Algorithms on compressed strings

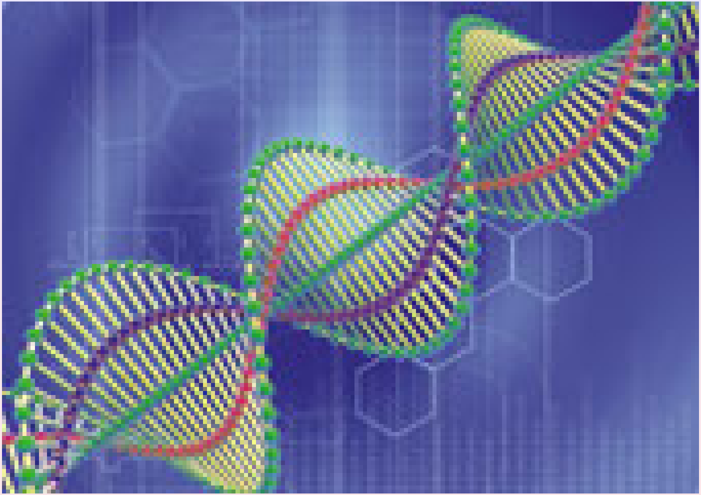
Lesha Khvorost

Indo-Russian Workshop On Algebra, Combinatorics and Complexity

3rd October 2008

Searching a criminal by his photo

Decoding DNA and feature extraction



Straight-line programs

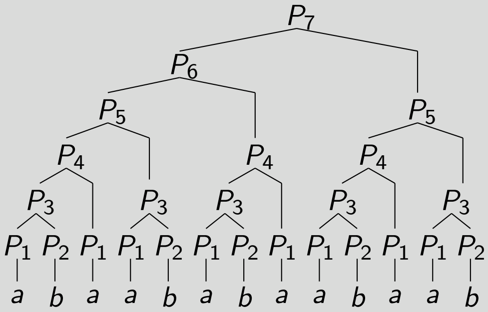
Definition

STRAIGHT-LINE PROGRAM (shortly SLP) over a terminal alphabet Σ is a context-free grammar $\mathcal{P}$ with ordered non-terminal symbols $P_1, \ldots, P_n$ (where P_n is the STARTING SYMBOL) such that there is exactly one production for each symbol: either $P_i \to a$, where $a \in \Sigma$ is a TERMINAL RULE, or $P_i \to P_l \cdot P_r$ for some $l, r < i$ is a NON-TERMINAL RULE.

The language generated by SLP $\mathcal{P}$ contains exactly one word.

Example

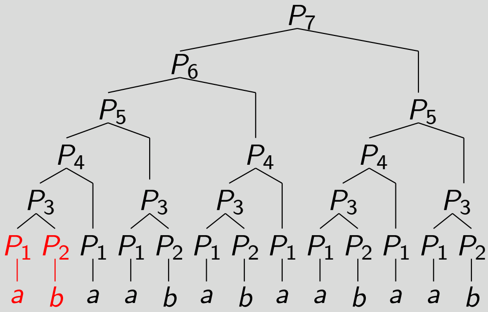
SLP that derives the 7-th word of Fibonacci:



Straight-line programs

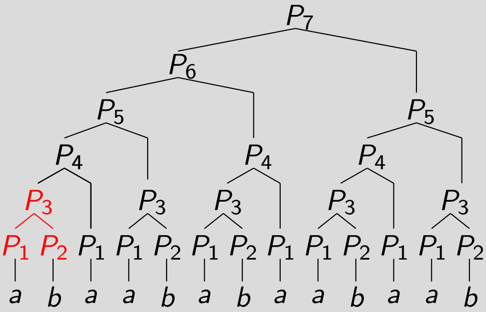
Example

SLP that derives the 7-th word of Fibonacci:



Example

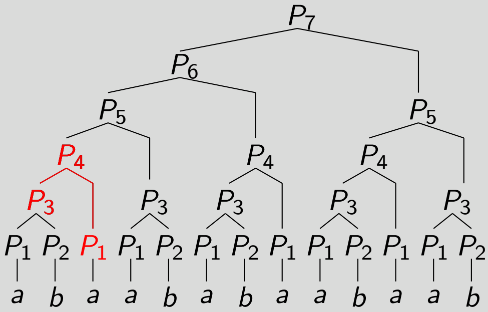
SLP that derives the 7-th word of Fibonacci:



