New Algorithms on Compressed Texts

Yury Lifshits

Steklov Institute of Mathematics, St.Petersburg, Russia
yura@logic.pdmi.ras.ru

Tallinn
20/03/2006
Fully Compressed Pattern Matching (FCPM)

**INPUT:** Compressed strings \( P \) and \( T \)

**OUTPUT:** Yes/No (whether \( P \) is a substring in \( T \)?)

**Example**

<table>
<thead>
<tr>
<th>Text</th>
<th>abaababaabaab</th>
<th>We know only compressed representation of ( P ) and ( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>baba</td>
<td></td>
</tr>
</tbody>
</table>
Fully Compressed Pattern Matching (FCPM)

**INPUT:** Compressed strings $P$ and $T$

**OUTPUT:** Yes/No (whether $P$ is a substring in $T$?)

**Example**

**Text:** abaa**baba**abaab  
**Pattern:** **baba**

We know only compressed representation of $P$ and $T$
Outline of the Talk

1. Processing Compressed Texts: Bird’s Eye View

2. Fully Compressed Pattern Matching: Idea of a New Algorithm
   - Idea of a new algorithm
   - ★ Detailed description

3. More Algorithms and Some Negative Results

4. Conclusions and Open Problems
Central idea

If some text is highly compressible, then it contains long identical segments and therefore it is likely that we can solve some problems more efficiently than in general case.
**Motivation**

**Reasons for algorithms on compressed texts:**
- Potentially faster than “unpack-and-solve”
Motivation

Reasons for algorithms on compressed texts:

- Potentially faster than “unpack-and-solve”
- Lower memory requirements
Motivation

Reasons for algorithms on compressed texts:

- Potentially faster than “unpack-and-solve”
- Lower memory requirements
- Theoretical applications: word equations in PSPACE, pattern matching in message sequence charts
Motivation

Reasons for algorithms on compressed texts:
- Potentially faster than “unpack-and-solve”
- Lower memory requirements
- Theoretical applications: word equations in PSPACE, pattern matching in message sequence charts

Real data with high level of repetitions:
- Genomes
Motivation

Reasons for algorithms on compressed texts:
- Potentially faster than “unpack-and-solve”
- Lower memory requirements
- Theoretical applications: word equations in PSPACE, pattern matching in message sequence charts

Real data with high level of repetitions:
- Genomes
- Internet logs, any statistical data
Motivation

Reasons for algorithms on compressed texts:
- Potentially faster than “unpack-and-solve”
- Lower memory requirements
- Theoretical applications: word equations in PSPACE, pattern matching in message sequence charts

Real data with high level of repetitions:
- Genomes
- Internet logs, any statistical data
- Automatically generated texts
Straight-line Programs

**Straight-line program** (SLP) is a
Context-free grammar generating *exactly one* string
Two types of productions: $X_i \rightarrow a$ and $X_i \rightarrow X_pX_q$
Straight-Line Programs

**Straight-line program** (SLP) is a
Context-free grammar generating **exactly one** string
Two types of productions: $X_i \rightarrow a$ and $X_i \rightarrow X_p X_q$

Most of practically used compression algorithms (Lempel-Ziv family, run-length encoding...) can be efficiently translated to SLP
Straight-Line Programs

**Straight-line program** (SLP) is a context-free grammar generating **exactly one** string.
Two types of productions: $X_i \rightarrow a$ and $X_i \rightarrow X_pX_q$

Most of practically used compression algorithms (Lempel-Ziv family, run-length encoding...) can be efficiently translated to SLP

**Example**

abaababaabaab

$X_1 \rightarrow b$, $X_2 \rightarrow a$
$X_3 \rightarrow X_2X_1$, $X_4 \rightarrow X_3X_2$
$X_5 \rightarrow X_4X_3$, $X_6 \rightarrow X_5X_4$
$X_7 \rightarrow X_6X_5$
Important Related Results

Algorithms on compressed texts:

- **Amir et al.'94**: Compressed Pattern Matching
- **Gaśieniec et al.'96**: Regular Language Membership
Important Related Results

Algorithms on compressed texts:
- **Amir et al.’94**: Compressed Pattern Matching
- **Gąsieniec et al.’96**: Regular Language Membership

The following problems are hard for compressed texts:
- **Lohrey’04**: Context-Free Language Membership
- **Berman et al.’02**: Two-dimensional Compressed Pattern Matching
Fully Compressed Pattern Matching (FCPM)

INPUT: SLP-compression of $P$ and of $T$
OUTPUT: Yes/No (whether $P$ is a substring in $T$?)
Fully Compressed Pattern Matching (FCPM)

