Combinatorial Methods for String and Graph

Combinatorial algorithms for grammar-based text compression

Shunsuke Inenaga
Kyushu University, Japan
Agenda

• Highly Repetitive Strings
• Grammar-based String Compression
• Straight-Line Program (SLP)
• Compressed String Processing (CSP) on SLP
• Reviews on CSP Algorithms and Related Results
• Conclusions and Future Work
**Highly Repetitive Strings (HRSs)**

**HRSs** are strings that contain a lot of repeats. Repeats may occur separately (not necessarily tandem).

Examples of HRSs are:

- Collection of DNA sequences of same species (two human genomes are 99.9% same)
- Software repositories (GitHub)
- Versioned documents (Wikipedia)

![ID DNA sequences]

<table>
<thead>
<tr>
<th>ID</th>
<th>DNA sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CATCTCCATCATCACCACCCTCCTCCTCAT...</td>
</tr>
<tr>
<td>2</td>
<td>CATCCCCATCATCACCACCCTCCTCCTCAT...</td>
</tr>
<tr>
<td>3</td>
<td>CATCTCCATCATCACTACCCTCCTCCTCAT...</td>
</tr>
<tr>
<td>4</td>
<td>CATCTCCATAATCACCACCCTCCTCCTCCTCAT...</td>
</tr>
<tr>
<td>5</td>
<td>CATCTCCATCATCACCACCCTCCTCCTACTCAT...</td>
</tr>
<tr>
<td>6</td>
<td>CATCTCCATCAACACCACCCTCCTCCTTTAT...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
**Statistical Compressors vs. HRS**

\[ H_k : k\text{-th order empirical entropy} \quad (k < N) \]

The \( k \)-th order empirical entropy \( H_k \) captures the dependence of symbols upon their \( k \)-long context.

\[ \text{Say it again plz?} \]

\[ 1101001100010000010010110101100 \]

\[ N \text{ characters} \]

\[ NH_k \text{ bits} \]
Statistical Compressors vs. HRS

The $k$-th order entropy model does not capture repetitiveness very well [Kreft & Navarro 2013].

But obviously, the above repetitive string can be represented with $NH_k + O(1)$ bits!
Grammar-based Compression [Keiffer & Yang 2000]

String $w$: $\text{abbbbaabbaabbbbaabbbbabb}$

Grammar Transform

CFG $G$:
- $S \rightarrow ACBBEA$
- $A \rightarrow Db$
- $B \rightarrow Cb$
- $C \rightarrow aD$
- $D \rightarrow aE$
- $E \rightarrow bb$

Grammar-compression in the Chomsky normal form is called an SLP (Straight Line Program)

Encoding for $G$: 0100010100010110101011101010110
Grammar-based Compression [Keiffer & Yang 2000]

String \( w \)

\[ w = \text{abbbaabbaabbbbaabbbbabbb} \]

Grammar Transform

CFG \( G \)

Derives only \( w \)

\[
\begin{align*}
S & \rightarrow ACBBEA \\
A & \rightarrow Db \\
B & \rightarrow Cb \\
C & \rightarrow aD \\
D & \rightarrow aE \\
E & \rightarrow bb
\end{align*}
\]

Grammar-compression in the Chomsky normal form is called an SLP (Straight Line Program)

This Talk

Encoding

Encoding for \( G \)

\[ 0100010100010110101011101010110 \]
An SLP is a sequence of \( n \) productions

\[
X_1 \rightarrow expr_1, \ X_2 \rightarrow expr_2, \cdots, \ X_n \rightarrow expr_n
\]

- \( expr_i = a \) \quad (a \in \Sigma) \n- \( expr_i = X_l X_r \) \quad (l, r < i)

SLP is grammar-compression in the Chomsky normal form.

- SLP is a widely accepted model for the outputs of grammar-based compressors.
- The size of the SLP is the number \( n \) of productions.
Example of SLP

SLP $S$

$X_7 \rightarrow X_5 X_6$
$X_6 \rightarrow X_5 X_4$
$X_5 \rightarrow X_3 X_4$
$X_4 \rightarrow X_1 X_2$
$X_3 \rightarrow X_1 X_1$
$X_2 \rightarrow b$
$X_1 \rightarrow a$

Derivation tree $T$ of SLP $S$

String represented by SLP $S$
DAG view of SLP

DAG for SLP $S$  Derivation tree $T$ of SLP $S$

This DAG is equivalent to the set of productions.
DAG view of SLP

DAG for SLP $S$  Derivation tree $T$ of SLP $S$

✓ Grammar-based compression captures repetitiveness in the string.
Grammar-based compression captures repetitiveness in the string.
Grammar-based Compressors

Computing the *smallest grammar* is NP-hard [Storer 1978].