Straight-line programs

Example

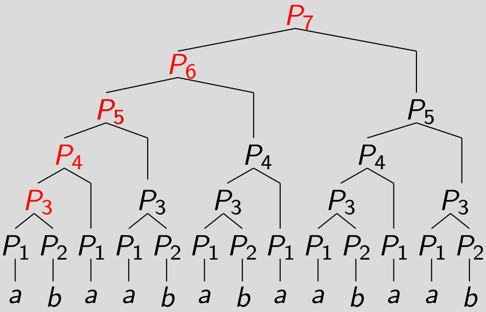
SLP that derives the 7-th word of Fibonacci:



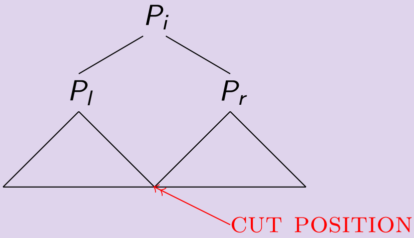
Straight-line programs

Example

SLP that derives the 7-th word of Fibonacci:



Cut position for non-terminal rule:



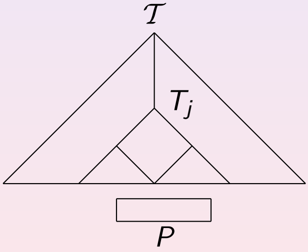
Pattern matching problem

INPUT: SLP $\mathcal{P}, \mathcal{T}$ that derive pattern P and text T correspondingly.
OUTPUT: compressed table that stores information about all occurrences P in T.

Pattern matching problem

Pattern localization idea:

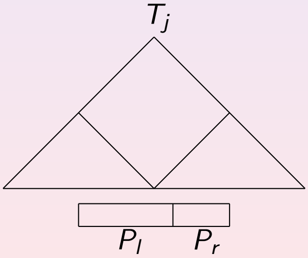
There is a unique rule $T_j \in \mathcal{T}$ that contains P and P touches its cut position.



Pattern matching problem

Iteration idea:

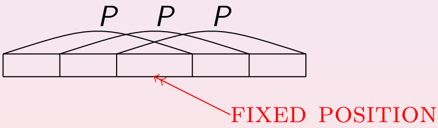
To find all occurrences of P inside T_j we need to know information about all occurrences of P_l and about all occurrences of P_r inside T_j.



Pattern matching problem

Idea of storing:

All occurrences P in T that touch some fixed position (for example, cut position) form an arithmetic progression.



Pattern matching problem

Result

This algorithm solves the pattern matching problem in $O(|\mathcal{P}| \cdot |\mathcal{T}|^2)$ time (it corresponds to $O(\log|P| \cdot \log|T|^2)$).

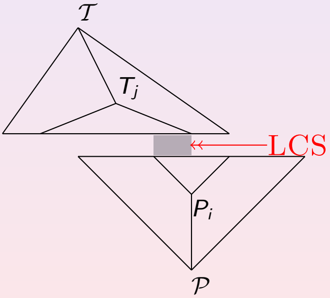
Longest common substring problem

INPUT: SLP $\mathcal{P}, \mathcal{T}$ that derive texts P and T correspondingly.
OUTPUT: SLP $\mathcal{S}$ that derives the longest common substring.

Longest common substring problem

Idea of localization

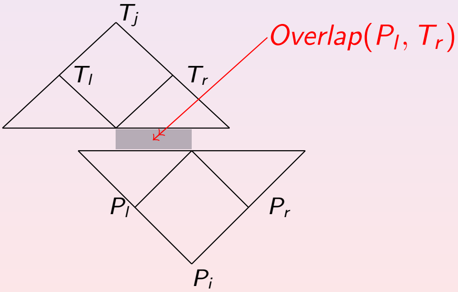
If the longest common substring is not empty, there exist rules $\mathcal{P}_i \in \mathcal{P}, \mathcal{T}_j \in \mathcal{T}$ such that the longest common substring locates between P_i and T_j and touches both cut positions.



Longest common substring problem

Idea of mutual disposition

Value of the longest common substring depends on mutual disposition of $\mathcal{P}_i$ relatively to $\mathcal{T}_j$. We need to compute and store all overlaps between every P_i and T_j efficiently. Next we extend every overlap efficiently.



Longest common substring problem

Result

This algorithm solves the longest common substring problem in $O(|\mathcal{P}|^4 \cdot \log|\mathcal{P}|)$ time (it corresponds to $O(\log|P|^4 \cdot \log\log|P|))$.

Palindrome searching problem

Definition

A non empty string P such that $P = P^R$ is said to be a palindrome where P^R denote the reversing string of P.

