Efficient Answer Set Counting with aspmc

Thomas Eiter, Markus Hecher, Rafael Kiesel

Vienna University of Technology

funded by FWF project W1255-N23

20th of September 2021

Der Wissenschaftsfonds.

LOGICAL METHODS IN COMPUTER SCIENCE

Algebraic Answer Set Counting

Interest in reasoning problems for Answer Set Programming (ASP) that go beyond consistency and entailment:

Algebraic Answer Set Counting

Interest in reasoning problems for Answer Set Programming (ASP) that go beyond consistency and entailment:

- Probabilistic Reasoning [De Raedt *et al.*, 2007], [Lee and Yang, 2017], [Baral *et al.*, 2009]

Algebraic Answer Set Counting

Interest in reasoning problems for Answer Set Programming (ASP) that go beyond consistency and entailment:

- Probabilistic Reasoning [De Raedt *et al.*, 2007], [Lee and Yang, 2017], [Baral *et al.*, 2009]
- Preferential Reasoning [Brewka *et al.*, 2015]

Algebraic Answer Set Counting

Interest in reasoning problems for Answer Set Programming (ASP) that go beyond consistency and entailment:

- Probabilistic Reasoning [De Raedt *et al.*, 2007], [Lee and Yang, 2017], [Baral *et al.*, 2009]
- Preferential Reasoning [Brewka *et al.*, 2015]
- Algebraic Answer Set Counting (AASC) [Eiter and Kiesel, 2020], [Kimmig *et al.*, 2011]

Solving AASC I

- Solving AASC instances can be #P-, NP- or OptP-hard

Solving AASC I

- Solving AASC instances can be #P-, NP- or OptP-hard
- Solvable via
  1. answer set enumeration
     $\hookrightarrow$ only feasible for "few" answer sets

Solving AASC I

- Solving AASC instances can be #P-, NP- or OptP-hard
- Solvable via
  1. answer set enumeration
     $\hookrightarrow$ only feasible for "few" answer sets
  2. dynamic programming on a tree decomposition
     $\hookrightarrow$ only feasible for very low treewidth

Solving AASC I

- Solving AASC instances can be #P-, NP- or OptP-hard
- Solvable via
  1. answer set enumeration
     $\hookrightarrow$ only feasible for "few" answer sets
  2. dynamic programming on a tree decomposition
     $\hookrightarrow$ only feasible for very low treewidth
  3. compilation into a *tractable circuit representation* like d-DNNF or SDD
     $\hookrightarrow$ compilers like c2d [Darwiche, 2004] work on CNFs

Solving AASC II

- We favor compilation
  $\hookrightarrow$ translate ASP to CNF via *cycle-breaking*

Solving AASC II

- We favor compilation
  $\hookrightarrow$ translate ASP to CNF via *cycle-breaking*
- Compilation has performance guarantees based on *treewidth*
  $\hookrightarrow$ find *treewidth-aware* cycle-breaking

Example Program

Non-ground smokers program

$$
\begin{aligned}
0.3 :: \mathrm{stress}(X) &\leftarrow \mathrm{person}(X) \\
\mathrm{smokes}(X) &\leftarrow \mathrm{stress}(X) \\
0.2 :: \mathrm{inf}(X, Y) &\leftarrow \mathrm{friend}(X, Y) \\
\mathrm{smokes}(Y) &\leftarrow \mathrm{smokes}(X), \mathrm{inf}(X, Y)
\end{aligned}
$$

Example Program

Non-ground smokers program

$$
\begin{aligned}
0.3 :: \mathrm{stress}(X) &\leftarrow \mathrm{person}(X) \\
\mathrm{smokes}(X) &\leftarrow \mathrm{stress}(X) \\
0.2 :: \mathrm{inf}(X, Y) &\leftarrow \mathrm{friend}(X, Y) \\
\mathrm{smokes}(Y) &\leftarrow \mathrm{smokes}(X), \mathrm{inf}(X, Y)
\end{aligned}
$$

And input data

$$
\begin{array}{lll}
\mathrm{person}(1) & \mathrm{person}(2) & \mathrm{person}(3) \\
\mathrm{friend}(1, 2) & \mathrm{friend}(2, 3) & \mathrm{friend}(3, 1)
\end{array}
$$

Example cont.

Putting the input data and the non-ground program together results in

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) &\leftarrow && \text{for } x = 1, 2, 3 \\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1, 2, 3 \\
0.2 :: \mathrm{inf}(x, y) &\leftarrow && \text{for } x + 1 \equiv y \bmod 3 \\
\mathrm{smokes}(y) &\leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) && \text{for } x + 1 \equiv y \bmod 3
\end{aligned}
$$

(Positive) Dependency Graph

The (positive) *dependency graph* of a program Π is the digraph $\mathrm{DEP}(\Pi) = (V, E)$, where

- $V = \mathcal{A}(\Pi)$ is the set of propositional variables that occur in Π
- $(b, a) \in E$ if there is a rule $a \leftarrow b_1, \ldots, b_n, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m$ in Π with $b = b_i$

(Positive) Dependency Graph

The (positive) *dependency graph* of a program Π is the digraph $\mathrm{DEP}(\Pi) = (V, E)$, where

- $V = \mathcal{A}(\Pi)$ is the set of propositional variables that occur in Π
- $(b, a) \in E$ if there is a rule $a \leftarrow b_1, \ldots, b_n, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m$ in Π with $b = b_i$

Theorem ([Fages, 1994])

If $\mathrm{DEP}(\Pi)$ *is acyclic, then* $\mathrm{Clark}(\Pi)$*, the Clark-completion of* Π*, is a propositional formula whose models are the answer sets of* Π*.*

Example cont.

The ground program...

