Grammar-based tree compression: combinatorics and algorithms

Markus Lohrey

Universität Siegen

June 20, 2024

What is grammar-based compression?

What is grammar-based compression?

Grammar-based compression origins in string (text) compression.

What is grammar-based compression?

Grammar-based compression origins in string (text) compression.

Idea: Compress a string s by a context-free grammar that only produces s (Storer, Szymansk 1982)

What is grammar-based compression?

Grammar-based compression origins in string (text) compression.

Idea: Compress a string s by a context-free grammar that only produces s (Storer, Szymansk 1982)

Definition (straight-line program – SLP)

An SLP is a context-free grammar $\mathcal{G}$ in Chomsky normal form that derives a unique word that is denoted by $\mathsf{val}(\mathcal{G})$.

What is grammar-based compression?

Grammar-based compression origins in string (text) compression.

Idea: Compress a string s by a context-free grammar that only produces s (Storer, Szymansk 1982)

Definition (straight-line program – SLP)

An SLP is a context-free grammar $\mathcal{G}$ in Chomsky normal form that derives a unique word that is denoted by $\mathrm{val}(\mathcal{G})$.

- For every variable A there is a unique production of the form $A \to BC$ or $A \to a$, and
- there are no cycles in derivations.

What is grammar-based compression?

Grammar-based compression origins in string (text) compression.

Idea: Compress a string s by a context-free grammar that only produces s (Storer, Szymansk 1982)

Definition (straight-line program – SLP)

An SLP is a context-free grammar $\mathcal{G}$ in Chomsky normal form that derives a unique word that is denoted by $\mathsf{val}(\mathcal{G})$.

- For every variable A there is a unique production of the form $A \to BC$ or $A \to a$, and
- there are no cycles in derivations.

The size of $\mathcal{G}$ is the number of variables ($=$ number of productions).

Example of an SLP

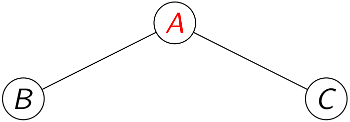
$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$

Example of an SLP

$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$

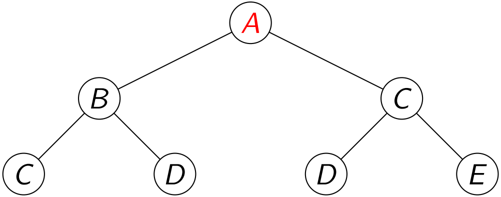
Example of an SLP

$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$



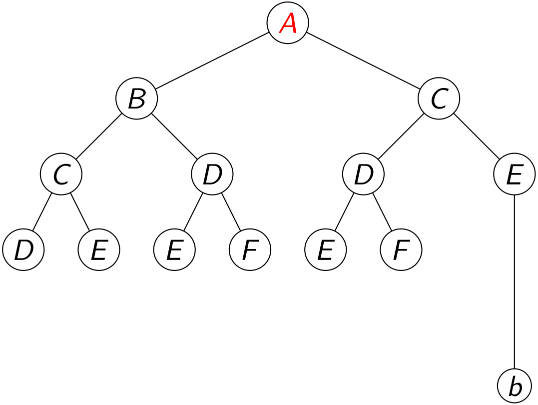
Example of an SLP

$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$



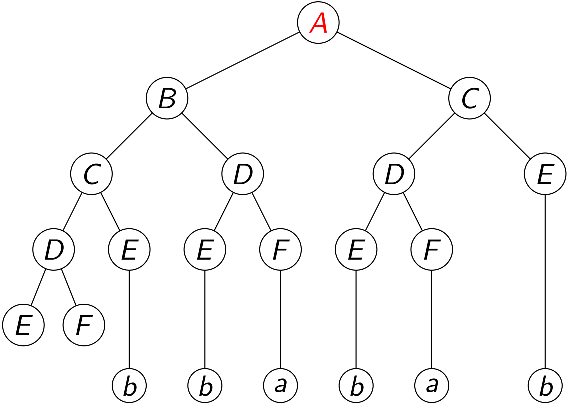
Example of an SLP

$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$



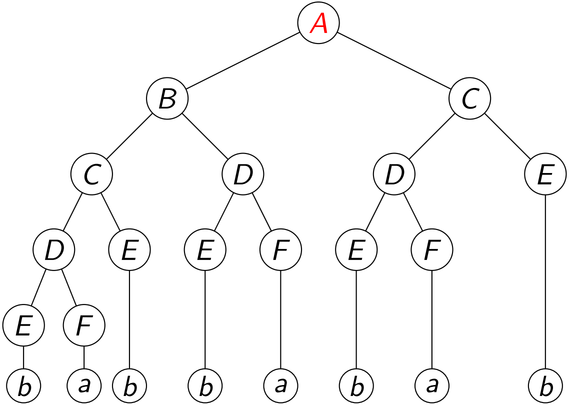
Example of an SLP

$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$



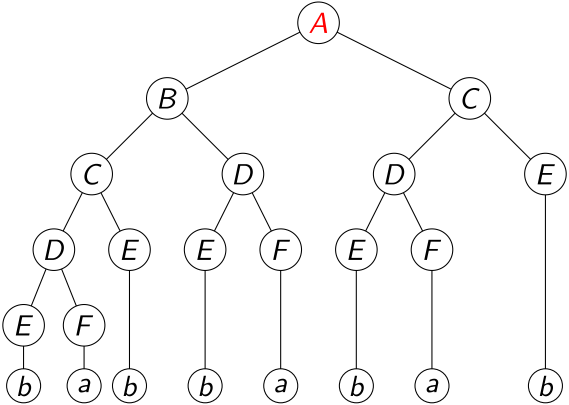
Example of an SLP

$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$



Example of an SLP

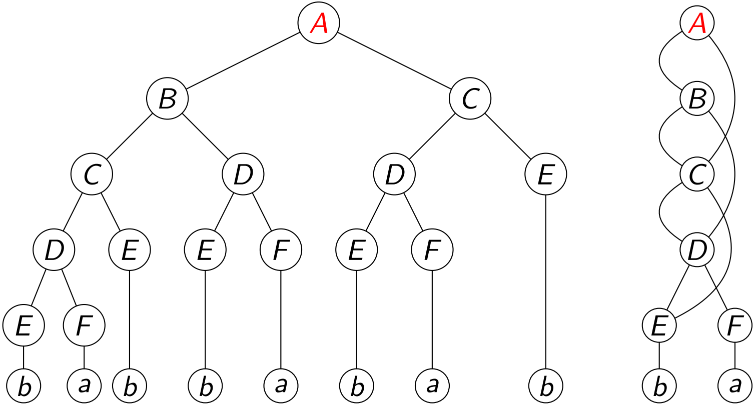
$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$