$O(\log(N/g))$ approximation ($g$ is the smallest grammar size)

- AVL grammar [Rytter 2003]
- $\alpha$-balanced grammar [Charikar et al. 2005]
- Recompression [Jez 2015]

Greedy Algorithms

- LZ78 [Ziv & Lempel 1978]
- Re-Pair [Larsson & Moffat 2000]
- Longest Match [Nakamura et al. 2009]

Locally Consistent Parsing

- ESP-grammar [Sakamoto et al. 2009]
- Recompression [Jez 2015]

Note: This list is far from being comprehensive.
Grammar-based Compressors

Computing the smallest grammar is NP-hard [Storer 1978].

\[ O(\log(N/g)) \] approximation (\( g \) is the smallest grammar size)
- AVL grammar [Rytter 2003]
- \( \alpha \)-balanced grammar [Charikar et al. 2005]
- Recompression [Jez 2015]

Greedy Algorithms
- LZ78 [Ziv & Lempel 1978]
- Re-Pair [Larsson & Moffat 2000]
- Longest Match [Nakamura et al. 2009]

Locally Consistent Parsing
- ESP-grammar [Sakamoto et al. 2009]
- Recompression [Jez 2015]

Note: This list is far from being comprehensive.
Re-Pair [Larsson & Moffat 2000]

\[ w = \text{a b c d x y a b c d x y a b c d} \]

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).
Re-Pair [Larsson & Moffat 2000]

\[ w = \begin{array}{cccccccccccc}
X_1 & c & d & x & y & X_1 & c & d & x & y & X_1 & c & d \\
\hline
a & b & c & d & x & y & a & b & c & d & x & y & a & b & c & d & X_1 \to a & b
\end{array} \]

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).
Re-Pair [Larsson & Moffat 2000]

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).
Re-Pair [Larsson & Moffat 2000]

<table>
<thead>
<tr>
<th></th>
<th>X₃</th>
<th>x</th>
<th>y</th>
<th>X₃</th>
<th>x</th>
<th>y</th>
<th>X₃</th>
<th>X₃ → X₂ d</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₂</td>
<td>d</td>
<td>x</td>
<td>y</td>
<td>X₂</td>
<td>d</td>
<td>x</td>
<td>y</td>
<td>X₂ → X₁ c</td>
</tr>
<tr>
<td>X₁</td>
<td>c</td>
<td>d</td>
<td>x</td>
<td>y</td>
<td>X₁</td>
<td>c</td>
<td>d</td>
<td>X₁ → a b</td>
</tr>
<tr>
<td>w</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>x</td>
<td>y</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).
Re-Pair [Larsson & Moffat 2000]

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>y</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_4$</td>
<td>y</td>
<td>$X_4$</td>
<td>y</td>
<td>$X_3$</td>
<td>$X_4 \to X_3 x$</td>
<td></td>
</tr>
<tr>
<td>$X_3$</td>
<td>x</td>
<td>y</td>
<td>$X_3$</td>
<td>x</td>
<td>y</td>
<td>$X_3 \to X_2 d$</td>
</tr>
<tr>
<td>$X_2$</td>
<td>d</td>
<td>x</td>
<td>y</td>
<td>$X_2$</td>
<td>d</td>
<td>x</td>
</tr>
<tr>
<td>$X_1$</td>
<td>c</td>
<td>d</td>
<td>x</td>
<td>y</td>
<td>$X_1$</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>x</td>
<td>y</td>
<td>a</td>
</tr>
</tbody>
</table>

$w =$
Re-Pair [Larsson & Moffat 2000]

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).
Re-Pair [Larsson & Moffat 2000]

<table>
<thead>
<tr>
<th></th>
<th>$S$</th>
<th>$S \rightarrow X_5X_5X_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_5$</td>
<td>$X_5$</td>
<td>$X_5 \rightarrow X_4y$</td>
</tr>
<tr>
<td>$X_4$</td>
<td>$y$</td>
<td>$X_4 \rightarrow X_3x$</td>
</tr>
<tr>
<td>$X_3$</td>
<td>$x$</td>
<td>$X_3 \rightarrow X_2d$</td>
</tr>
<tr>
<td>$X_2$</td>
<td>$d$</td>
<td>$X_2 \rightarrow X_1c$</td>
</tr>
<tr>
<td>$X_1$</td>
<td>$c$</td>
<td>$X_1 \rightarrow a\ b$</td>
</tr>
</tbody>
</table>

$w =$ \[
\begin{array}{cccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{x} & \text{y} & \text{a} & \text{b} & \text{c} & \text{d} \\
\end{array}
\]

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).
Re-Pair outperforms $H_k$ on real-world repetitive strings.

<table>
<thead>
<tr>
<th>File</th>
<th>Re-Pair</th>
<th>$H_0$</th>
<th>$H_4$</th>
<th>$H_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA sequences (influenza)</td>
<td>3.31%</td>
<td>24.63%</td>
<td>23.88%</td>
<td>13.25%</td>
</tr>
<tr>
<td>Source Codes (kernel)</td>
<td>1.13%</td>
<td>67.25%</td>
<td>19.25%</td>
<td>7.75%</td>
</tr>
<tr>
<td>Wikipedia (Einstein)</td>
<td>0.10%</td>
<td>62.00%</td>
<td>13.25%</td>
<td>3.50%</td>
</tr>
<tr>
<td>Documents (CIA world leaders)</td>
<td>1.78%</td>
<td>43.38%</td>
<td>7.63%</td>
<td>3.13%</td>
</tr>
</tbody>
</table>

This is a part of results reported in the statistics on Pizza and Chile highly-repetitive corpus. http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf
## Approximation to Smallest Grammar

Approximation ratios of grammar-based compressors to the smallest grammar of size $g$

