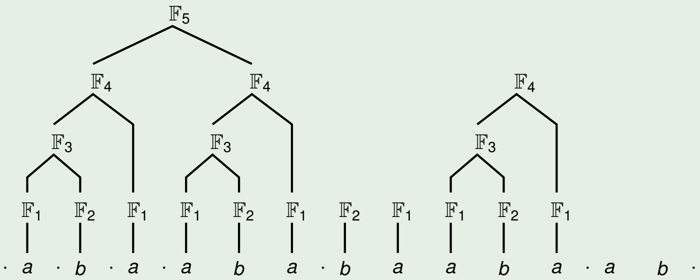
Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example



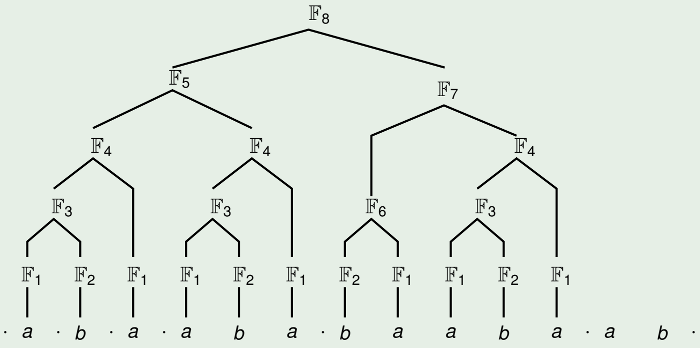
Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example

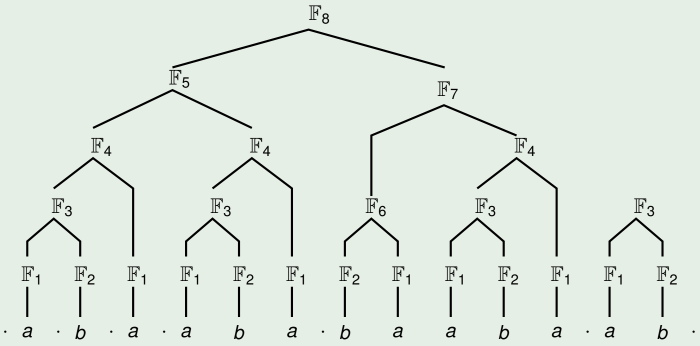


Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;
OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example



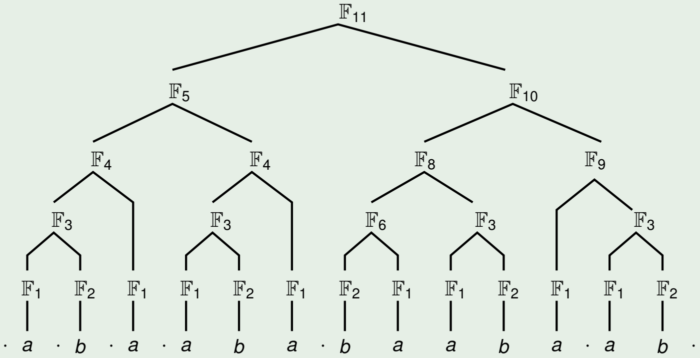
Rytter's Approach

SLP Construction Problem

INPUT: A text S and its LZ-factorization $F_1, F_2, \ldots, F_k$;

OUTPUT: An SLP $\mathbb{S}$ that derives text S;

Example



SLP Construction Example

Theorem (Rytter)

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

Features

- Sequential factors processing;
- Use AVL-trees as SLPs data representation;

SLP Construction Example

Theorem (Rytter)

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

Features

- Sequential factors processing;
- Use AVL-trees as SLPs data representation;

Bottleneck of The Algorithm

Sequential factors processing

At every iteration the algorithm concatenates potentially huge and small AVL-trees.

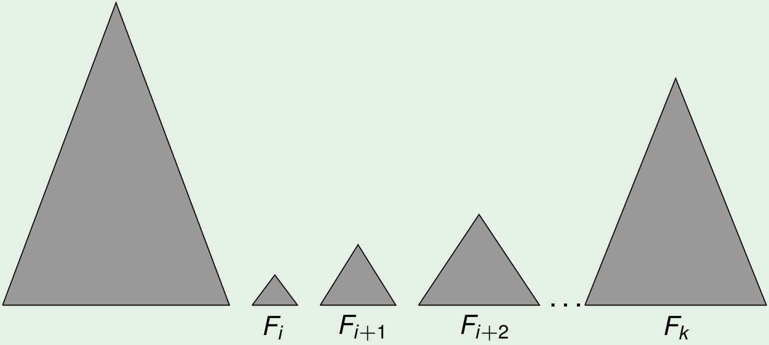
Example

Bottleneck of The Algorithm

Sequential factors processing

At every iteration the algorithm concatenates potentially huge and small AVL-trees.

Example



Modern Approach

Key idea

To optimize number of rotations by coupling factors into groups and calculating optimal concatenation order.

Theorem

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

Lemma

Given sequence of SLPs of length k, one can compute optimal order of concatenation using $O(k^3)$ time.

Modern Approach

Key idea

To optimize number of rotations by coupling factors into groups and calculating optimal concatenation order.

Theorem

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

Lemma

Given sequence of SLPs of length k, one can compute optimal order of concatenation using $O(k^3)$ time.

Modern Approach

Key idea

To optimize number of rotations by coupling factors into groups and calculating optimal concatenation order.