Palindrome searching problem

INPUT: SLP $\mathcal{P}$ that derives text P
OUTPUT: compressed table that stores information about all occurrences of palindromes in P

Palindrome searching problem

Idea of localization

If we fix some palindrome *pal* in P, there exist rule $\mathcal{P}_i \in \mathcal{P}$ which contains *pal* and the palindrome touches its cut position.

Idea of partition

For every rule $\mathcal{P}_i \in \mathcal{P}$ we can distinguish three sets of palindromes:

1. $PPals(\mathcal{P}_i)$ – set of prefix palindromes;
2. $IPals(\mathcal{P}_i)$ – set of inner palindromes (that touch cut position of $\mathcal{P}_i$);
3. $SPals(\mathcal{P}_i)$ – set of suffix palindromes;

Palindrome searching problem

Idea of gathering palindromes

For every rule $\mathcal{P}_i \in \mathcal{P}$ we need to extend sets $SPals$, $PPals$ efficiently, since next equality holds

$$IPals(\mathcal{P}_i) = Ext_{\mathcal{P}_i}(SPals(\mathcal{P}_l) \cup Ext_{\mathcal{P}_i}(PPals(\mathcal{P}_r)) \cup CPals(\mathcal{P}_i)),$$

where $CPals(\mathcal{P}_i)$ – set of palindromes which center position matches with cut position of $\mathcal{P}_i$.

Palindrome searching problem

Result

This algorithm solves the palindrome searching problem in $O(|\mathcal{P}|^4)$ time (it corresponds to $O(\log|P|^4)$).

Square searching problem

Definition

A non empty string xx is said to be a square.

Square searching problem

INPUT: SLP $\mathcal{P}$ that derives text P
OUTPUT: compressed structure that stores information about all occurrences of squares in P

Square searching problem

Idea of localization

If we fix some square xx in P, there exist rule $P_i \in \mathcal{P}$ which contains xx and the square touch its cut position.

Square searching problem

Idea of partition

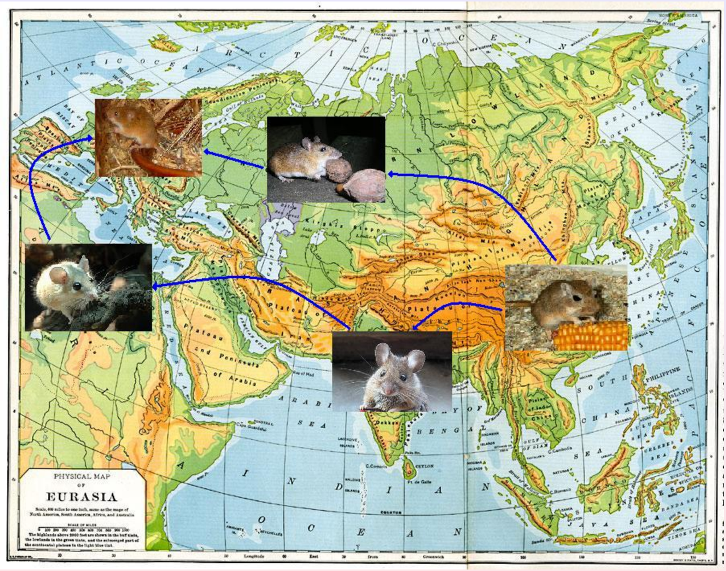
The range of length of searching squares depends on the length of text that derives from every rule. Hence we can partition every rule on polynomial number of parts and we'll process each part independently.

Square searching problem

Result

This algorithm solves the square searching problem in $O(|\mathcal{P}|^6)$ time (it corresponds to $O(\log |P|^6)$).

Genesis and migration of mouses problem



Hamming distance problem

INPUT: SLP $\mathcal{P}, \mathcal{T}$ that derive texts P and T of equal length
OUTPUT: value of Hamming distance between texts P and T

Hamming distance problem

Failure of accumulation idea

We can't use iteration by rules since mutual disposition of rules is very important.

Failure of partition idea

We can't partition rules on polynomial number of parts since searching objects (mismatches) are tiny.

Hamming distance problem

Result

Hamming distance problem is $\#P$-complete.

Final remarks

Connection with practice

SLP compression model is closer to representation that we obtain using family of Lempel-Ziv algorithms.

Connection with theory

Techniques that are applied in algorithms on compressed strings in terms of SLP are useful for theoretical proofs.

Efficiency of the model

Using this model we can find either "well-accumulated" (well-stored) objects or enough large objects (and their count doesn't matter). But problems which search (or count) large number of small objects are generally *NP*-hard.

Haven't you slept yet?