<table>
<thead>
<tr>
<th>Compressors</th>
<th>Upper bound</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-Pair</td>
<td>$O\left(\frac{N}{\log N}^{2/3}\right)$ [1]</td>
<td>$\Omega\left(\frac{\log N}{\log\log N}\right)$ [2]</td>
</tr>
<tr>
<td>LongestMatch</td>
<td>$O\left(\frac{N}{\log N}^{2/3}\right)$ [1]</td>
<td>$\Omega\left(\log\log N\right)$ [1]</td>
</tr>
<tr>
<td>Greedy</td>
<td>$O\left(\frac{N}{\log N}^{2/3}\right)$ [1]</td>
<td>$&gt; 1.37...$ [1]</td>
</tr>
<tr>
<td>LZ78</td>
<td>$O\left(\frac{N}{\log N}^{2/3}\right)$ [1]</td>
<td>$\Omega\left(\frac{N}{\log N}^{2/3}\right)$ [2]</td>
</tr>
<tr>
<td>AVL-grammar</td>
<td>$O\left(\log\left(\frac{N}{g}\right)\right)$ [3]</td>
<td>-</td>
</tr>
<tr>
<td>$\alpha$-balanced grammar</td>
<td>$O\left(\log\left(\frac{N}{g}\right)\right)$ [2]</td>
<td>-</td>
</tr>
<tr>
<td>Recompression</td>
<td>$O\left(\log\left(\frac{N}{g}\right)\right)$ [4]</td>
<td>-</td>
</tr>
<tr>
<td>ESP-grammar</td>
<td>$O\left(\log^2 N \log^* N\right)$ [5][6]</td>
<td>-</td>
</tr>
</tbody>
</table>

Approximation to Smallest Grammar

Approximation ratios of grammar-based compressors to the smallest grammar of size $g$

<table>
<thead>
<tr>
<th>Compressors</th>
<th>Upper bound</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-Pair</td>
<td>$O((N / \log N)^{2/3})$ [1]</td>
<td>$\Omega(\log N / \log \log N)$ [2]</td>
</tr>
<tr>
<td>LongestMatch</td>
<td>$O((N / \log N)^{2/3})$ [1]</td>
<td>$\Omega(\log \log N)$ [1]</td>
</tr>
<tr>
<td>Greedy</td>
<td>$O((N / \log N)^{2/3})$ [1]</td>
<td>$&gt; 1.37...$ [1]</td>
</tr>
<tr>
<td>LZ78</td>
<td>$O((N / \log N)^{2/3})$ [1]</td>
<td>$\Omega((N / \log N)^{2/3})$ [2]</td>
</tr>
<tr>
<td>AVL-grammar</td>
<td>$O(\log(N / g))$ [3]</td>
<td>-</td>
</tr>
<tr>
<td>$\alpha$-balanced grammar</td>
<td>$O(\log(N / g))$ [2]</td>
<td></td>
</tr>
<tr>
<td>Recompression</td>
<td>$O(\log(N / g))$ [4]</td>
<td></td>
</tr>
<tr>
<td>ESP-grammar</td>
<td>$O(\log^2 N \log^* N)$ [5][6]</td>
<td></td>
</tr>
</tbody>
</table>

We still do not know which one is the best in theory.

Compressed String Processing (CSP)

Techniques that perform various operations on compressed data without decompression.

### Merits of CSP
- Memory saving
- Faster computation

---

**Naive**

1. decompress
2. process
3. output

**CSP**

1. directly process
2. output
The values of $n$ and $h$ vary depending on the grammar compressor that was used for producing SLP.

- We wish to design CSP algorithms that can work on any SLP, independently of the compressor.
For any SLP, \( \log_2 N \leq h \leq n \) always holds. \( \Rightarrow N \in O(2^n) \).

Therefore, CSP algorithms that run in time & space \( O(\text{poly}(n)\text{polylog}(N)) \) are interesting and can be useful.
# (Incomplete) List of Known Results on CSP for SLPs

<table>
<thead>
<tr>
<th></th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[Lifshits 2007]; [Jez 2015]</td>
</tr>
<tr>
<td>Text Indexing</td>
<td>[Claude &amp; Navarro 2012];</td>
</tr>
<tr>
<td></td>
<td>[Maruyama et al. 2013];</td>
</tr>
<tr>
<td></td>
<td>[Tsuruta et al. 2020];</td>
</tr>
<tr>
<td>Subsequence / VLDC Pattern Matching</td>
<td>[Cegielski 2000]; [Tiskin 2009];</td>
</tr>
<tr>
<td></td>
<td>[Yamamoto et al. 2011]</td>
</tr>
<tr>
<td>Random Access / Substring Extraction</td>
<td>[Belazzougui et al. 2013]; [Bille et al. 2015]</td>
</tr>
<tr>
<td>Longest Common Extension</td>
<td>[Karpinski et al. 1997]; [Miyazaki et al. 1997];</td>
</tr>
<tr>
<td></td>
<td>[Bille et al. 2016]; [Nishimoto et al. 2016];</td>
</tr>
<tr>
<td></td>
<td>[I 2017]</td>
</tr>
<tr>
<td></td>
<td>[Gawrychowski 2012]</td>
</tr>
<tr>
<td>Longest Common Substring</td>
<td>[Matsubara et al. 2009]</td>
</tr>
<tr>
<td>String Regularities (palindromes, repetitions etc.)</td>
<td>[Matsubara et al. 2009]; [Inenaga &amp; Bannai 2012];</td>
</tr>
<tr>
<td></td>
<td>[I et al. 2015]</td>
</tr>
</tbody>
</table>
## String Primitives