Theorem

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

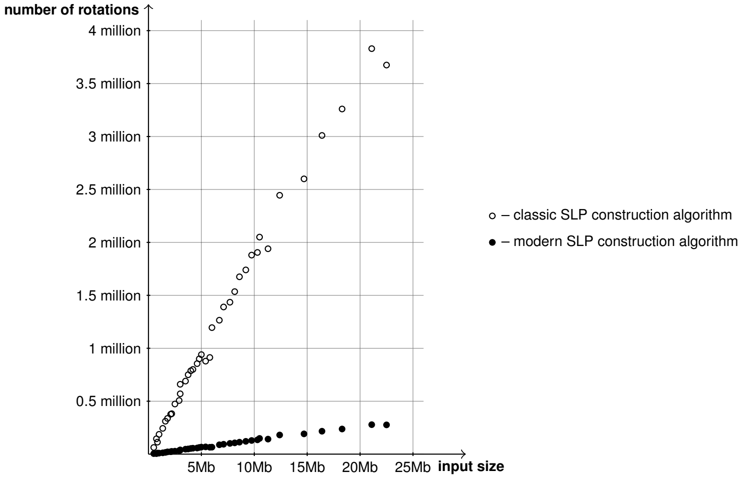
Lemma

Given sequence of SLPs of length k, one can compute optimal order of concatenation using $O(k^3)$ time.

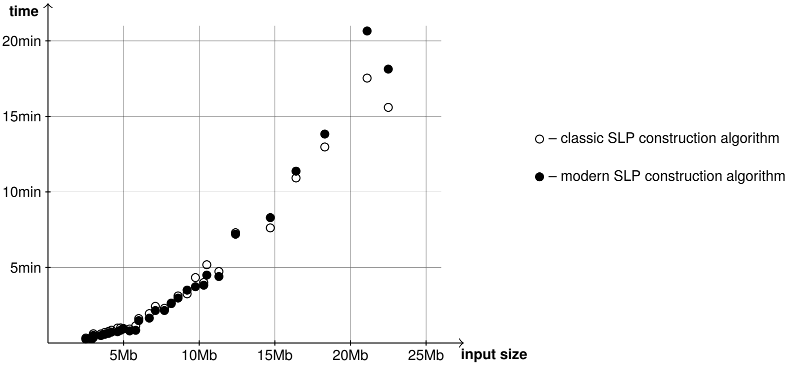
Types of Input Data

- Fibonacci words;
- DNA sequences from DNA Data Bank of Japan (`http://www.ddbj.nig.ac.jp/`);
- Random texts on 4-letters alphabet;

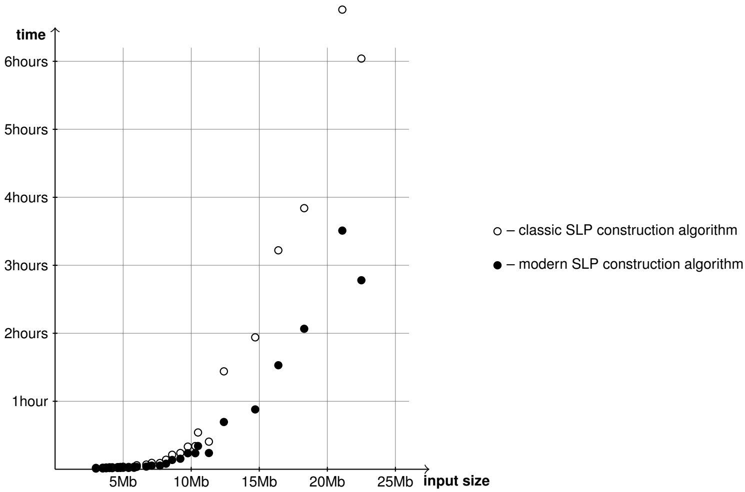
Number of Rotations Results



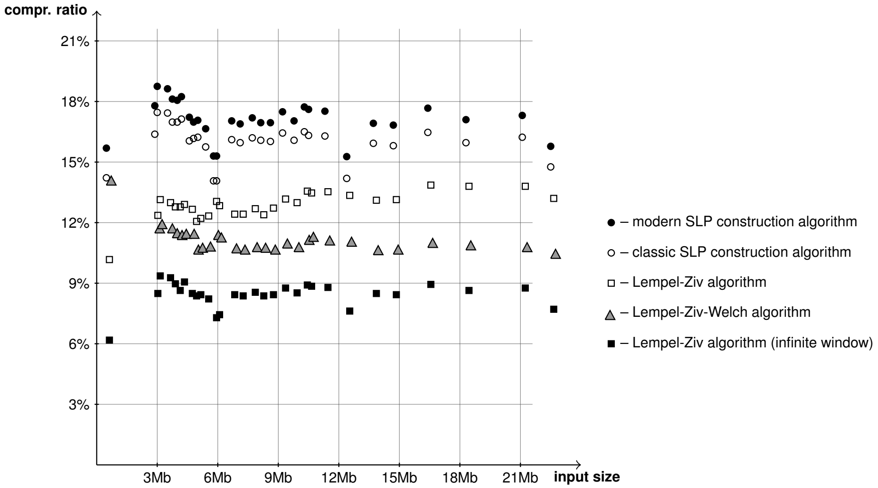
Speed Results in Operational Memory



Speed Results on File System



Compression Results



Conclusion

- We present modification of the algorithm for SLP Construction Problem;
- We compare performance of both algorithms on practise;
- We compare compression ratio provided by classic encoding algorithms and by two SLP-encoding algorithms;

Open Problem

How to optimally choose size of a group of factors?

Conclusion

- We present modification of the algorithm for SLP Construction Problem;
- We compare performance of both algorithms on practise;
- We compare compression ratio provided by classic encoding algorithms and by two SLP-encoding algorithms;

Open Problem

How to optimally choose size of a group of factors?

Questions?