$\text{val}(\mathcal{G}) = babbabab \qquad |\mathcal{G}| = 6$

Example of an SLP

$A \to BC,\ B \to CD,\ C \to DE,\ D \to EF,\ E \to b,\ F \to a$



$\text{val}(\mathcal{G}) = babbabab \qquad |\mathcal{G}| = 6$

Grammar-based string compression

An SLP $\mathcal{G}$ can be seen as a compressed representation of $\mathsf{val}(\mathcal{G})$.

Grammar-based string compression

An SLP $\mathcal{G}$ can be seen as a compressed representation of $\mathsf{val}(\mathcal{G})$.

Grammar-based compressor = algorithm that computes from a given word w a hopefully small SLP $\mathcal{G}$ with $\mathsf{val}(\mathcal{G}) = w$.

Grammar-based string compression

An SLP $\mathcal{G}$ can be seen as a compressed representation of $\mathsf{val}(\mathcal{G})$.

Grammar-based compressor $=$ algorithm that computes from a given word w a hopefully small SLP $\mathcal{G}$ with $\mathsf{val}(\mathcal{G}) = w$.

Examples: LZ78, BiSection, RePair, Sequitur, ...

Grammar-based string compression

An SLP $\mathcal{G}$ can be seen as a compressed representation of $\mathsf{val}(\mathcal{G})$.

Grammar-based compressor = algorithm that computes from a given word w a hopefully small SLP $\mathcal{G}$ with $\mathsf{val}(\mathcal{G}) = w$.

Examples: LZ78, BiSection, RePair, Sequitur, ...

Let $w \in \Sigma^*$ be a word of length n and $\sigma = |\Sigma|$.

Grammar-based string compression

An SLP $\mathcal{G}$ can be seen as a compressed representation of $\text{val}(\mathcal{G})$.

Grammar-based compressor = algorithm that computes from a given word w a hopefully small SLP $\mathcal{G}$ with $\text{val}(\mathcal{G}) = w$.

Examples: LZ78, BiSection, RePair, Sequitur, ...

Let $w \in \Sigma^*$ be a word of length n and $\sigma = |\Sigma|$.

Let $\text{opt}(w)$ be the size of a smallest SLP for w.

Grammar-based string compression

An SLP $\mathcal{G}$ can be seen as a compressed representation of $\text{val}(\mathcal{G})$.

Grammar-based compressor $=$ algorithm that computes from a given word w a hopefully small SLP $\mathcal{G}$ with $\text{val}(\mathcal{G}) = w$.

Examples: LZ78, BiSection, RePair, Sequitur, ...

Let $w \in \Sigma^*$ be a word of length n and $\sigma = |\Sigma|$.

Let $\text{opt}(w)$ be the size of a smallest SLP for w.

Lower bound: $\text{opt}(w) \geq \log_2 n$

Grammar-based string compression

An SLP $\mathcal{G}$ can be seen as a compressed representation of $\text{val}(\mathcal{G})$.

Grammar-based compressor = algorithm that computes from a given word w a hopefully small SLP $\mathcal{G}$ with $\text{val}(\mathcal{G}) = w$.

Examples: LZ78, BiSection, RePair, Sequitur, ...

Let $w \in \Sigma^*$ be a word of length n and $\sigma = |\Sigma|$.

Let $\text{opt}(w)$ be the size of a smallest SLP for w.

Lower bound: $\text{opt}(w) \geq \log_2 n$

Berstel, Brlek 1987

$\text{opt}(w) \leq \mathcal{O}\left(\frac{n}{\log_\sigma n}\right)$ (assuming $\sigma \geq 2$).

Computing small SLPs

The smallest grammar problem

INPUT: A word w

OUTPUT: An SLP $\mathcal{G}$ for w of size $\mathrm{opt}(w)$

Computing small SLPs

The smallest grammar problem

INPUT: A word w

OUTPUT: An SLP $\mathcal{G}$ for w of size $\mathrm{opt}(w)$

Charikar et al. 2002

The smallest grammar problem cannot be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$.

Computing small SLPs

The smallest grammar problem

INPUT: A word w

OUTPUT: An SLP $\mathcal{G}$ for w of size $\text{opt}(w)$

Charikar et al. 2002

The smallest grammar problem cannot be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$.

Even worse: Unless $\mathsf{P} = \mathsf{NP}$, there is no polynomial time algorithm that produces for every word w an SLP of size $8569/8568 \cdot \text{opt}(w)$.

Computing small SLPs

The smallest grammar problem

INPUT: A word w

OUTPUT: An SLP $\mathcal{G}$ for w of size $\text{opt}(w)$

Charikar et al. 2002

The smallest grammar problem cannot be solved in polynomial time unless $\text{P} = \text{NP}$.

Even worse: Unless $\text{P} = \text{NP}$, there is no polynomial time algorithm that produces for every word w an SLP of size $8569/8568 \cdot \text{opt}(w)$.

Charikar et al. 2002, Rytter 2004, Jez 2013

There is a linear time algorithm that produces for every word w of length n an SLP of size at most $\mathcal{O}(\log(n) \cdot \text{opt}(w))$.