Space complexities are evaluated by the number of **words** (not bits) unless otherwise stated.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>query time</th>
<th>preprocess. time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>random access</td>
<td>$O(\log N)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>[Bille et al. 2015]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>substring extraction</td>
<td>$O(m + \log N)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>[Bille et al. 2015]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCE queries</td>
<td>$O(\log N)$</td>
<td>$O(n + z \log \frac{N}{z})$</td>
<td>$O(n+z \log \frac{N}{z})$</td>
</tr>
<tr>
<td>[I 2017]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $m$ is the length of the substring to extract.
- $z$ is # of phrases in the LZ77 factorization.
## Text Mining / String Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q$-gram frequencies</td>
<td>$O(qn)$</td>
<td>$O(qn)$</td>
</tr>
<tr>
<td>[Goto et al. 2013]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>most frequent substring</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>[Goto et al. 2013]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>longest repeating substring</td>
<td>$O(n^4 \log n)$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>[Inenaga &amp; Bannai 2012]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>longest common substring</td>
<td>$O(n^4 \log n)$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>[Matsubara et al. 2009]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $q$-gram is a string of length $q$. 
## String Regularities

<table>
<thead>
<tr>
<th>algorithm</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>repetitions (runs) [I et al. 2015]</td>
<td>$O(n^3h)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>palindromes [Matsubara et al. 2009]</td>
<td>$O(nh(n+h \log N))$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>gapped palindromes [I et al. 2015]</td>
<td>$O(nh(n^2 + g \log N))$</td>
<td>$O(n(n+g))$</td>
</tr>
<tr>
<td>periods [I et al. 2015]</td>
<td>$O(n^2h)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>covers [I et al. 2015]</td>
<td>$O(nh(n + \log^2 N))$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

- $g$ is the fixed gap length (usually a constant).
Important Remark

Derivation trees are used only for illustrative purposes, and are not explicitly constructed in CPS algorithms.

CSP on SLPs can be seen as algorithmic technique that performs various kinds of operations on the DAG for SLP, not on the derivation tree.
Problem ($q$-gram frequencies on SLP)

Given an SLP $S$ which represents a string $w$ and a positive integer $q$, compute the number of occurrences of all substrings of length $q$ in $w$. 
Uncompressed $q$-gram Frequencies

Compute # occurrences of each length-$q$ substring in string $w$.

Eg) 3-gram frequencies ($q = 3$)

input $w = \underline{a b a b b b b b b b a b a b}$

output

<table>
<thead>
<tr>
<th>aba</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>abb</td>
<td>1</td>
</tr>
<tr>
<td>bab</td>
<td>3</td>
</tr>
<tr>
<td>bba</td>
<td>1</td>
</tr>
<tr>
<td>bbb</td>
<td>3</td>
</tr>
</tbody>
</table>

Lots of applications for $q$-gram frequencies: NLP, Bioinformatics, Text Mining, etc.
Solution for Uncompressed String

✓ Given an uncompressed string \( w \), we can solve the \( q \)-gram frequencies problem in \( O(N) \) time, by e.g. using the suffix array and LCP array of \( w \).

\[ q = 3 \]

<table>
<thead>
<tr>
<th>SA</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

\$ 

a $ 

aba $ 

ababa $ 

abababa $ 

bababa $ 

babababa $
Solution for Uncompressed String

✓ Given an uncompressed string \( w \), we can solve the \( q \)-gram frequencies problem in \( O(N) \) time, by e.g. using the suffix array and LCP array of \( w \).

\[
q = 3
\]

<table>
<thead>
<tr>
<th></th>
<th>SA</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>ab</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>aba</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>ababa</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>bababa</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ba $</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>baba $</td>
<td>4</td>
</tr>
</tbody>
</table>

$ \text{abababa}$

$ \text{ababa}$

$ \text{aba}$

$ \text{a}$

$ \text{bababa}$

$ \text{aba}$

$ \text{ab}$

$ \text{a}$
Solution for Uncompressed String

✓ Given an uncompressed string $w$, we can solve the $q$-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of $w$.

$q = 3$

<table>
<thead>
<tr>
<th>SA</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>−</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

$q = 3$

$\$$

$a$ $\$$

$aba$ $\$$

$aaba$ $\$$

$ababa$ $\$$

$baba$ $\$$

$baba$ $\$$

$baba$ $\$$

$baba$ $\$$

$baba$ $\$$

$baba$ $\$$
Solution for Uncompressed String

✓ Given an uncompressed string \( w \), we can solve the \( q \)-gram frequencies problem in \( O(N) \) time, by e.g. using the suffix array and LCP array of \( w \).

\[
\begin{array}{c|c|c}
\text{SA} & \text{LCP} & \text{String} \\
8 & - & $ \\
7 & 0 & a$ \\
5 & 1 & <3 \text{aba}$ \\
3 & 3 & \geq 3 \text{ababa}$ \\
1 & 5 & \geq 3 \text{abababa}$ \\
6 & 0 & \text{ba}$ \\
4 & 2 & \text{bababa}$ \\
2 & 4 & \text{babababa}$ \\
\end{array}
\]
Solution for Uncompressed String

✓ Given an uncompressed string $w$, we can solve the $q$-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of $w$.