**INPUT:** SLP-compression of $P$ and of $T$

**OUTPUT:** Yes/No (whether $P$ is a substring in $T$?)

Let $m$ and $n$ be the sizes of straight-line programs generating correspondingly $P$ and $T$

- **Gaśieniec et al.'96:** $O((n + m)^5 \log^3 |T|)$ algorithm
- **Miyazaki et al.'97:** $O(n^2 m^2)$ algorithm
Fully Compressed Pattern Matching (FCPM)

**INPUT:** SLP-compression of $P$ and of $T$

**OUTPUT:** Yes/No (whether $P$ is a substring in $T$?)

Let $m$ and $n$ be the sizes of straight-line programs generating correspondingly $P$ and $T$

- **Gąsieniec et al.’96:** $O((n + m)^5 \log^3 |T|)$ algorithm
- **Miyazaki et al.’97:** $O(n^2 m^2)$ algorithm
- **Lifshits’06:** $O(n^2 m)$ algorithm
Basic Lemma

Notation:

Position = place between neighbor characters.
Occurrence = starting position of a substring
Basic Lemma

Notation:

Position = place between neighbor characters.
Occurrence = starting position of a substring

Lemma

All occurrences of $P$ in $T$ touching any given position form a single arithmetical progression

\[ P \quad P \quad P \]

Common position
Let $P_1, \ldots, P_m$ and $T_1, \ldots, T_n$ be the compression symbols.
Let $P_1, \ldots, P_m$ and $T_1, \ldots, T_n$ be the compression symbols.

A **cut** is a merging position for $X_i = X_rX_s$. 

**AP-table**

For every $1 \leq i \leq m$, $1 \leq j \leq n$ let $AP[i, j]$ be a code of ar.pr. of occurrences of $P_i$ in $T_j$ that touches the cut of $T_j$. $P_1 \ldots P_m T_1 \ldots T_n P_i T_j$.
Let $P_1, \ldots, P_m$ and $T_1, \ldots, T_n$ be the compression symbols.

A cut is a merging position for $X_i = X_rX_s$.

**AP-table:**
For every $1 \leq i \leq m, 1 \leq j \leq n$ let $AP[i,j]$ be a code of ar.pr. of occurrences of $P_i$ in $T_j$ that touches the cut of $T_j$.
Two Claims

**Claim 1:** We can solve all variants of FCPM from AP-table in linear time:
- Find the first occurrence
- Count the number of all occurrences
- Check whether there is an occurrence from the given position
- Compute a “compressed” representation of all occurrences
Two Claims

**Claim 1:** We can solve all variants of FCPM from AP-table in linear time:
- Find the first occurrence
- Count the number of all occurrences
- Check whether there is an occurrence from the given position
- Compute a “compressed” representation of all occurrences

**Claim 2:** We can compute the whole AP-table by dynamic programming method using $O(n)$ time for every element
Getting the answer

AP-table:
for every $1 \leq i \leq m, 1 \leq j \leq n$ let $AP[i, j]$ be a code of ar.pr. of occurrences of $P_i$ in $T_j$ that touches the cut of $T_j$
Getting the answer

AP-table:
for every $1 \leq i \leq m, 1 \leq j \leq n$ let $AP[i, j]$ be a code of ar.pr. of occurrences of $P_i$ in $T_j$
that touches the cut of $T_j$

How to check whether $P$ occurs in $T$ from AP-table?
Getting the answer

AP-table:

for every $1 \leq i \leq m, 1 \leq j \leq n$ let $AP[i, j]$ be a code of ar.pr. of occurrences of $P_i$ in $T_j$
that touches the cut of $T_j$

How to check whether $P$ occurs in $T$ from AP-table?

Answer:

$P$ occurs in $T$ iff there is $j$ such that $AP[m, j]$ is nonempty
Computing AP-table

Order of computation:
from $j=1$ to $n$ do
  from $i=1$ to $m$ do
    compute $AP[i,j]$
Computing AP-table

Order of computation:
from $j=1$ to $n$ do
  from $i=1$ to $m$ do
    compute $AP[i,j]$

Basis: one-letter $P_i$ or one-letter $T_j$
Induction step: $P_i$ and $T_j$ are composite texts
Computing AP-table

Order of computation:
from \( j=1 \) to \( n \) do
  from \( i=1 \) to \( m \) do
    compute \( \text{AP}[i,j] \)

Basis: one-letter \( P_i \) or one-letter \( T_j \)