Balancing straight-line program

Ganardi, Jeż, L 2021

From a given SLP $\mathcal{G}$ of size n such that $w := \mathrm{val}(\mathcal{G})$ has length N, one can compute in time $\mathcal{O}(n)$ an SLP $\mathcal{H}$ such that:

- $\mathrm{val}(\mathcal{H}) = \mathrm{val}(\mathcal{G})$
- $|\mathcal{H}| \in \mathcal{O}(n)$
- $\mathrm{depth}(\mathcal{H}) \in \mathcal{O}(\log N)$

Balancing straight-line program

Ganardi, Jeż, L 2021

From a given SLP $\mathcal{G}$ of size n such that $w := \text{val}(\mathcal{G})$ has length N, one can compute in time $\mathcal{O}(n)$ an SLP $\mathcal{H}$ such that:

- $\text{val}(\mathcal{H}) = \text{val}(\mathcal{G})$
- $|\mathcal{H}| \in \mathcal{O}(n)$
- $\text{depth}(\mathcal{H}) \in \mathcal{O}(\log N)$

Corollary: random access in logarithmic time on compressed words.

Balancing straight-line program

Ganardi, Jeż, L 2021

From a given SLP $\mathcal{G}$ of size n such that $w := \mathsf{val}(\mathcal{G})$ has length N, one can compute in time $\mathcal{O}(n)$ an SLP $\mathcal{H}$ such that:

- $\mathsf{val}(\mathcal{H}) = \mathsf{val}(\mathcal{G})$
- $|\mathcal{H}| \in \mathcal{O}(n)$
- $\mathsf{depth}(\mathcal{H}) \in \mathcal{O}(\log N)$

Corollary: random access in logarithmic time on compressed words.

From a given SLP $\mathcal{G}$ one can built in linear time a data structure that allows to solve for $w = \mathsf{val}(\mathcal{G})$ the following problem in time $\mathcal{O}(\log |w|)$:

- Input: a position $i \in [1, |w|]$
- Output: the i-th symbol of w.

Balancing straight-line program

Ganardi, Jeż, L 2021

From a given SLP $\mathcal{G}$ of size n such that $w := \mathrm{val}(\mathcal{G})$ has length N, one can compute in time $\mathcal{O}(n)$ an SLP $\mathcal{H}$ such that:

- $\mathrm{val}(\mathcal{H}) = \mathrm{val}(\mathcal{G})$
- $|\mathcal{H}| \in \mathcal{O}(n)$
- $\mathrm{depth}(\mathcal{H}) \in \mathcal{O}(\log N)$

Corollary: random access in logarithmic time on compressed words.

From a given SLP $\mathcal{G}$ one can built in linear time a data structure that allows to solve for $w = \mathrm{val}(\mathcal{G})$ the following problem in time $\mathcal{O}(\log |w|)$:

- Input: a position $i \in [1, |w|]$
- Output: the i-th symbol of w.

Has been shown by Bille et al. using several complicated data structures.

Tree compression I: directed acyclic graphs

Fix an alphabet Γ of symbols.

Tree compression I: directed acyclic graphs

Fix an alphabet Γ of symbols.

We consider rooted trees, where nodes are labelled with symbols from Γ, and every node has arbitrarily many children that are ordered.

Tree compression I: directed acyclic graphs

Fix an alphabet Γ of symbols.

We consider rooted trees, where nodes are labelled with symbols from Γ, and every node has arbitrarily many children that are ordered.

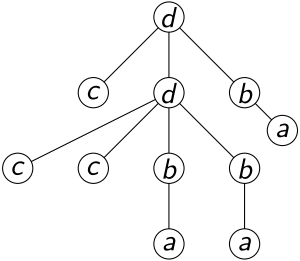
Directed acyclic graphs (DAGs) are the standard way to compress trees.

Tree compression I: directed acyclic graphs

Fix an alphabet Γ of symbols.

We consider rooted trees, where nodes are labelled with symbols from Γ, and every node has arbitrarily many children that are ordered.

Directed acyclic graphs (DAGs) are the standard way to compress trees.

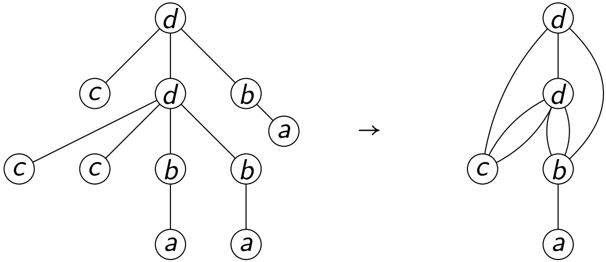


Tree compression I: directed acyclic graphs

Fix an alphabet Γ of symbols.

We consider rooted trees, where nodes are labelled with symbols from Γ, and every node has arbitrarily many children that are ordered.

Directed acyclic graphs (DAGs) are the standard way to compress trees.



DAGs and tree grammars

A DAG can be seen as a regular tree grammar:

DAGs and tree grammars

A DAG can be seen as a regular tree grammar:

- The nodes of the DAG are nonterminals of the grammar

DAGs and tree grammars

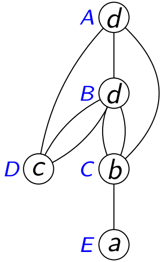
A DAG can be seen as a regular tree grammar:

- The nodes of the DAG are nonterminals of the grammar
- Productions are of the form $A \to a(A_1, A_2, \ldots, A_k)$.

DAGs and tree grammars

A DAG can be seen as a regular tree grammar:

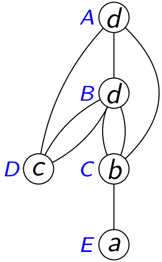
- The nodes of the DAG are nonterminals of the grammar
- Productions are of the form $A \to a(A_1, A_2, \ldots, A_k)$.



DAGs and tree grammars

A DAG can be seen as a regular tree grammar:

- The nodes of the DAG are nonterminals of the grammar
- Productions are of the form $A \to a(A_1, A_2, \ldots, A_k)$.



$$
\begin{aligned}
A &\to d(D, B, C) \\
B &\to d(D, D, C, C) \\
C &\to b(E) \\
D &\to c \\
E &\to a
\end{aligned}
$$

Minimal DAGs

Clearly, every tree has a unique minimal DAG: merge nodes in which isomorphic subtrees are rooted as long as possible.