$q = 3$

<table>
<thead>
<tr>
<th>SA</th>
<th>LCP</th>
<th>Output (SApos, #occ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>-</td>
<td>$$</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>a $</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>&lt;3 aba $</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>≥3 ababa $</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>≥3 abababa $</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>&lt;3 ba $</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>baba $</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>bababa $</td>
</tr>
</tbody>
</table>

(3, 3)
Solution for Uncompressed String

✓ Given an uncompressed string \( w \), we can solve the \( q \)-gram frequencies problem in \( O(N) \) time, by e.g. using the suffix array and LCP array of \( w \).

\[
\begin{array}{c|c}
SA & LCP \\
\hline
8 & - \\
7 & 0 \\
5 & 1 \\
3 & 3 \\
1 & 5 \\
6 & 0 \\
4 & 2 \\
2 & 4 \\
\end{array}
\]

Output \((SApos, \#occ)\) : \((3, 3)\)
**Solution for Uncompressed String**

✓ Given an uncompressed string \( w \), we can solve the \( q \)-gram frequencies problem in \( O(N) \) time, by e.g. using the suffix array and LCP array of \( w \).

\[
\begin{array}{c|c|c}
\text{SA} & \text{LCP} & \text{Output} (\text{SApos} \text{, } \#\text{occ}) \\
8 & - & $ \\
7 & 0 & a$ \\
5 & 1 & abab$ \\
3 & 3 & abababa$ \\
1 & 5 & baba$ \\
6 & 0 & <3 \\
4 & 2 & baba$ \\
2 & 4 & \geq 3 \\
\end{array}
\]

\( q = 3 \)
Compressed $q$-gram Frequencies

Compute the number of occurrences of each length-$q$ substring in the grammar-compressed string $w$.

**Input**
- **CFG** deriving only $w$
  
  $$\begin{align*}
  S &\rightarrow BC \text{ } CB \\
  C &\rightarrow bb \\
  B &\rightarrow AA \\
  A &\rightarrow ab
  \end{align*}$$

**Output**
- 3-gram frequencies

$$\begin{array}{|c|c|}
\hline
 \text{aba} & 2 \\
 \text{abb} & 1 \\
 \text{bab} & 3 \\
 \text{bba} & 1 \\
 \text{bbb} & 3 \\
\hline
\end{array}$$
Stabbing

An integer interval \([b, e]\) \((1 \leq b \leq e \leq N)\) is said to be stabbed by a variable \(X_i\), if the LCA of the \(b\)th and \(e\)th leaves of the derivation tree \(T\) is labeled by \(X_i\).
Observation

✓ Assume that the occurrence of a \( q \)-gram \( p \) starting at position \( j \) is stabbed by an occurrence of \( X_i \) in \( T \).

✓ Then clearly, in any other occurrence of \( X_i \) in \( T \), there is another stabbed occurrence of \( p \).
Hence, the $q$-gram frequencies problem on SLP reduces to the following sub-problems:

**Sub-Problem 1**

For each variable $X_i$, count the number of occurrences of $X_i$ in the derivation tree $T$.

**Sub-Problem 2**

For each variable $X_i$, count the number of occurrences of every $q$-gram stabbed by $X_i$. 
Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable $X_i$ in the derivation tree $T$ in $O(n)$ time.
Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable $X_i$ in the derivation tree $T$ in $O(n)$ time.

✓ The root occurs exactly once.

![Derivation Tree Diagram]

```
1 7 6
5
3 4
1 2

a b
```
We can count # of occurrences of every variable $X_i$ in the derivation tree $T$ in $O(n)$ time.

For each node in a topological order, propagate its number of occurrences to its children.
Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable $X_i$ in the derivation tree $T$ in $O(n)$ time.

✓ For each node in a topological order, propagate its number of occurrences to its children.
Solving Sub-Problem 1

We can count # of occurrences of every variable $X_i$ in the derivation tree $T$ in $O(n)$ time.

For each node in a topological order, propagate its number of occurrences to its children.
Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable $X_i$ in the derivation tree $T$ in $O(n)$ time.

✓ For each node in a topological order, propagate its number of occurrences to its children.
Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable $X_i$ in the derivation tree $T$ in $O(n)$ time.

✓ For each node in a topological order, propagate its number of occurrences to its children.
Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable $X_i$ in the derivation tree $T$ in $O(n)$ time.
Solving Sub-Problem 2

For each variable $X_i$, count the number of occurrences of every $q$-gram stabbed by $X_i$. 
Solving Sub-Problem 2

**Key Observation:** Each variable $X_i$ can stab at most $q-1$ occurrences of $q$-grams.
Solving Sub-Problem 2

- We “partially” decompress the substring $t_i = X_l[|X_l|-q+2..|X_l|] X_r[1..q-1]$ of length $2q-2$. 

![Diagram showing the decomposition of strings $X_l$, $X_i$, and $X_r$.]
Solving Sub-Problem 2

✓ We “partially” decompress the substring 
\[ t_i = X_i[|X_l| - q + 2 \ldots |X_l|] X_r[1 \ldots q - 1] \] of length \( 2q - 2 \).

All \( q \)-grams stabbed by \( X_i \) occur inside \( t_i \).
Solving Sub-Problem 2

Lemma

For all variables $X_i$, we can count the number of occurrences of every $q$-gram stabbed by $X_i$ in $O(qn)$ time and space.

✓ For every variable $X_i$, the substring $t_i$ can be computed in a total of $O(qn)$ time, by a simple DP (to be explained in the next slide).
To compute $t_i$, it is enough to compute the prefix and suffix of length $q-1$ of each variable.
Solving Sub-Problem 2

For all variables $X_i$, we can count the number of occurrences of every $q$-gram stabbed by $X_i$ in $O(qn)$ time and space.