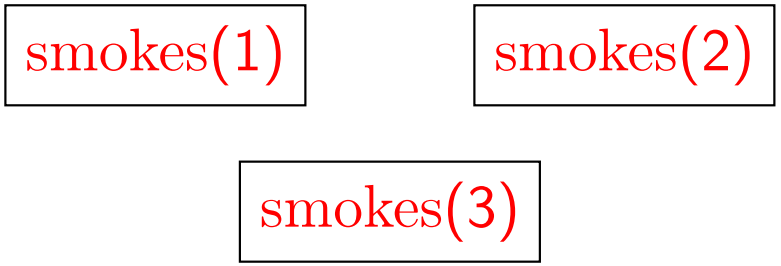
$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) \leftarrow & \qquad && \text{for } x = 1, 2, 3 \\
\mathrm{smokes}(x) \leftarrow \mathrm{stress}(x) & && \text{for } x = 1, 2, 3 \\
0.2 :: \mathrm{inf}(x, y) \leftarrow & && \text{for } x + 1 \equiv y \bmod 3 \\
\mathrm{smokes}(y) \leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) & && \text{for } x + 1 \equiv y \bmod 3
\end{aligned}
$$

Example cont.

The ground program...

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) \leftarrow & \qquad && \text{for } x = 1, 2, 3 \\
\mathrm{smokes}(x) \leftarrow \mathrm{stress}(x) & && \text{for } x = 1, 2, 3 \\
0.2 :: \mathrm{inf}(x, y) \leftarrow & && \text{for } x + 1 \equiv y \bmod 3 \\
\mathrm{smokes}(y) \leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) & && \text{for } x + 1 \equiv y \bmod 3
\end{aligned}
$$

...has dependency graph

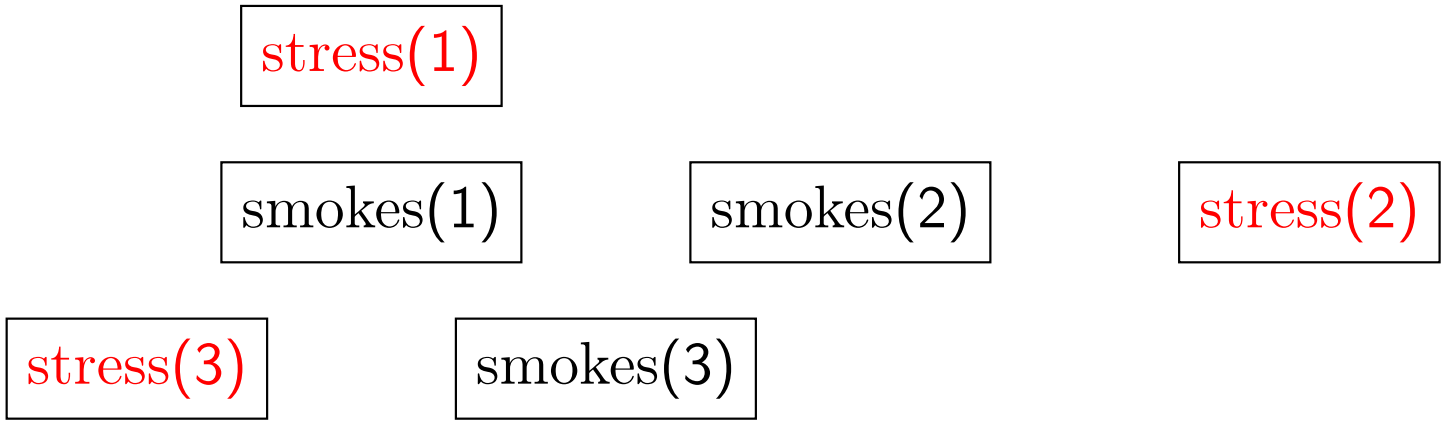


Example cont.

The ground program...

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) &\leftarrow && \text{for } x = 1, 2, 3 \\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1, 2, 3 \\
0.2 :: \mathrm{inf}(x, y) &\leftarrow && \text{for } x + 1 \equiv y \bmod 3 \\
\mathrm{smokes}(y) &\leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) && \text{for } x + 1 \equiv y \bmod 3
\end{aligned}
$$

...has dependency graph

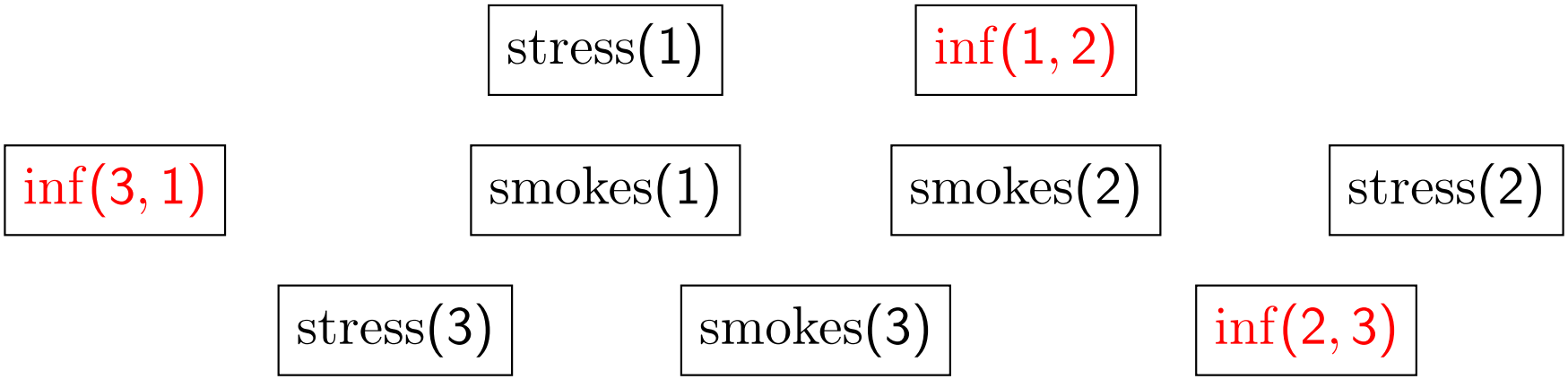


Example cont.

The ground program...