Downey, Sethi, Tarjan 1980

For a given tree, its minimal DAG can be computed in linear time.

DAGs and asymptotic combinatorics

Bousquet-Mélou, L, Maneth, Noeth 2015

The average number of nodes of the minimal DAG for a uniformly chosen tree of size n with $k = |\Gamma|$ node labels is

$$\sqrt{\frac{\ln(4k)}{\pi}} \cdot \frac{n}{\sqrt{\ln n}} \cdot \big(1 + o(1)\big).$$

- Extends a result of Flajolet, Sipala and Steyaert for binary unlabelled trees.
- Similar results that apply to certain classes of random tree models were recently shown by Seelbach-Benkner and Wagner.

Tree compression II: forest straight-line programs

Let's consider forests = ordered sequences of trees.

Tree compression II: forest straight-line programs

Let's consider forests = ordered sequences of trees.

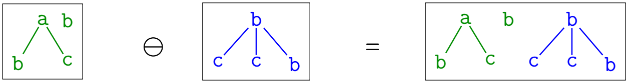
There are two operations for constructing forests:

Tree compression II: forest straight-line programs

Let's consider forests = ordered sequences of trees.

There are two operations for constructing forests:

Horizontal concatenation:

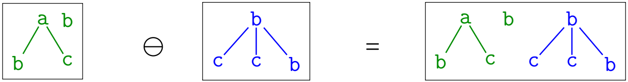


Tree compression II: forest straight-line programs

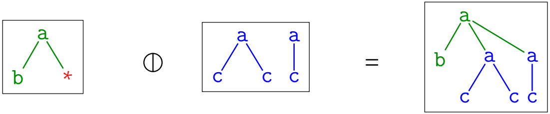
Let's consider forests = ordered sequences of trees.

There are two operations for constructing forests:

Horizontal concatenation:



Vertical concatenation:

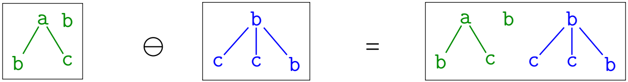


Tree compression II: forest straight-line programs

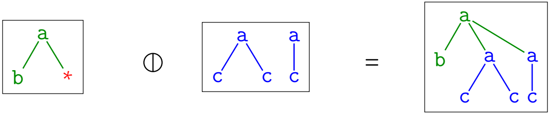
Let's consider forests = ordered sequences of trees.

There are two operations for constructing forests:

Horizontal concatenation:



Vertical concatenation:



A (forest) context is a forest, where exactly one leaf is labelled with the special symbol $* \notin \Gamma$.

Forest algebra expressions

Forests and forest contexts can be also written as parenthesized expressions:

- A forest : $a(b\,c)\ b(b(c\,c\,a)\,a)\ a$
- A forest context: $a(b\,c)\ b(b(c\,c\,*)\,a)\ a$

Then we have

- $F \ominus G = F\,G$
- $F \varobar G = F[* \to G]$

A forest algebra expression is an expression that is built from the constants

- a and $a_* := a(*)$ for $a \in \Gamma$ and
- the binary operations $\ominus$ and $\varobar$.

Expressions must be well-typed (F = type of forests, C = type of contexts):

Forest algebra expressions

Forests and forest contexts can be also written as parenthesized expressions:

- A forest : $a(b\,c)\ b(b(c\,c\,a)\,a)\ a$
- A forest context: $a(b\,c)\ b(b(c\,c\,*)\,a)\ a$

Then we have

- $F \ominus G = F\,G$
- $F \oplus G = F[* \to G]$

A forest algebra expression is an expression that is built from the constants

- a and $a_* := a(*)$ for $a \in \Gamma$ and
- the binary operations $\ominus$ and $\oplus$.

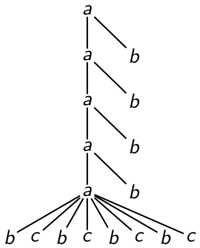
Expressions must be well-typed (F = type of forests, C = type of contexts):
F $\ominus$ F, F $\ominus$ C, C $\ominus$ F, C $\oplus$ F and C $\oplus$ C are allowed.

Forest straight-line programs

A forest straight-line program (FSLP) is a forest algebra expression that is represented as a DAG (Gascon, L, Maneth, Reh, Sieber 2018).

Forest straight-line programs

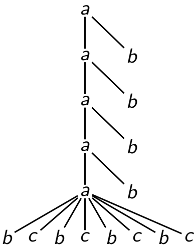
A forest straight-line program (FSLP) is a forest algebra expression that is represented as a DAG (Gascon, L, Maneth, Reh, Sieber 2018).



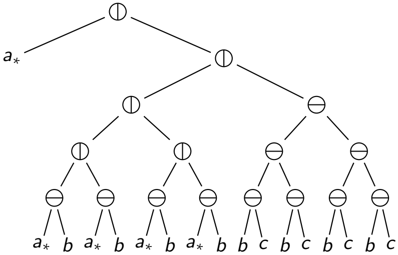
forest F

Forest straight-line programs

A forest straight-line program (FSLP) is a forest algebra expression that is represented as a DAG (Gascon, L, Maneth, Reh, Sieber 2018).