Induction step: \( P_i \) and \( T_j \) are composite texts

We design a special auxiliary procedure that extracts useful information from already computed part of AP-table for computing a new element \( \text{AP}[i,j] \)
Auxiliary Procedure: Local PM

$LocalPM(i, j, [\alpha, \beta])$ returns occurrences of $P_i$ in $T_j$ inside the interval $[\alpha, \beta]$
Auxiliary Procedure: Local PM

LocalPM\((i, j, [\alpha, \beta])\) returns occurrences of \(P_i\) in \(T_j\) inside the interval \([\alpha, \beta]\)

Important properties:

- Local PM uses values \(AP[i,k]\) for \(1 \leq k \leq j\)
- It is defined only when \(|\beta - \alpha| \leq 3|P_i|\)
- It works in time \(O(n)\)
- The output of Local PM is a pair of ar.pr.
**Auxiliary Procedure: Local PM**

*LocalPM*(i, j, [α, β]) returns occurrences of *P* in *T* inside the interval [α, β]

**Important properties:**
- Local PM uses values AP[i,k] for 1 ≤ k ≤ j
- It is defined only when |β − α| ≤ 3|P|
- It works in time *O*(n)
- The output of Local PM is a pair of ar.pr.

**Proposition:** answer of Local PM indeed could be always represented by pair of ar.pr.
Computing the next element

Let $P_i = P_r P_s$, and let $|P_r| \geq |P_s|$
Computing the next element

Let $P_i = P_r P_s$, and let $|P_r| \geq |P_s|$

Naive approach

1. Compute all occurrences of $P_r$ around cut of $T_j$
Computing the next element

Let $P_i = P_r P_s$, and let $|P_r| \geq |P_s|$

**Naive approach**

1. Compute all occurrences of $P_r$ around cut of $T_j$
2. Compute all occurrences of $P_s$ around cut of $T_j$
Computing the next element

Let $P_i = P_rP_s$, and let $|P_r| \geq |P_s|$.

**Naive approach**

1. Compute all occurrences of $P_r$ around cut of $T_j$
2. Compute all occurrences of $P_s$ around cut of $T_j$
3. Shift the latter by $|P_r|$ and intersect
Computing the next element

Let $P_i = P_rP_s$, and let $|P_r| \geq |P_s|$

**Naive approach**

1. Compute all occurrences of $P_r$ around cut of $T_j$
2. Compute all occurrences of $P_s$ around cut of $T_j$
3. Shift the latter by $|P_r|$ and intersect
Computing the next element

Let $P_i = P_r P_s$, and let $|P_r| \geq |P_s|$

Naive approach

1. Compute all occurrences of $P_r$ around cut of $T_j$
2. Compute all occurrences of $P_s$ around cut of $T_j$
3. Shift the latter by $|P_r|$ and intersect

Remark: we can do only step 1 by Local PM
Computing the next element

Let \( P_i = P_r P_s \), and let \( |P_r| \geq |P_s| \)

**Naive approach**

1. Compute all occurrences of \( P_r \) around cut of \( T_j \)
2. Compute all occurrences of \( P_s \) around cut of \( T_j \)
3. Shift the latter by \( |P_r| \) and intersect

**Remark:** we can do only step 1 by Local PM

**Idea:** not all occurrences of \( P_s \) but only these that are starting at the ends of \( P_r \) ones.
Computing the next element II

Some blackboard explanation...
Computing the next element II

Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
2. Divide all ends to “continental” and “seaside”
Computing the next element II

Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
2. Divide all ends to “continental” and “seaside”
3. Check one continental

Total complexity: Local PM for $P_r$ + $2$ Local PM for $P_s$ + $2$ point checks for $P_s$

$O(n)$

We are done!

(Modulo basic computation of AP-table and realization of Local PM)

Yury Lifshits (Steklov Inst. of Math)

New Algorithms on Compressed Texts

Tallinn’06 16 / 23
Computing the next element II

Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
2. Divide all ends to “continental” and “seaside”
3. Check one continental
4. Check all seaside (by Local PM)
Computing the next element II

Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
2. Divide all ends to “continental” and “seaside”
3. Check one continental
4. Check all seaside (by Local PM)
5. The same for the second ar.pr.

Total complexity: $O(n)$
Computing the next element II

Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
2. Divide all ends to “continental” and “seaside”
3. Check one continental
4. Check all seaside (by Local PM)
5. The same for the second ar.pr.

Total complexity:

- Local PM for $P_r$
- 2 Local PM for $P_s$
- 2 point checks for $P_s$

$O(n)$

We are done!

(Modulo basic computation of AP-table and realization of Local PM)
Computing the next element II

Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
2. Divide all ends to “continental” and “seaside”
3. Check one continental
4. Check all seaside (by Local PM)
5. The same for the second ar.pr.