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) &\leftarrow && \text{for } x = 1, 2, 3 \\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1, 2, 3 \\
0.2 :: \mathrm{inf}(x, y) &\leftarrow && \text{for } x + 1 \equiv y \bmod 3 \\
\mathrm{smokes}(y) &\leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) && \text{for } x + 1 \equiv y \bmod 3
\end{aligned}
$$

...has dependency graph

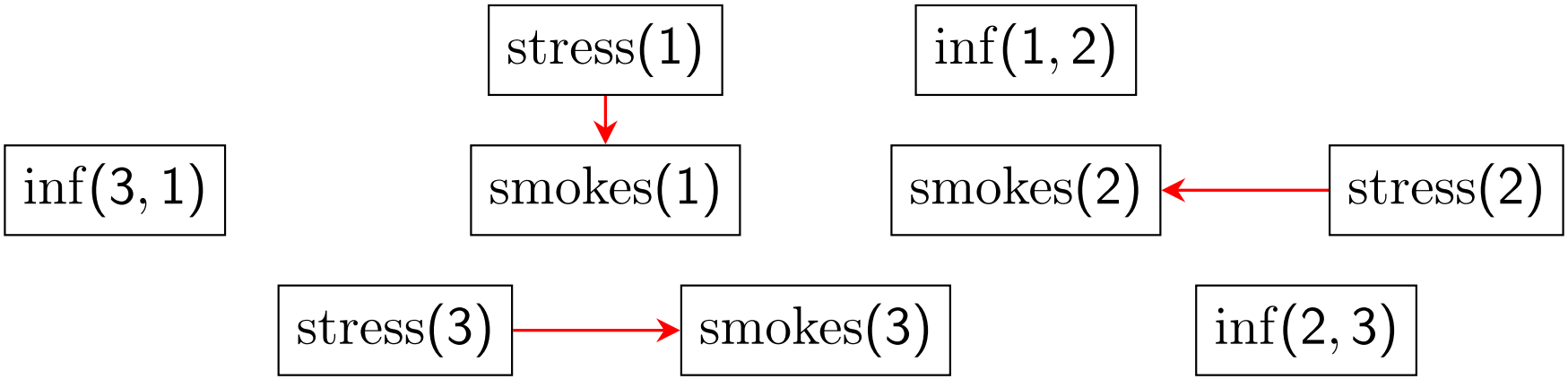


Example cont.

The ground program...

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) \leftarrow & \qquad && \text{for } x = 1, 2, 3 \\
\mathrm{smokes}(x) \leftarrow \mathrm{stress}(x) & && \text{for } x = 1, 2, 3 \\
0.2 :: \mathrm{inf}(x, y) \leftarrow & && \text{for } x + 1 \equiv y \bmod 3 \\
\mathrm{smokes}(y) \leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) & && \text{for } x + 1 \equiv y \bmod 3
\end{aligned}
$$

...has dependency graph

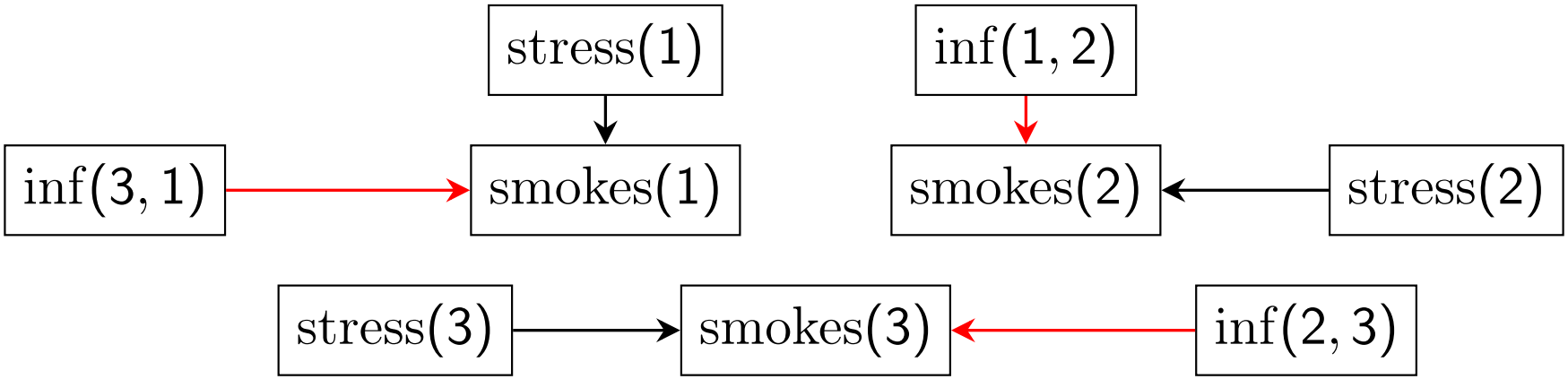


Example cont.

The ground program...

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) &\leftarrow && \text{for } x = 1, 2, 3 \\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1, 2, 3 \\
0.2 :: \mathrm{inf}(x, y) &\leftarrow && \text{for } x + 1 \equiv y \bmod 3 \\
\mathrm{smokes}(y) &\leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) && \text{for } x + 1 \equiv y \bmod 3
\end{aligned}
$$