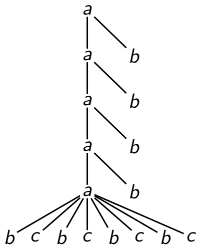
forest F



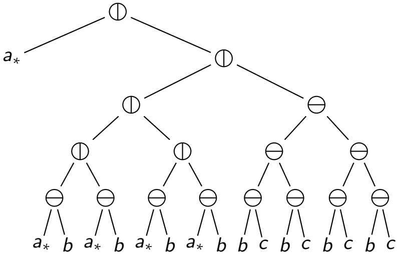
forest algebra expression for F

Forest straight-line programs

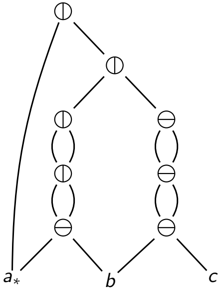
A forest straight-line program (FSLP) is a forest algebra expression that is represented as a DAG (Gascon, L, Maneth, Reh, Sieber 2018).



forest F



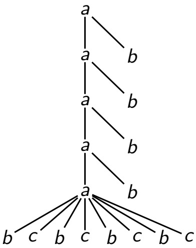
forest algebra expression for F



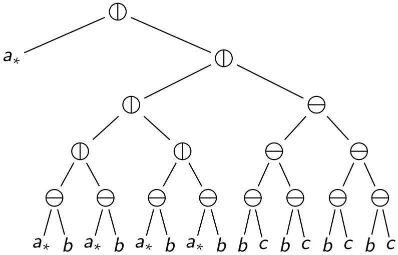
FSLP for F

Forest straight-line programs

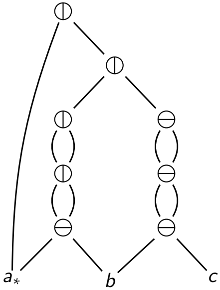
A forest straight-line program (FSLP) is a forest algebra expression that is represented as a DAG (Gascon, L, Maneth, Reh, Sieber 2018).



forest F



forest algebra expression for F



FSLP for F

For an FSLP $\mathcal{G}$ we denote with $\text{val}(\mathcal{G})$ the forest produced by $\mathcal{G}$.

Forest straight-line programs

- Two related formalisms:

Forest straight-line programs

- Two related formalisms:
  - tree straight-line programs (Bussato, L, Maneth 2005): for node-labelled binary trees

Forest straight-line programs

- Two related formalisms:
  - tree straight-line programs (Bussato, L, Maneth 2005): for node-labelled binary trees
  - top DAGs (Bille, Gørtz, Landau, Weimann 2013): very similar to FSLPs
- A (string) SLP is an FSLP that only uses the constants a for $a \in \Gamma$ and the operation $\ominus$.

  Such an FSLP produces a forest consisting of a chain of singleton trees.

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

Proof idea:

1. Partition the input forest F of size n into $\Theta\left(\frac{n}{\ell}\right)$ many subforests and subcontexts of size in $[\ell, 2\ell]$.

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\big(\frac{n}{\log_k n}\big)$.

Proof idea:

1. Partition the input forest F of size n into $\Theta\big(\frac{n}{\ell}\big)$ many subforests and subcontexts of size in $[\ell, 2\ell]$.
2. One can choose $\ell = \Theta(\log_k n)$ such that the total number of forests and contexts of size in $[\ell, 2\ell]$ is $\mathcal{O}\big(\frac{n}{\log_k n}\big)$.

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

Proof idea:

1. Partition the input forest F of size n into $\Theta\left(\frac{n}{\ell}\right)$ many subforests and subcontexts of size in $[\ell, 2\ell]$.
2. One can choose $\ell = \Theta(\log_k n)$ such that the total number of forests and contexts of size in $[\ell, 2\ell]$ is $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.
3. The FSLP consists of two parts, both of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$:

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

Proof idea:

1. Partition the input forest F of size n into $\Theta\left(\frac{n}{\ell}\right)$ many subforests and subcontexts of size in $[\ell, 2\ell]$.
2. One can choose $\ell = \Theta(\log_k n)$ such that the total number of forests and contexts of size in $[\ell, 2\ell]$ is $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.
3. The FSLP consists of two parts, both of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$:
   - A forest algebra expression for the forest obtained by contracting the subforests and subcontexts from 1.

Small FSLPs always exist

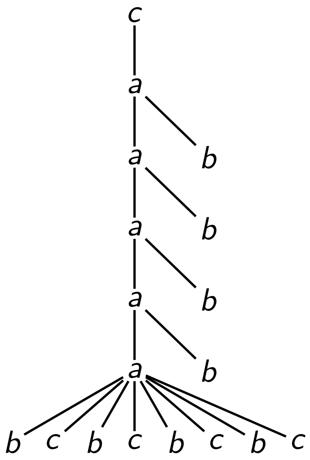
Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

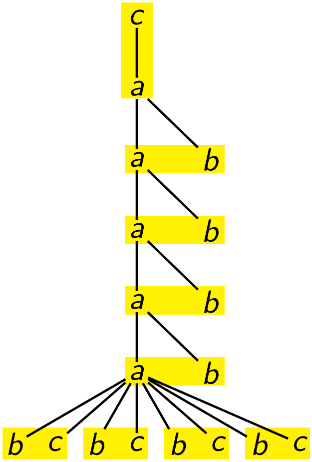
Proof idea:

1. Partition the input forest F of size n into $\Theta\left(\frac{n}{\ell}\right)$ many subforests and subcontexts of size in $[\ell, 2\ell]$.
2. One can choose $\ell = \Theta(\log_k n)$ such that the total number of forests and contexts of size in $[\ell, 2\ell]$ is $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.
3. The FSLP consists of two parts, both of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$:
    - A forest algebra expression for the forest obtained by contracting the subforests and subcontexts from 1.
    - A DAG producing the subforests and subcontexts from 1.

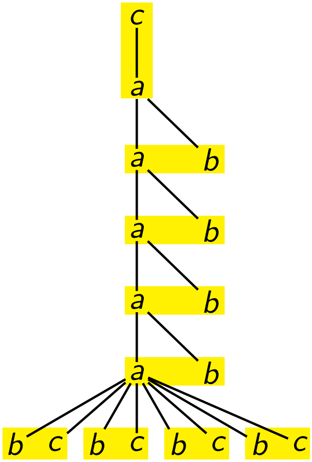
Small FSLPs always exist

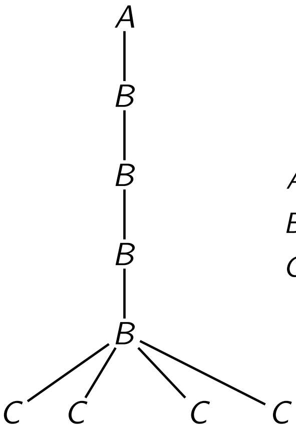


Small FSLPs always exist



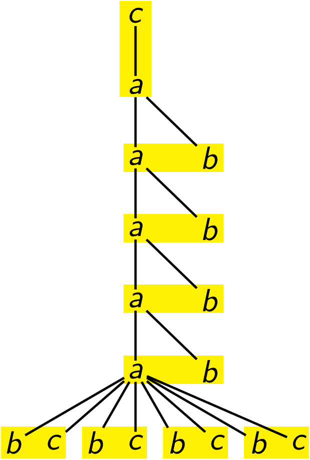
Small FSLPs always exist





$$A \rightarrow c_* \obar a_*$$
$$B \rightarrow a_* \ominus b$$
$$C \rightarrow b \ominus c$$