**Total complexity:**

Local PM for $P_r$
+ 2 Local PM for $P_s$
+ 2 point checks for $P_s$

$O(n)$
Computing the next element II

Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
2. Divide all ends to “continental” and “seaside”
3. Check one continental
4. Check all seaside (by Local PM)
5. The same for the second ar.pr.

**Total complexity:**
- Local PM for $P_r$
- + 2 Local PM for $P_s$
- + 2 point checks for $P_s$

$O(n)$

We are done!
Computing the next element II

Some blackboard explanation...

1. Take the first ar.pr of $P_r$ occurrences
2. Divide all ends to “continental” and “seaside”
3. Check one continental
4. Check all seaside (by Local PM)
5. The same for the second ar.pr.

Total complexity:

- Local PM for $P_r$
- + 2 Local PM for $P_s$
- + 2 point checks for $P_s$
- $O(n)$

We are done! (Modulo basic computation of AP-table and realization of Local PM)
Covers and Periods

A **period** of a string $T$ is a string $W$ such that $T$ is a prefix of $W^k$ for some integer $k$.
Covers and Periods

A **period** of a string $T$ is a string $W$ such that $T$ is a prefix of $W^k$ for some integer $k$

![Diagram of a period]

A **cover** of a string $T$ is a string $C$ such that any character in $T$ is covered by some occurrence of $C$ in $T$

![Diagram of a cover]
Covers and Periods

A **period** of a string $T$ is a string $W$ such that $T$ is a prefix of $W^k$ for some integer $k$.

A **cover** of a string $T$ is a string $C$ such that any character in $T$ is covered by some occurrence of $C$ in $T$.

**Compressed Periods/Covers:** given a compressed string $T$, to find the shortest period/cover and compute a “compressed” representation of all periods/covers.
Subsequence Problems

**Compressed Window Subsequence:** given a pattern \( P \), a compressed string \( T \), and an integer \( k \), to determine whether \( P \) is a scattered subsequence in some window of length \( k \) in the text \( T \)

**Example**

\[
\begin{align*}
T &: \text{ abaababaabaab} \\
P &: \text{ babab} \\
k &: \text{ 6}
\end{align*}
\]
Subsequence Problems

**Compressed Window Subsequence:** given a pattern $P$, a compressed string $T$, and an integer $k$, to determine whether $P$ is a scattered subsequence in some window of length $k$ in the text $T$

**Example**

$T$: abaa | **babaab** | aab

$P$: babab

$k$: 6
Subsequence Problems

**Compressed Window Subsequence:** given a pattern $P$, a compressed string $T$, and an integer $k$, to determine whether $P$ is a **scattered** subsequence in some window of length $k$ in the text $T$

**Example**

$T$: abaa | **babaab** | aab  
$P$: babab  
$k$: 6

**Fully Compressed Subsequence Problem:** given compressed strings $P$ and $T$, to determine whether $P$ is a **scattered** subsequence in $T$

**Example**

$T$: abaababaabaab
$P$: baabaabaab
Subsequence Problems

**Compressed Window Subsequence:** given a pattern $P$, a compressed string $T$, and an integer $k$, to determine whether $P$ is a scattered subsequence in some window of length $k$ in the text $T$

**Example**

$T$: abaa|babaab|aab  
$P$: babab  
k: 6

**Fully Compressed Subsequence Problem:** given compressed strings $P$ and $T$, to determine whether $P$ is a scattered subsequence in $T$

**Example**

$T$: abaababaabaab  
$P$: baabaabaab
Hamming Distance and LCS

**Compressed Hamming Distance:** given compressed strings $T_1$ and $T_2$, to compute Hamming distance (the number of characters which differ) between them.

**Example**

$T_1$: abaababaabaab

$T_2$: baababababaab
Hamming Distance and LCS

**Compressed Hamming Distance:** given compressed strings $T_1$ and $T_2$, to compute Hamming distance (the number of characters which differ) between them

**Example**

$T_1$: \textcolor{red}{abaababa}abaab \quad HD(T_1, T_2) = 7$
$T_2$: \textcolor{red}{baababab}abaab
Hamming Distance and LCS

**Compressed Hamming Distance:** given compressed strings $T_1$ and $T_2$, to compute Hamming distance (the number of characters which differ) between them