Then, we construct the suffix array and LCP array for strings $t_1, ..., t_n$ in $O(|t_1 ... t_n|) \subseteq O(qn)$ time.
The problem of computing $q$-gram frequencies on SLP can be solved in $O(qn)$ time and space.

- Usually $q$ is a small constant (from 2 to 4)
  - In most practical situations, this algorithm works in $O(n)$ time & space.

- Decompression-then-compute method takes $O(N) \subseteq O(2^n)$ time in the worst case.
## Experimental Results

Running time (sec.) on XML data (200MB)

<table>
<thead>
<tr>
<th>$q$</th>
<th>(1) Naive $O(qN)$ time</th>
<th>(2) SA $O(N)$ time</th>
<th>(3) Goto et al. $O(qn)$ time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>22.9</td>
<td>41.7</td>
<td>6.5</td>
</tr>
<tr>
<td>3</td>
<td>55.7</td>
<td>41.7</td>
<td>11.0</td>
</tr>
<tr>
<td>4</td>
<td>93.3</td>
<td>41.7</td>
<td>16.3</td>
</tr>
<tr>
<td>5</td>
<td>129.3</td>
<td>41.7</td>
<td>21.3</td>
</tr>
<tr>
<td>6</td>
<td>158.7</td>
<td>41.7</td>
<td>25.8</td>
</tr>
<tr>
<td>7</td>
<td>181.1</td>
<td>41.7</td>
<td>30.1</td>
</tr>
<tr>
<td>8</td>
<td>198.3</td>
<td>41.9</td>
<td>34.2</td>
</tr>
</tbody>
</table>

**Note:** CSP algorithm by Goto et al. is the fastest even if we subtract decompression times (3.6 sec.) from (1) and (2), for all values of $q$ tested here.

Goto et al.'s method is by far the fastest for important values of $q = 2..4$. 
Finding Repetitions from SLP

Problem (finding repetitions from SLP)

Given an SLP $S$ which represents a string $w$, compute all squares and runs that occur in $w$.

squares
(of form $xx$)

Note: There are more squares

runs
(maximal repetition $x^k x'$)
Stabbed Runs

✓ For each run in the string $w$, there is a unique variable $X_i$ that stabs the run.
In any other occurrences of $X_i$ in the derivation tree, the same run is stabbed by $X_i$. 
Computing runs in string $w$ reduces to computing the stabbed runs for each variable $X_i$. 

![Diagram showing the relationship between $T$, $w$, and $X_i$.]
For each variable $X_i$, we first compute (the beginning and ending positions of) the stabbed squares.
We then determine how long the periodicity continues to the right and to the left, using LCE.

- We can efficiently perform LCE without expanding $X_i$. 

![Diagram of Stabbed Runs](image-url)
We then determine how long the periodicity continues to the right and to the left, using LCE.

- We can efficiently perform LCE without expanding $X_i$. 
Finding Repetitions on SLP

**Theorem [I et al. 2015]**

$O(n \log N)$-size representation of all runs and squares can be computed in $O(n^3h)$ time with $O(n^2)$ working space.

- There are at most $N-1$ runs [Bannai et al. 2017].
  - Naive representation of runs requires $O(N) \subseteq O(2^n)$ space in the worst case.

- We can compactly represent all runs within $O(n \log N)$ space using periodicities.
Problem 5 (finding palindromes on SLP)

Given an SLP $S$ which represents a string $w$, compute all maximal palindromes of $w$.

maximal palindromes

$abbbbaabbbbbbabbbabab$
Finding Palindromes from SLP

Problem 5 (finding palindromes on SLP)

Given an SLP $S$ which represents a string $w$, compute all maximal palindromes of $w$.

maximal palindromes $\quad a b b b a a b b b b a b b b a a b$
Problem 5 (finding palindromes on SLP)

Given an SLP $S$ which represents a string $w$, compute all maximal palindromes of $w$. 
Problem 5 (finding palindromes on SLP)

Given an SLP $S$ which represents a string $w$, compute all maximal palindromes of $w$.

Maximal palindromes:

- $abbbbaabbbbababbbbaab$,

There are $N$ integer positions and $N-1$ half-integer positions.

$\Rightarrow$ There are $2N-1$ maximal palindromes in a string of length $N$. 

---

```
Finding Palindromes from SLP
```
Stabbed Palindromes

✓ For each variable $X_i$, there can be 3 different types of stabbed maximal palindromes.
Type 1 maximal palindromes of $X_i$ can be computed by extending the arms of the suffix palindromes of $X_l$. But there can be too many ($O(N)$) suffix palindromes...
Lemma [Apostolico et al. 1995]

For any string of length $N$, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

- We can extend the suffix palindromes belonging to the same arithmetic progression in a batch.
- This batched LCE can be performed efficiently using the periodicity of suffix palindromes.
Batched LCE for Suffix Palindromes

For any string of length $N$, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

Lemma [Apostolico et al. 1995]
Batched LCE for Suffix Palindromes

Lemma [Apostolico et al. 1995]

For any string of length $N$, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.
Batched LCE for Suffix Palindromes

For any string of length $N$, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

Lemma [Apostolico et al. 1995]
Batched LCE for Suffix Palindromes

For any string of length $N$, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

Lemma [Apostolico et al. 1995]

For any string of length $N$, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.
Batched LCE for Suffix Palindromes

$G$ (arithmetic progression)
Batched LCE for Suffix Palindromes

$G$ (arithmetic progression)
Batched LCE for Suffix Palindromes

For each single arithmetic progression $G$, three LCE queries are sufficient.