Small FSLPs always exist



$$X_0 \to A \oplus X_1$$
$$X_1 \to B \oplus X_2$$
$$X_2 \to B \oplus X_3$$
$$X_3 \to B \oplus X_4$$
$$X_4 \to B \oplus X_5$$
$$X_5 \to C \ominus X_6$$
$$X_6 \to C \ominus X_7$$
$$X_7 \to C \ominus X_8$$
$$X_7 \to C \ominus C$$

$$A \to c_* \oplus a_*$$
$$B \to a_* \ominus b$$
$$C \to b \ominus c$$

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

Related work:

- Ganardi, Hucke, L, Seelbach Benkner 2019:
  used for universal tree coding

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

Related work:

- Ganardi, Hucke, L, Seelbach Benkner 2019:
  used for universal tree coding
- Munro, Nicholson, Seelbach Benkner, Wild 2021:
  similar two-step approach; universal tree coding + efficient querying

Small FSLPs always exist

Hucke, L, Noeth 2014

There is an algorithm that produces in linear time from a tree (or forest) of size n with k different node labels an FSLP of size $\mathcal{O}\left(\frac{n}{\log_k n}\right)$.

Related work:

- Ganardi, Hucke, L, Seelbach Benkner 2019: used for universal tree coding
- Munro, Nicholson, Seelbach Benkner, Wild 2021: similar two-step approach; universal tree coding + efficient querying
- L, Maneth, Mennicke 2013: TreeRePair; a practical algorithm for computing small FSLPs

The smallest grammar problem for trees

L, Jeż 2013

There is a linear time algorithm that produces for every forest F of size n an FSLP of size $O(\log(n) \cdot \mathrm{opt}(F))$.

Balancing forest straight-line program

Ganardi, Jeż, L 2021

From a given FSLP $\mathcal{G}$ of size n such that $F := \mathrm{val}(\mathcal{G})$ has size N, one can compute in time $\mathcal{O}(n)$ an FSLP $\mathcal{H}$ such that:

- $\mathrm{val}(\mathcal{H}) = \mathrm{val}(\mathcal{G})$
- $|\mathcal{H}| \in \mathcal{O}(n)$
- $\mathrm{depth}(\mathcal{H}) \in \mathcal{O}(\log N)$

Balancing forest straight-line program

Ganardi, Jeż, L 2021

From a given FSLP $\mathcal{G}$ of size n such that $F := \mathrm{val}(\mathcal{G})$ has size N, one can compute in time $\mathcal{O}(n)$ an FSLP $\mathcal{H}$ such that:

- $\mathrm{val}(\mathcal{H}) = \mathrm{val}(\mathcal{G})$
- $|\mathcal{H}| \in \mathcal{O}(n)$
- $\mathrm{depth}(\mathcal{H}) \in \mathcal{O}(\log N)$

Corollary: random access in logarithmic time on compressed forests.

Balancing forest straight-line program

Ganardi, Jeż, L 2021

From a given FSLP $\mathcal{G}$ of size n such that $F := \mathsf{val}(\mathcal{G})$ has size N, one can compute in time $\mathcal{O}(n)$ an FSLP $\mathcal{H}$ such that:

- $\mathsf{val}(\mathcal{H}) = \mathsf{val}(\mathcal{G})$
- $|\mathcal{H}| \in \mathcal{O}(n)$
- $\mathsf{depth}(\mathcal{H}) \in \mathcal{O}(\log N)$

Corollary: random access in logarithmic time on compressed forests.

From a given FSLP $\mathcal{G}$ one can built in linear time a data structure that allows to solve for $F = \mathsf{val}(\mathcal{G})$ the following problem in time $\mathcal{O}(\log |F|)$:

- Input: a preorder number of a node v in F
- Output: the label of the node v.

FSLPs in database theory

Goal: For a given

- huge tree (e.g. an XML tree structure) that is stored compressed as an FSLP and
- a query formulated in a suitable query language

we want to enumerate all query results.

FSLPs in database theory

Goal: For a given

- huge tree (e.g. an XML tree structure) that is stored compressed as an FSLP and
- a query formulated in a suitable query language

we want to enumerate all query results.

We assume that queries are formulated in MSO (monadic 2nd order logic):

FSLPs in database theory

Goal: For a given

- huge tree (e.g. an XML tree structure) that is stored compressed as an FSLP and
- a query formulated in a suitable query language

we want to enumerate all query results.

We assume that queries are formulated in MSO (monadic 2nd order logic):

- there are two types of variables:
  - x, y, z, x' etc. for tree nodes
  - X, Y, Z, Z' etc. for sets of tree nodes

FSLPs in database theory

- atomic formulas (x, y are node variables, X is a node set variable):
  - $x = y$
  - $x \in X$,
  - $\text{label}(x) = a$ for $a \in \Gamma$
  - $\text{parent}(x, y)$ (x is the parent node of y)
  - $\text{leftsibling}(x, y)$ (x is the left sibling of y)

FSLPs in database theory

- atomic formulas (x, y are node variables, X is a node set variable):
  - $x = y$
  - $x \in X$,
  - $\text{label}(x) = a$ for $a \in \Gamma$
  - $\text{parent}(x, y)$ (x is the parent node of y)
  - $\text{leftsibling}(x, y)$ (x is the left sibling of y)
- larger formulas are constructed from atomic formulas using
  - boolean operators ($\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$) and
  - quantification ($\exists x : \phi$, $\forall x : \phi$, $\exists X : \phi$, $\forall X : \phi$)

FSLPs in database theory

- atomic formulas (x, y are node variables, X is a node set variable):
  - $x = y$
  - $x \in X$,
  - $\text{label}(x) = a$ for $a \in \Gamma$
  - $\text{parent}(x, y)$ (x is the parent node of y)
  - $\text{leftsibling}(x, y)$ (x is the left sibling of y)
- larger formulas are constructed from atomic formulas using
  - boolean operators ($\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$) and
  - quantification ($\exists x : \phi$, $\forall x : \phi$, $\exists X : \phi$, $\forall X : \phi$)

Consider now a forest F and an MSO formula $\phi(X)$ where X is the only free variable X in ϕ.