...has dependency graph

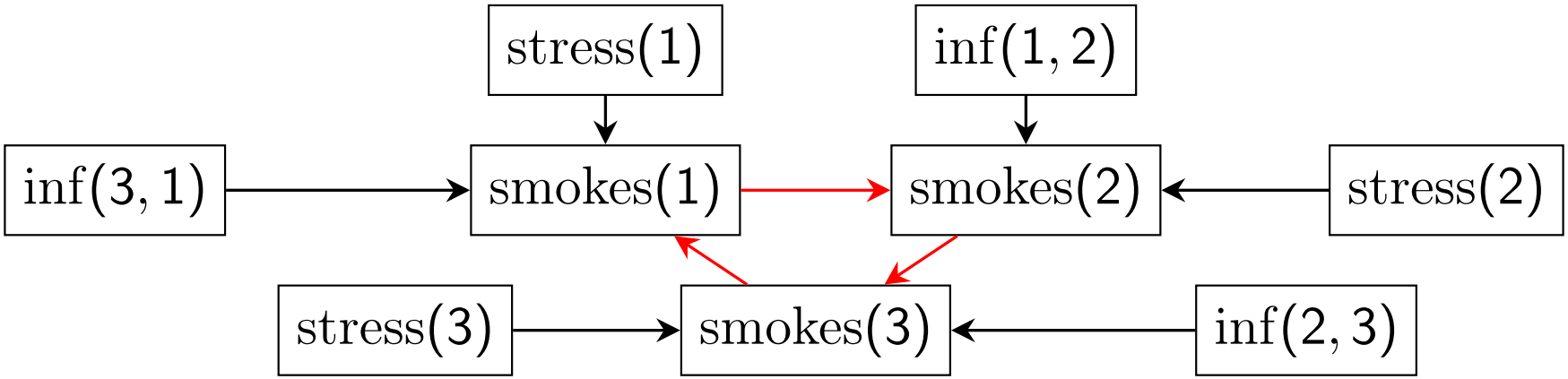


Example cont.

The ground program...

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) \leftarrow & \qquad && \text{for } x = 1, 2, 3 \\
\mathrm{smokes}(x) \leftarrow \mathrm{stress}(x) & && \text{for } x = 1, 2, 3 \\
0.2 :: \mathrm{inf}(x, y) \leftarrow & && \text{for } x + 1 \equiv y \bmod 3 \\
\mathrm{smokes}(y) \leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) & && \text{for } x + 1 \equiv y \bmod 3
\end{aligned}
$$

...has dependency graph



Idea

- We need all the possible *acyclic* derivations for each atom

Idea

- We need all the possible *acyclic* derivations for each atom
- **Problem:** we can only get all the derivations if we have those of the other atoms

Idea

- We need all the possible *acyclic* derivations for each atom
- **Problem:** we can only get all the derivations if we have those of the other atoms
- **Idea:** introduce copies of atoms that
  - capture increasing subsets of the derivations

Idea

- We need all the possible *acyclic* derivations for each atom
- **Problem:** we can only get all the derivations if we have those of the other atoms
- **Idea:** introduce copies of atoms that
  - capture increasing subsets of the derivations
  - only use derivations with probabilistic facts or already introduced copies

Idea

- We need all the possible *acyclic* derivations for each atom
- **Problem:** we can only get all the derivations if we have those of the other atoms
- **Idea:** introduce copies of atoms that
  - capture increasing subsets of the derivations
  - only use derivations with probabilistic facts or already introduced copies
- Do this iteratively until a (semantic) fixed point is reached

Idea

- We need all the possible *acyclic* derivations for each atom
- **Problem:** we can only get all the derivations if we have those of the other atoms
- **Idea:** introduce copies of atoms that
  - capture increasing subsets of the derivations
  - only use derivations with probabilistic facts or already introduced copies
- Do this iteratively until a (semantic) fixed point is reached
- Insight: the order in which atoms are considered is crucial!

Example cont.

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) &\leftarrow && \text{for } x = 1, 2, 3 && (1)\\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1, 2, 3 && \\
0.2 :: \mathrm{inf}(x, y) &\leftarrow && \text{for } x + 1 \equiv y \bmod 3 && (2)\\
\mathrm{smokes}(y) &\leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) && \text{for } x + 1 \equiv y \bmod 3 &&
\end{aligned}
$$

Can be unfolded to (1), (2) and

Example cont.

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) &\leftarrow && \text{for } x = 1,2,3 && (1)\\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1,2,3 && \\
0.2 :: \mathrm{inf}(x,y) &\leftarrow && \text{for } x+1 \equiv y \bmod 3 && (2)\\
\mathrm{smokes}(y) &\leftarrow \mathrm{smokes}(x), \mathrm{inf}(x,y) && \text{for } x+1 \equiv y \bmod 3 &&
\end{aligned}
$$

Can be unfolded to (1), (2) and

$\mathrm{smokes}(1)^1 \leftarrow \mathrm{stress}(1)$ $\quad$ $\mathrm{smokes}(1)^1 \leftarrow \mathrm{inf}(3,1), \bot$

no copy of $\mathrm{smokes}(3) \hookrightarrow$ use falsum

Example cont.

$$0.3 :: \mathrm{stress}(x) \leftarrow \qquad \text{for } x = 1, 2, 3 \qquad (1)$$

$$\mathrm{smokes}(x) \leftarrow \mathrm{stress}(x) \qquad \text{for } x = 1, 2, 3$$

$$0.2 :: \mathrm{inf}(x, y) \leftarrow \qquad \text{for } x + 1 \equiv y \bmod 3 \qquad (2)$$

$$\mathrm{smokes}(y) \leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) \qquad \text{for } x + 1 \equiv y \bmod 3$$

Can be unfolded to (1), (2) and

$$\mathrm{smokes}(1)^1 \leftarrow \mathrm{stress}(1) \qquad \mathrm{smokes}(1)^1 \leftarrow \mathrm{inf}(3, 1), \bot$$

$$\mathrm{smokes}(2)^1 \leftarrow \mathrm{stress}(2) \qquad \mathrm{smokes}(2)^1 \leftarrow \mathrm{inf}(1, 2), \mathrm{smokes}(1)^1$$

Example cont.

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) &\leftarrow && \text{for } x = 1,2,3 && (1)\\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1,2,3 && \\
0.2 :: \mathrm{inf}(x,y) &\leftarrow && \text{for } x+1 \equiv y \bmod 3 && (2)\\
\mathrm{smokes}(y) &\leftarrow \mathrm{smokes}(x), \mathrm{inf}(x,y) && \text{for } x+1 \equiv y \bmod 3 &&
\end{aligned}
$$

Can be unfolded to (1), (2) and

$$
\begin{aligned}
\mathrm{smokes}(1)^1 &\leftarrow \mathrm{stress}(1) & \mathrm{smokes}(1)^1 &\leftarrow \mathrm{inf}(3,1), \bot\\
\mathrm{smokes}(2)^1 &\leftarrow \mathrm{stress}(2) & \mathrm{smokes}(2)^1 &\leftarrow \mathrm{inf}(1,2), \mathrm{smokes}(1)^1\\
\mathrm{smokes}(3)^1 &\leftarrow \mathrm{stress}(3) & \mathrm{smokes}(3)^1 &\leftarrow \mathrm{inf}(2,3), \mathrm{smokes}(2)^1
\end{aligned}
$$