$G$ (arithmetic progression)
Finding Palindromes on SLP

Theorem [Matsubara et al. 2009]

$O(n \log N)$-size representation of all maximal palindromes can be computed in $O(n h (n + h \log N))$ time using $O(n^2)$ space.
Finding Gapped Palindromes on SLP

Problem (finding gapped palindromes on SLP)

Given an SLP $S$ which represents a string $w$ and a positive integer $g$, compute all $g$-gapped palindromes that occur in $w$.

3-gapped palindromes ($g = 3$)

```
abababcbabbaaabbabca
```
Stabbed $g$-gapped Palindromes

✓ There are 3 types of $g$-gapped palindromes which are stabbed by each variable $X_i$. 

- Type 1
- Type 2
- Type 3
Finding Gapped Palindromes on SLP

Theorem [I et al. 2015]

$O(n (\log N + g))$-size representation of all $g$-gapped palindromes can be computed in $O(nh (n^2 + g \log N))$ time using $O(n^2)$ space.

- Unfortunately, Apostolico et al.’s lemma does not hold for gapped palindromes.
- Instead, we can use a similar technique to the solution for computing stabbed squares.
More on LCE, height $h$, and Stabbing

We have seen that many of the CSP algorithms on SLPs use

- **LCE (Longest Common Extension) queries**, their efficiency depends on
- **the height $h$ of the derivation tree**, and their key concepts are
- **stabbed occurrences at variable boundaries**.

In the next slides, we will briefly review the recent developments on these three concepts.
LCE (longest common extension) queries

- Recall that extension of periodicities (for runs) and extension of arms (for palindromes) can be efficiently computed by an LCE query on SLP.

- LCE($i, j$) for string $w$ returns the length of the longest common prefix of $w[i..N]$ and $w[j..N]$.

$$w = \text{abbabcbabaabbabca}$$

$LCE(3, 13) = 4$
Theorem [I 2017]

A data structure of size $O(n + z \log (N / z))$ which answers LCE queries on SLP in $O(\log N)$ time can be constructed in $O(n \log (N / n))$ time.

$z = \text{size of LZ77 factorization}$

I’s algorithm above uses a kind of locally consistent parsing called *Recompression* [Jez 2015] that transforms a given SLP into another small SLP of size $O(z \log (N / z))$. 
In the grammar produced by Recompression, the occurrences of the same substring are compressed “almost” in the same way.

In each occurrence of a substring \( y \), there is a unique sequence of symbols called “common sequences”.

- Common sequences for \( y \)
LCE with Recompression

LCE\((i, j)\) can be computed by matching the common sequences of the LCE sub-strings that begin at positions \(i\) and \(j\).

# of traversed nodes is bounded by \(O(\log N)\).
Balancing SLPs

- An SLP is said to be balanced if its height \( h = O(\log N) \).

- Given an SLP of size \( n \), all existing approximation algorithms to the smallest grammar
  - AVL grammar [Rytter 2003];
  - \( \alpha \)-balanced grammar [Charikar et al. 2005];
  - Recompression [Jez 2015];

are able to produce a balanced SLP. But their grammar sizes blow up to \( O(g \log(N/g)) \), which can be larger than \( n \) when \( n \) is quite small.
Recently, Ganardi et al. (FOCS 2019) showed how to transform a given SLP of size $n$ into a balanced SLP of size $O(n)$.

New $O(\log N)$-time random access and $O(m + \log N)$-time substring extraction algorithms with $O(n)$ space, which alternate Bille et al.’s previous algorithms.
We precompute the decompression length of every variable in a bottom-up manner.
We precompute the **decompression length** of every variable in a bottom-up manner.
$O(\log N)$-time Random Access

Balanced SLP

Derivation Tree

We precompute the **decompression length** of every variable in a bottom-up manner.
We precompute the **decompression length** of every variable in a bottom-up manner.
We precompute the **decompression length** of every variable in a bottom-up manner.
For a given position $i$ in $w$, we can random access to the $i$-th char. in a top-down manner.
\(O(\log N)\)-time Random Access

Balanced SLP

\[
\begin{array}{ccc}
1 & 2 & 3 \\
a & b & a \\
1 & 2 & 1 \\
\end{array}
\]

```
1 1 1 1 2
```

Derivation Tree

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
a & a & a & b & a & b & a \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

For a given position \(i\) in \(w\), we can random access to the \(i\)-th char. in a top-down manner.
For a given position $i$ in $w$, we can random access to the $i$-th char. in a top-down manner.
Balanced SLP Derivation Tree

For a given position $i$ in $w$, we can random access to the $i$-th char. in a top-down manner.
For a given position $i$ in $w$, we can random access to the $i$-th char. in a top-down manner.
**O**(log *N*)-time Random Access

Balanced SLP

Derivation Tree

For a given position *i* in *w*, we can random access to the *i*-th char. in a top-down manner.
$O(\log N)$-time Random Access

Balanced SLP

Derivation Tree

For a given position $i$ in $w$, we can random access to the $i$-th char. in a top-down manner.
For a given position $i$ in $w$, we can random access to the $i$-th char. in a top-down manner.
For a given position $i$ in $w$, we can random access to the $i$-th char. in a top-down manner.
Recently, Ganardi et al. (FOCS 2019) showed how to transform a given SLP of size $n$ into a balanced SLP of size $O(n)$.