FSLPs in database theory

- atomic formulas (x, y are node variables, X is a node set variable):
  - $x = y$
  - $x \in X$,
  - $\text{label}(x) = a$ for $a \in \Gamma$
  - $\text{parent}(x, y)$ (x is the parent node of y)
  - $\text{leftsibling}(x, y)$ (x is the left sibling of y)
- larger formulas are constructed from atomic formulas using
  - boolean operators ($\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$) and
  - quantification ($\exists x : \phi$, $\forall x : \phi$, $\exists X : \phi$, $\forall X : \phi$)

Consider now a forest F and an MSO formula $\phi(X)$ where X is the only free variable X in ϕ.

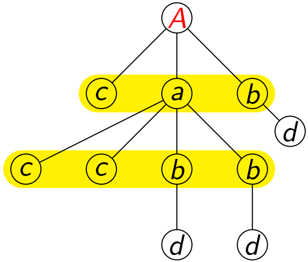
Then $\text{query}(\phi(X), F)$ is the sets $A \subseteq \text{nodes}(F)$ such that $F \models \phi(A)$.

FSLPs in database theory

Example: $\phi = \exists x(\mathsf{label}(x) = a \wedge \forall y : y \in X \longleftrightarrow \mathsf{parent}(x, y))$

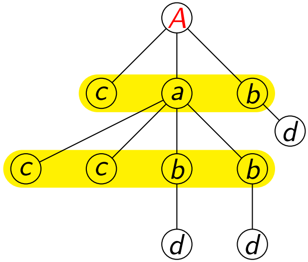
FSLPs in database theory

Example: $\phi = \exists x(\text{label}(x) = a \wedge \forall y : y \in X \longleftrightarrow \text{parent}(x, y))$



FSLPs in database theory

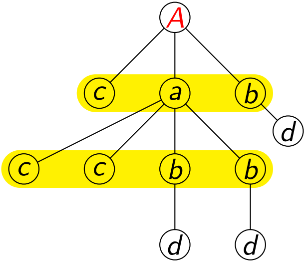
Example: $\phi = \exists x(\text{label}(x) = a \wedge \forall y : y \in X \longleftrightarrow \text{parent}(x,y))$



Note: $\text{query}(\phi(X), F)$ may have size $2^{|F|}$, i.e., size $2^{2^{\mathcal{O}(|\mathcal{G}|)}}$ if F is given by the FSLP $\mathcal{G}$.

FSLPs in database theory

Example: $\phi = \exists x(\text{label}(x) = a \wedge \forall y : y \in X \longleftrightarrow \text{parent}(x, y))$



Note: $\text{query}(\phi(X), F)$ may have size $2^{|F|}$, i.e., size $2^{2^{\mathcal{O}(|\mathcal{G}|)}}$ if F is given by the FSLP $\mathcal{G}$.

What does it mean to enumerate efficiently $\text{query}(\phi(X), \text{val}(\mathcal{G}))$?

Enumeration problems

- An enumeration problem is a function E that maps an input x to a finite set $E(x) = \{y_1, \ldots, y_k\}$ of k different objects y_i.

Enumeration problems

- An enumeration problem is a function E that maps an input x to a finite set $E(x) = \{y_1, \ldots, y_k\}$ of k different objects y_i.
- An enumeration algorithm $\mathcal{A}$ for E is an algorithm that prints on input x sequentially a list $y_{\pi(1)}, y_{\pi(1)}, \ldots, y_{\pi(k)}$ for a permutation π.

Enumeration problems

- An enumeration problem is a function E that maps an input x to a finite set $E(x) = \{y_1, \ldots, y_k\}$ of k different objects y_i.
- An enumeration algorithm $\mathcal{A}$ for E is an algorithm that prints on input x sequentially a list $y_{\pi(1)}, y_{\pi(1)}, \ldots, y_{\pi(k)}$ for a permutation π.
- $\mathcal{A}$ starts with a preprocessing phase finishing at time $t_0 =: T_{\mathsf{pre}}(x)$.

Enumeration problems

- An enumeration problem is a function E that maps an input x to a finite set $E(x) = \{y_1, \ldots, y_k\}$ of k different objects y_i.
- An enumeration algorithm $\mathcal{A}$ for E is an algorithm that prints on input x sequentially a list $y_{\pi(1)}, y_{\pi(1)}, \ldots, y_{\pi(k)}$ for a permutation π.
- $\mathcal{A}$ starts with a preprocessing phase finishing at time $t_0 =: T_{\text{pre}}(x)$.
- $\mathcal{A}$ works in linear preprocessing time if $T_{\text{pre}}(x) \leq \mathcal{O}(|x|)$.

Enumeration problems

- An enumeration problem is a function E that maps an input x to a finite set $E(x) = \{y_1, \dots, y_k\}$ of k different objects y_i.
- An enumeration algorithm $\mathcal{A}$ for E is an algorithm that prints on input x sequentially a list $y_{\pi(1)}, y_{\pi(1)}, \dots, y_{\pi(k)}$ for a permutation π.
- $\mathcal{A}$ starts with a preprocessing phase finishing at time $t_0 =: T_{\text{pre}}(x)$.
- $\mathcal{A}$ works in linear preprocessing time if $T_{\text{pre}}(x) \leq \mathcal{O}(|x|)$.
- Assume that printing $y_{\pi(i)}$ is completed at time t_i $(t_1 < t_2 < \dots < t_k)$.