Example cont.

$$0.3 :: \mathrm{stress}(x) \leftarrow \qquad \text{for } x = 1, 2, 3 \qquad (1)$$
$$\mathrm{smokes}(x) \leftarrow \mathrm{stress}(x) \qquad \text{for } x = 1, 2, 3$$
$$0.2 :: \mathrm{inf}(x, y) \leftarrow \qquad \text{for } x + 1 \equiv y \bmod 3 \qquad (2)$$
$$\mathrm{smokes}(y) \leftarrow \mathrm{smokes}(x), \mathrm{inf}(x, y) \qquad \text{for } x + 1 \equiv y \bmod 3$$

Can be unfolded to (1), (2) and

$$\mathrm{smokes}(1)^1 \leftarrow \mathrm{stress}(1) \quad \mathrm{smokes}(1)^1 \leftarrow \mathrm{inf}(3, 1), \bot$$
$$\mathrm{smokes}(2)^1 \leftarrow \mathrm{stress}(2) \quad \mathrm{smokes}(2)^1 \leftarrow \mathrm{inf}(1, 2), \mathrm{smokes}(1)^1$$
$$\mathrm{smokes}(3)^1 \leftarrow \mathrm{stress}(3) \quad \mathrm{smokes}(3)^1 \leftarrow \mathrm{inf}(2, 3), \mathrm{smokes}(2)^1$$
$$\mathrm{smokes}(1)^2 \leftarrow \mathrm{stress}(1) \quad \mathrm{smokes}(1)^2 \leftarrow \mathrm{inf}(3, 1), \mathrm{smokes}(3)^1$$

Example cont.

$$
\begin{aligned}
0.3 :: \mathrm{stress}(x) &\leftarrow && \text{for } x = 1,2,3 && (1)\\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1,2,3 \\
0.2 :: \mathrm{inf}(x,y) &\leftarrow && \text{for } x+1 \equiv y \bmod 3 && (2)\\
\mathrm{smokes}(y) &\leftarrow \mathrm{smokes}(x), \mathrm{inf}(x,y) && \text{for } x+1 \equiv y \bmod 3
\end{aligned}
$$

Can be unfolded to (1), (2) and

$$
\begin{aligned}
\mathrm{smokes}(1)^1 &\leftarrow \mathrm{stress}(1) & \mathrm{smokes}(1)^1 &\leftarrow \mathrm{inf}(3,1), \bot\\
\mathrm{smokes}(2)^1 &\leftarrow \mathrm{stress}(2) & \mathrm{smokes}(2)^1 &\leftarrow \mathrm{inf}(1,2), \mathrm{smokes}(1)^1\\
\mathrm{smokes}(3)^1 &\leftarrow \mathrm{stress}(3) & \mathrm{smokes}(3)^1 &\leftarrow \mathrm{inf}(2,3), \mathrm{smokes}(2)^1\\
\mathrm{smokes}(1)^2 &\leftarrow \mathrm{stress}(1) & \mathrm{smokes}(1)^2 &\leftarrow \mathrm{inf}(3,1), \mathrm{smokes}(3)^1\\
\mathrm{smokes}(2)^2 &\leftarrow \mathrm{stress}(2) & \mathrm{smokes}(2)^2 &\leftarrow \mathrm{inf}(1,2), \mathrm{smokes}(1)^2
\end{aligned}
$$

Example cont.

$$
\begin{array}{rll}
0.3 :: \mathrm{stress}(x) \leftarrow & \text{for } x = 1,2,3 & (1) \\
\mathrm{smokes}(x) \leftarrow \mathrm{stress}(x) & \text{for } x = 1,2,3 & \\
0.2 :: \mathrm{inf}(x,y) \leftarrow & \text{for } x+1 \equiv y \bmod 3 & (2) \\
\mathrm{smokes}(y) \leftarrow \mathrm{smokes}(x), \mathrm{inf}(x,y) & \text{for } x+1 \equiv y \bmod 3 &
\end{array}
$$

Can be unfolded to (1), (2) and

$$
\begin{array}{ll}
\mathrm{smokes}(1)^1 \leftarrow \mathrm{stress}(1) & \mathrm{smokes}(1)^1 \leftarrow \mathrm{inf}(3,1), \bot \\
\mathrm{smokes}(2)^1 \leftarrow \mathrm{stress}(2) & \mathrm{smokes}(2)^1 \leftarrow \mathrm{inf}(1,2), \mathrm{smokes}(1)^1 \\
\mathrm{smokes}(3) \leftarrow \mathrm{stress}(3) & \mathrm{smokes}(3) \leftarrow \mathrm{inf}(2,3), \mathrm{smokes}(2)^1 \\
\mathrm{smokes}(1) \leftarrow \mathrm{stress}(1) & \mathrm{smokes}(1) \leftarrow \mathrm{inf}(3,1), \mathrm{smokes}(3) \\
\mathrm{smokes}(2) \leftarrow \mathrm{stress}(2) & \mathrm{smokes}(2) \leftarrow \mathrm{inf}(1,2), \mathrm{smokes}(1)
\end{array}
$$

Algorithm 1 $\mathrm{T}_{\mathcal{P}}$-Unfold(Π, s)

Input A program Π and an unfolding sequence $s \in \mathcal{A}(\Pi)^*$.
Output An acyclic program Π'.

```
last = {a ↦ ⊥ | a ∈ A(Π)}
cnt = {a ↦ 0 | a ∈ A(Π)}
for i = 1, ..., len(s) do
    if isLastOccurrence(s_i, i, s) then
        head = s_i
    else
        head = s_i^(cnt(s_i)+1)
    for s_i ← b_1, ..., b_n, not c_1, ..., not c_m ∈ Π do
        Add head ← last(b_1), ..., last(b_n), not c_1, ..., not c_m
    last(s_i) = head
    cnt(s_i) = cnt(s_i) + 1
```

Algorithm 2 $\mathrm{T}_{\mathcal{P}}$-Unfold(Π, s)

Input A program Π and an unfolding sequence $s \in \mathcal{A}(\Pi)^*$.
Output An acyclic program Π'.