- New $O(\log N)$-time random access and $O(m + \log N)$-time substring extraction algorithms with $O(n)$ space, which alternate Bille et al.’s previous algorithms.

- In addition, every $h$ term in the time complexity for other operations on SLPs can be replaced with $\log N$. 

**Balancing SLPs [cont.]**
A set $\Gamma \subseteq \{1, \ldots, N\}$ of positions in a string $w$ of length $N$ is called a string attractor of $w$, if any substring $y$ of $w$ has an occurrence $y = w[i..j]$ that contains an element $k$ of $\Gamma$ (i.e. $k \in [i..j]$).
String Attractors

\{1, 4, 7, 11, 12\}

\[ w = \begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array} \]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCD, ABCCDA, BCCDAB, CCDABC, CDAABC, CDABCC, DABCDA, ABCCCD, DABCCD, ABCCDA, BCCDAB, ABCCDAB, BCCDABC, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABCC, DABCCDABCCA

CABCCDABCCA
String Attractors

\{1, 4, 7, 11, 12\}

\[ w = \begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array} \]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, BCCDBC, CCDABC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CDABCCDABC, DABCCDABCC, CDABCCDACCA, CDABCCDABCCA
String Attractors

\{1, 4, 7, 11, 12\}

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
\]

\[w = \text{C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABBCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDA, BCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA}\]
String Attractors

\{1, 4, 7, 11, 12\}

\[ w = \begin{array}{cccccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array} \]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, BCCDABCCA, CDABCCDAC, ABCCDACC
String Attractors

\{1, 4, 7, 11, 12\}

\[w = \]

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
\]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDBCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCCA

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDBCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCCA
String Attractors

\{1, 4, 7, 11, 12\}

\[w = \begin{array}{cccccccccccc}
    1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}\]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABCC, DABCC, ABCCCD, BCCDAB, CCDABC, DABCCA, ABCCDA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCD, DABCCDA, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA
String Attractors

\{1, 4, 7, 11, 12\}

\[ w = \begin{array}{ccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array} \]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, ABCCDAB, BCCDABC, CCDABCC, DABCCDAB, ABCCDABC, BCCDABCC, CDABCCDAB, DABCCDABC, ABCCDABCC, CDABCCDABCC, DABCCDABCCA
String Attractors

\{1, 4, 7, 11, 12\}

\[ w = \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array} \]

String Attractors

\{1, 4, 7, 11, 12\}

\[ w = \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array} \]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, DABCCDA, ABCCCD, DABCCCD, ABCCDCC, DABCCDCC, ABCCDACC, DABCCDACC, ABCCDACC, DABCCDACC, ABCCDACC, DABCCDACC
String Attractors

\{1, 4, 7, 11, 12\}

\[ w = \begin{array}{ccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array} \]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCD, ABCCDA, BCCDAB, CCDBAC, DABCBA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCCA
String Attractors

\{1, 4, 7, 11, 12\}

\[ w = \begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\end{array} \]

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CDABCC, DAABCC, BCCDABC, CCDABCC, DABCCDA, DABCCBCC, ABCCDABCC, CDABCCDABC, DABCCDABCC
String Attractors

A set $\Gamma \subseteq \{1, \ldots, N\}$ of positions in a string $w$ of length $N$ is called a string attractor of $w$, if any substring $y$ of $w$ has an occurrence $y = w[i..j]$ that contains an element $k$ of $\Gamma$ (i.e. $k \in [i..j]$).

- Every string $w$ has a string attractor since $\{1, \ldots, |w|\}$ is clearly a string attractor of $w$.
- String Attractors generalize to the notion of “stabbing” by SLP boundaries.
SLP as String Attractor

DAG for SLP $S$

Derivation tree $T$ of SLP $S$
String Attractor as Lower Bound for SLP size

Theorem

Let $\gamma = \text{the smallest string attractor size}$, $z = \# \text{ phrases in the LZ77 factorization}$, $g = \text{the smallest SLP size}$, for the same string $w$. Then, $\gamma \leq z \leq g$ holds.

- $\gamma \leq g$ because any substring of $w$ must be stabbed by at least one variable of the SLP.
- $\gamma \leq z$ follows from similar arguments.
- $z \leq g$ was proved in the literature [Rytter 2003].
Indexing with String Attractor Space

Theorem [Ettienne et al. 2020 (arXiv)]

There exist compressed indexing structures that perform pattern matching queries in $O(m + (occ + 1)\log^\varepsilon N)$ time with $O(\gamma \log(N/\gamma))$ space, or in $O(m + occ)$ time with $O(\gamma \log(N/\gamma) \log^\varepsilon N)$ space.

$\gamma =$ smallest attractor size; $m =$ pattern length; $occ =$ # pattern occurrences; $\varepsilon > 0$

✓ Computing the smallest string attractor is NP-hard [Kempa & Prezza 2017].

✓ However, these indexing structures can be built without knowing the smallest attractor size $\gamma$. 
Conclusions and Future Work

- Grammar-based compression is a powerful compression scheme for highly repetitive strings.
- A variety of string processing can be performed directly on grammar-compressed strings.
  - How can we close the gap for the upper and lower bounds of Re-Pair’s approximation ratio to the smallest grammar?
  - Can we perform various processing within space proportional to the smallest string attractor size $\gamma$?