Enumeration problems

- An enumeration problem is a function E that maps an input x to a finite set $E(x) = \{y_1, \ldots, y_k\}$ of k different objects y_i.
- An enumeration algorithm $\mathcal{A}$ for E is an algorithm that prints on input x sequentially a list $y_{\pi(1)}, y_{\pi(1)}, \ldots, y_{\pi(k)}$ for a permutation π.
- $\mathcal{A}$ starts with a preprocessing phase finishing at time $t_0 =: T_{\mathsf{pre}}(x)$.
- $\mathcal{A}$ works in linear preprocessing time if $T_{\mathsf{pre}}(x) \leq \mathcal{O}(|x|)$.
- Assume that printing $y_{\pi(i)}$ is completed at time t_i $(t_1 < t_2 < \cdots < t_k)$.
- $\mathcal{A}$ works in output-linear delay if $t_i - t_{i-1} \leq \mathcal{O}(|y_i|)$ for all $1 \leq i \leq k$.

FSLPs in database theory

L, Schmid 2024

Fix a query $\phi(X)$. One can enumerate $\text{query}(\phi(X), \text{val}(\mathcal{G}))$ for a given FSLP $\mathcal{G}$ in

- linear preprocessing time and
- output-linear delay.

FSLPs in database theory

L, Schmid 2024

Fix a query $\phi(X)$. One can enumerate query$(\phi(X), \mathsf{val}(\mathcal{G}))$ for a given FSLP $\mathcal{G}$ in

- linear preprocessing time and
- output-linear delay.

Previous results:

- Bagan 2006, Courcelle 2009: linear preprocessing and output-linear delay for uncompressed trees

FSLPs in database theory

L, Schmid 2024

Fix a query $\phi(X)$. One can enumerate query$(\phi(X), \mathsf{val}(\mathcal{G}))$ for a given FSLP $\mathcal{G}$ in

- linear preprocessing time and
- output-linear delay.

Previous results:

- Bagan 2006, Courcelle 2009: linear preprocessing and output-linear delay for uncompressed trees
- Schmid, Schweikardt 2021: linear preprocessing and logarithmic delay for compressed strings (and a fragment of MSO)

FSLPs in database theory

L, Schmid 2024

Fix a query $\phi(X)$. One can enumerate query$(\phi(X), \mathsf{val}(\mathcal{G}))$ for a given FSLP $\mathcal{G}$ in

- linear preprocessing time and
- output-linear delay.

Previous results:

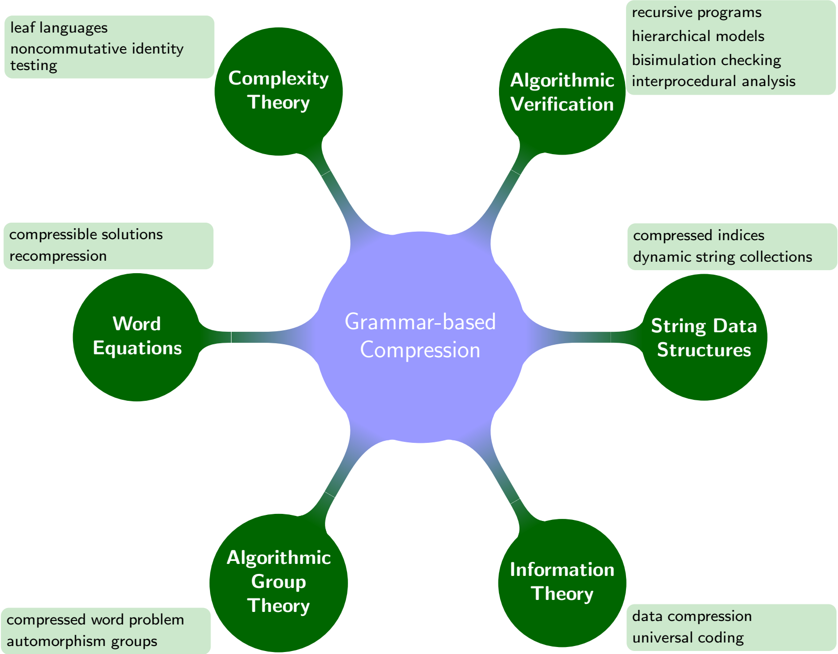
- Bagan 2006, Courcelle 2009: linear preprocessing and output-linear delay for uncompressed trees
- Schmid, Schweikardt 2021: linear preprocessing and logarithmic delay for compressed strings (and a fragment of MSO)
- Muñoz, Riveros 2023: linear preprocessing and output-linear delay for compressed strings (and a fragment of MSO)

FSLPs in database theory

Proof strategy: Let Γ be the set node labels of our trees.

1. Translate the MSO-query $\phi(X)$ into a node-selecting tree automaton $\mathcal{A}$ (a tree automaton working on the label set $\Gamma \times \{0, 1\}$).
2. Reduce enumeration of $\text{query}(\mathcal{A}, \text{val}(\mathcal{G}))$ to the enumeration of $\text{query}(\mathcal{B}, \text{unfold}(\mathcal{G}))$, where
   - $\text{unfold}(\mathcal{G})$ is the forest algebra expression obtained by unfolding $\mathcal{G}$ and
   - $\mathcal{B}$ is a leaf-selecting tree automaton.
3. Bagan solved the previous enumeration problem for the case where the tree $\text{unfold}(\mathcal{G})$ is given explicitly.

   We extend Bagan's algorithm to DAG-compressed trees.


Grammar-based Compression
Complexity Theory
leaf languages
noncommutative identity testing
Algorithmic Verification
recursive programs
hierarchical models
bisimulation checking
interprocedural analysis
Word Equations
compressible solutions
recompression
String Data Structures
compressed indices
dynamic string collections
Algorithmic Group Theory
compressed word problem
automorphism groups
Information Theory
data compression
universal coding