1: last $= \{a \mapsto \bot \mid a \in \mathcal{A}(\Pi)\}$
2: cnt $= \{a \mapsto 0 \mid a \in \mathcal{A}(\Pi)\}$
3: **for** $i = 1, \ldots, \mathrm{len}(s)$ **do**
4: **if** isLastOccurrence(s_i, i, s) **then**
5: head $= s_i$
6: **else**
7: head $= s_i^{\mathrm{cnt}(s_i)+1}$
8: **for** $s_i \leftarrow b_1, \ldots, b_n, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m \in \Pi$ **do**
9: Add head $\leftarrow \mathrm{last}(b_1), \ldots, \mathrm{last}(b_n), \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m$
10: last(s_i) = head
11: cnt(s_i) = cnt(s_i) + 1

Properties

- Acyclicity ?

Properties

- Acyclicity ✓

Properties

- Acyclicity ✓
- Faithfulness, i.e., bijective preserving of answer sets ?

Properties

- Acyclicity ✓
- Faithfulness, i.e., bijective preserving of answer sets

Theorem

✓, *if for every simple directed path* $\pi = (a_1, \ldots, a_n)$ *in* $\mathrm{DEP}(\Pi)$ *there is a directed path* $\pi_c = (a_1^{c_1}, \ldots, a_n^{c_n})$ *in* $\mathrm{DEP}(\mathrm{T}_{\mathcal{P}}\text{-Unfold}(\Pi, s))$.

Good Unfolding Sequences?

- We need path preserving unfolding sequences

Good Unfolding Sequences?

- We need path preserving unfolding sequences
- What else makes an unfolding sequence *good*?

Good Unfolding Sequences?

- We need path preserving unfolding sequences
- What else makes an unfolding sequence *good*?

Lemma

Let Π be an answer set program with treewidth k and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. If every variable occurs at most m times in s, then the treewidth of $\mathrm{T}_{\mathcal{P}}$-Unfold(Π, s) is less or equal to $k \cdot m$.

Good Unfolding Sequences?

- We need path preserving unfolding sequences
- What else makes an unfolding sequence *good*?

Lemma

Let Π be an answer set program with treewidth k and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. If every variable occurs at most m times in s, then the treewidth of $\mathrm{T}_{\mathcal{P}}$-Unfold(Π, s) is less or equal to $k \cdot m$.

- We are interested in path preserving m-unfolding sequences with small m

Component-Boosted Backdoor Size

Definition ($\mathrm{cbs}(G)$)

Let G be a digraph. Then $\mathrm{cbs}(G)$, the component-boosted backdoor size of G, is

- *1, if G is acyclic (which includes $V(G) = \emptyset$)*

Component-Boosted Backdoor Size

Definition ($\mathrm{cbs}(G)$)

Let G be a digraph. Then $\mathrm{cbs}(G)$, the component-boosted backdoor size of G, is

- *1, if G is acyclic (which includes $V(G) = \emptyset$)*
- *2, if G is a polytree, i.e. the undirected version of G is connected and acyclic*

Component-Boosted Backdoor Size

Definition ($\mathrm{cbs}(G)$)

Let G be a digraph. Then $\mathrm{cbs}(G)$, the component-boosted backdoor size of G, is

- *1, if G is acyclic (which includes $V(G) = \emptyset$)*
- *2, if G is a polytree, i.e. the undirected version of G is connected and acyclic*
- $\max\{\mathrm{cbs}(C) \mid C \in \mathrm{SCC}(G)\}$, *if G is cyclic but not strongly connected*

Component-Boosted Backdoor Size

Definition ($\mathrm{cbs}(G)$)

Let G be a digraph. Then $\mathrm{cbs}(G)$, the component-boosted backdoor size of G, is

- *1, if G is acyclic (which includes $V(G) = \emptyset$)*
- *2, if G is a polytree, i.e. the undirected version of G is connected and acyclic*
- $\max\{\mathrm{cbs}(C) \mid C \in \mathrm{SCC}(G)\}$, *if G is cyclic but not strongly connected*
- $\min\{\mathrm{cbs}(G \setminus S) \cdot (|S| + 1) \mid S \subseteq V(G)\}$ *otherwise*

Component-Boosted Backdoor Size

Definition ($\mathrm{cbs}(G)$)

Let G be a digraph. Then $\mathrm{cbs}(G)$, the component-boosted backdoor size of G, is

- *1, if G is acyclic (which includes $V(G) = \emptyset$)*
- *2, if G is a polytree, i.e. the undirected version of G is connected and acyclic*
- $\max\{\mathrm{cbs}(C) \mid C \in \mathrm{SCC}(G)\}$, *if G is cyclic but not strongly connected*
- $\min\{\mathrm{cbs}(G \setminus S) \cdot (|S| + 1) \mid S \subseteq V(G)\}$ *otherwise*

Intuitively, $\mathrm{cbs}(G)$ measures the *cyclicity* of G by decomposition into "easy to solve" subgraphs

Component-Boosted Backdoor Size cont.

How does this help us?

Component-Boosted Backdoor Size cont.

How does this help us?

Theorem

For every digraph G there exists a path preserving $\mathrm{cbs}(G)$-unfolding sequence.

Component-Boosted Backdoor Size cont.

How does this help us?

Theorem

For every digraph G there exists a path preserving $\mathrm{cbs}(G)$-unfolding sequence.