**Example**

$T_1$: abaababaabaab

$T_2$: baababababaab

$HD(T_1, T_2) = 7$

**Compressed LCS:** given compressed strings $T_1$ and $T_2$, to compute the length of the longest common subsequence

**Example**

$T_1$: abaababaabaab

$T_2$: baababababaab
Hamming Distance and LCS

**Compressed Hamming Distance:** given compressed strings $T_1$ and $T_2$, to compute Hamming distance (the number of characters which differ) between them

**Example**

$T_1$: abaababaabaab  
$T_2$: baababababaab

$HD(T_1, T_2) = 7$

**Compressed LCS:** given compressed strings $T_1$ and $T_2$, to compute the length of the longest common subsequence

**Example**

$T_1$: abaababaabaab  
$T_2$: baababababaab

$LCS(T_1, T_2) = 12$
A **fingerprint** is a set of used characters of any substring of $T$. A **fingerprint table** is the set of all fingerprints.

**Example**

**Text:** abacaba
A **fingerprint** is a set of used characters of any substring of $T$. A **fingerprint table** is the set of all fingerprints.

**Example**

**Text:** abacaba

**Fingerprint Table:** $\emptyset \{a\} \{b\} \{c\} \{a,b\} \{a,c\} \{a,b,c\}$
A **fingerprint** is a set of used characters of any substring of $T$. A **fingerprint table** is the set of all fingerprints.

**Example**

**Text:** abacaba

**Fingerprint Table:** $\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{a, b, c\}$

**Compressed Fingerprint Table:** given a compressed string $T$, to compute a fingerprint table
Check Your Intuition

Which of the following problems have polynomial algorithms?

1. Periods
2. Longest Common Subsequence
3. Hamming distance
4. Covers
5. Fingerprint Table
6. Compressed Window Subsequence
7. Fully Compressed Subsequence Problem

Answer: red-on-grey problems have polynomial algorithms, black ones are NP-hard.

Yury Lifshits (Steklov Inst. of Math)
New Algorithms on Compressed Texts
Tallinn’06
Check Your Intuition

Which of the following problems have polynomial algorithms?

1. Periods
2. Longest Common Subsequence
3. Hamming distance
4. Covers
5. Fingerprint Table
6. Compressed Window Subsequence
7. Fully Compressed Subsequence Problem

Answer: **red-on-grey** problems have polynomial algorithms, black ones are NP-hard
Main points:

- New field: algorithms working on compressed objects (including strings) without unpacking them
Summary

Main points:

- New field: algorithms working on compressed objects (including strings) without unpacking them
- New algorithm: fully compressed pattern matching in cubic time
Summary

Main points:

- New field: algorithms working on compressed objects (including strings) without unpacking them
- New algorithm: fully compressed pattern matching in cubic time
Summary

Main points:
- New field: algorithms working on compressed objects (including strings) without unpacking them
- New algorithm: fully compressed pattern matching in cubic time

Open Problems
- To construct a $O(nm \log |T|)$ algorithm for Fully Compressed Pattern Matching
- To construct $O(nm)$ algorithms for edit distance, where $n$ is the length of $T_1$ and $m$ is the compressed size of $T_2$
Yu. Lifshits
Solving Classical String Problems on Compressed Texts.
*Draft, 2006.*

Yu. Lifshits and M. Lohrey
Querying and Embedding Compressed Texts.
*to be submitted, 2006.*

P. Cégielski, I. Guessarian, Yu. Lifshits and Yu. Matiyasevich
Window Subsequence Problems for Compressed Texts.
accepted to “Computer Science in Russia”, 2006.
Contact: Yury Lifshits
Email: yura@logic.pdmi.ras.ru
Home page: http://logic.pdmi.ras.ru/~yura/

Yu. Lifshits
Solving Classical String Problems on Compressed Texts.
*Draft*, 2006.

Yu. Lifshits and M. Lohrey
Querying and Embedding Compressed Texts.
*to be submitted*, 2006.

P. Cégielski, I. Guessarian, Yu. Lifshits and Yu. Matiyasevich
Window Subsequence Problems for Compressed Texts.
accepted to “Computer Science in Russia”, 2006.

Thanks for attention.
Contact: **Yury Lifshits**
Email: yura@logic.pdmi.ras.ru
Home page: http://logic.pdmi.ras.ru/~yura/

Yu. Lifshits
Solving Classical String Problems on Compressed Texts.
*Draft, 2006.*

Yu. Lifshits and M. Lohrey
Querying and Embedding Compressed Texts.
*to be submitted, 2006.*

P. Cégielski, I. Guessarian, Yu. Lifshits and Yu. Matiyasevich
Window Subsequence Problems for Compressed Texts.
accepted to “Computer Science in Russia”, 2006.

Thanks for attention. **Questions?**