For the original point of interest this means that:

Theorem

For every answer set program Π, there exists an unfolding sequence $s \in \mathcal{A}(\Pi)^*$ such that

1. *the answer sets are preserved bijectively*
2. *the treewidth of $\mathrm{T}_{\mathcal{P}}$-Unfold(Π, s) is less or equal to $k \cdot \mathrm{cbs}(\mathrm{DEP}(\Pi))$, where k is the treewidth of Π.*

Scenarios

S1 Probabilistic reasoning: Computing probabilities for atoms of Problog programs

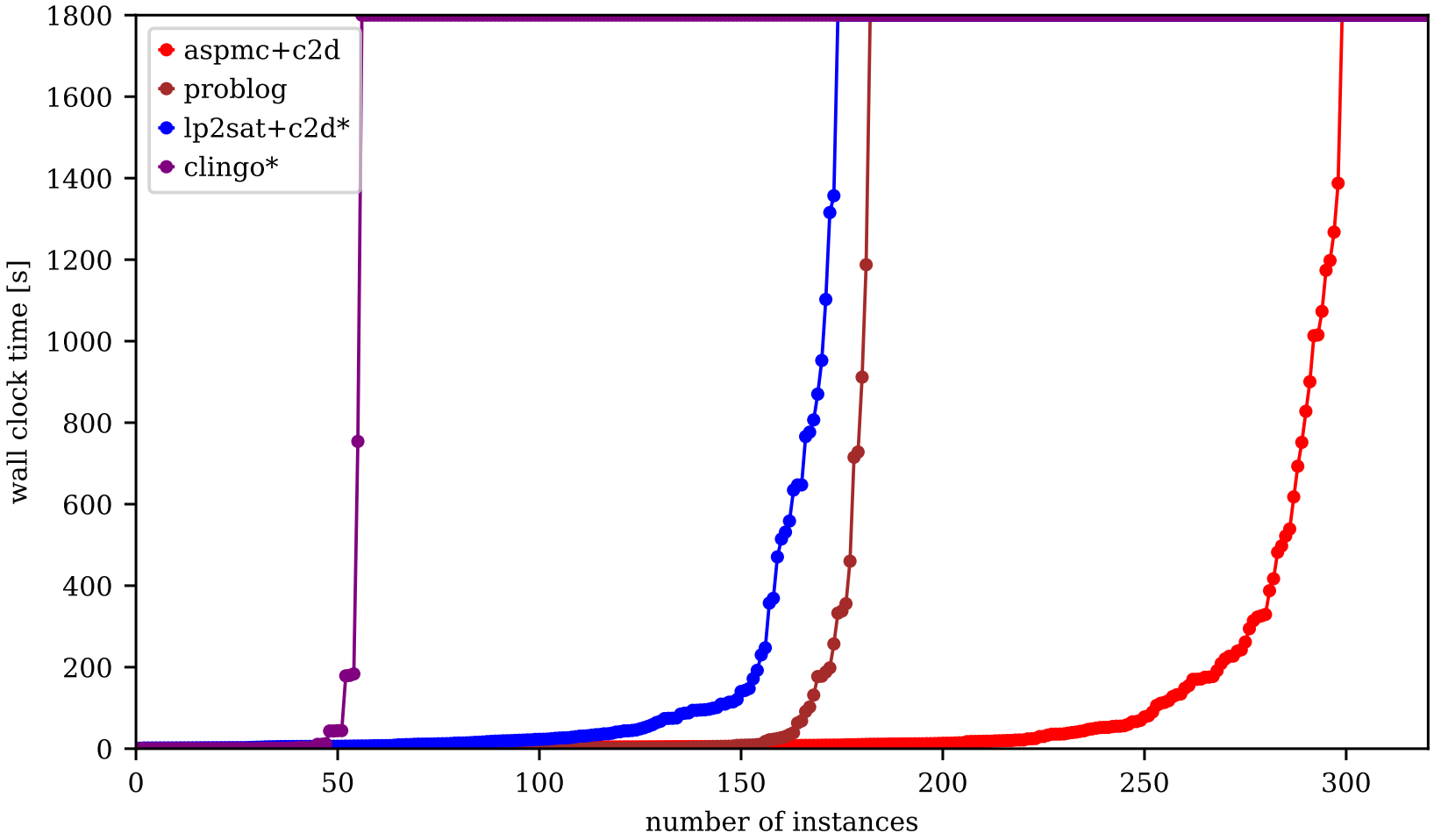
S2 Counting (small number of solutions on average): Counting the number of different paths between stations in public transport networks

S3 Counting (many solutions on average): Counting conflict-free extensions in abstract argumentation

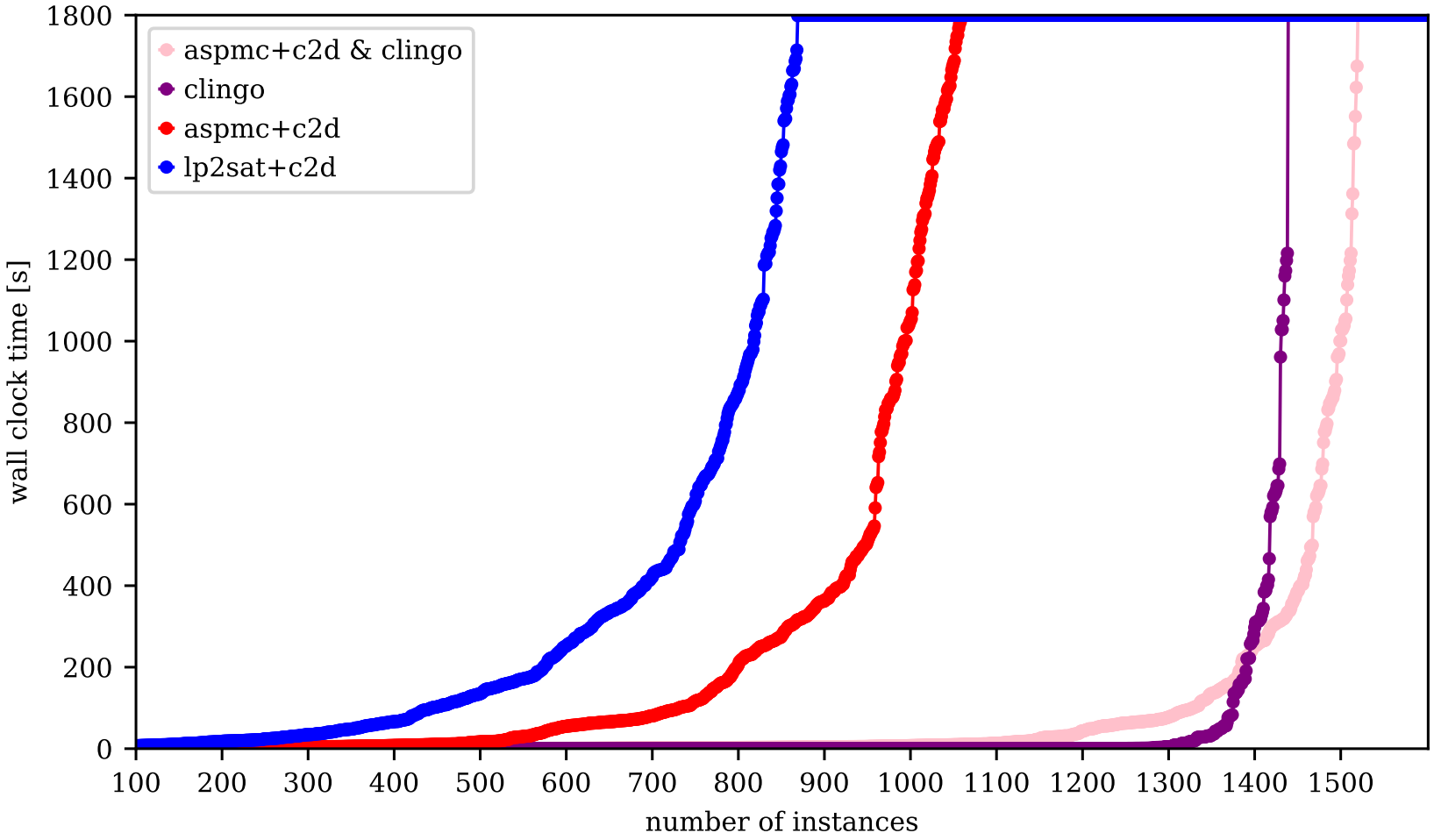
Solvers

- Problog, version 2.1.0.42, run with arguments "-k sdd"
- clingo, version 5.4.0, run with arguments "-q -n 0"
- lp2sat+c2d: cycle breaking due to [Bomanson, 2017] followed by compilation using c2d [Darwiche, 2004]
- aspmc+c2d: our cycle breaking followed by compilation using c2d [Darwiche, 2004]

Results S1


aspmc+c2d
problog
lp2sat+c2d*
clingo*
wall clock time [s]
number of instances
0
200
400
600
800
1000
1200
1400
1600
1800
0
50
100
150
200
250
300

Results S2


aspmc+c2d & clingo
clingo
aspmc+c2d
lp2sat+c2d
wall clock time [s]
number of instances
0
200
400
600
800
1000
1200
1400
1600
1800
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500

Results S3

solver configuration	$\sum$	tw ranges 0-300	300-600	>600	unique	time[h]
aspmc+c2d	**241**	**185**	**26**	**30**	**12**	**45.16**
lp2sat+c2d	182	182	0	0	0	73.85
clingo	144	97	21	26	2	94.78

Conclusions

- $\mathrm{cbs}(.)$ measures cyclicity of digraphs

Conclusions

- $\mathrm{cbs}(.)$ measures cyclicity of digraphs
- $\mathrm{T}_{\mathcal{P}}$-unfolding allows treewidth-aware cycle-breaking

Conclusions

- $\mathrm{cbs}(.)$ measures cyclicity of digraphs
- $\mathrm{T}_{\mathcal{P}}$-unfolding allows treewidth-aware cycle-breaking
- Our prototypical implementation partially outperforms other solvers

Chitta Baral, Michael Gelfond, and Nelson Rushton.
Probabilistic reasoning with answer sets.
Theory and Practice of Logic Programming, 9(1):57–144, 2009.

Jori Bomanson.
lp2normal - A normalization tool for extended logic programs.
In *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, pages 222–228. Springer, 2017.

Gerhard Brewka, James Delgrande, Javier Romero, and Torsten Schaub.
asprin: Customizing answer set preferences without a headache.
In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

Adnan Darwiche.

New advances in compiling CNF into decomposable negation normal form.
In *ECAI*, pages 328–332. IOS Press, 2004.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen.
Problog: A probabilistic prolog and its application in link discovery.
In *IJCAI*, volume 7, pages 2462–2467. Hyderabad, 2007.

Thomas Eiter and Rafael Kiesel.
Weighted lars for quantitative stream reasoning.
In *Proc. ECAI'20*, 2020.

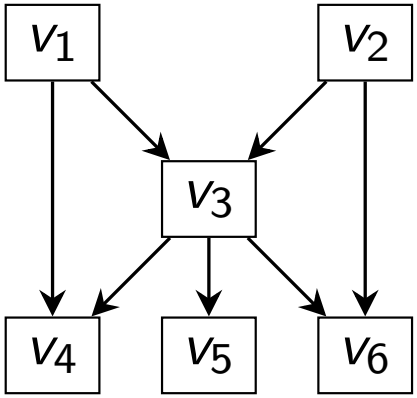
François Fages.
Consistency of clark's completion and existence of stable models.
Journal of Methods of logic in computer science, 1(1):51–60, 1994.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt.
An algebraic prolog for reasoning about possible worlds.
In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

Joohyung Lee and Zhun Yang.
Lpmln, weak constraints, and p-log.
In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

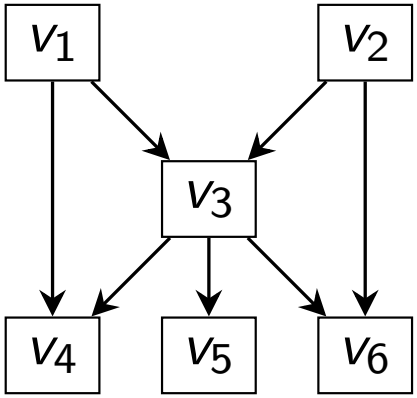
Case 1: DAG

- We can take any unfolding sequence s that is in topological order



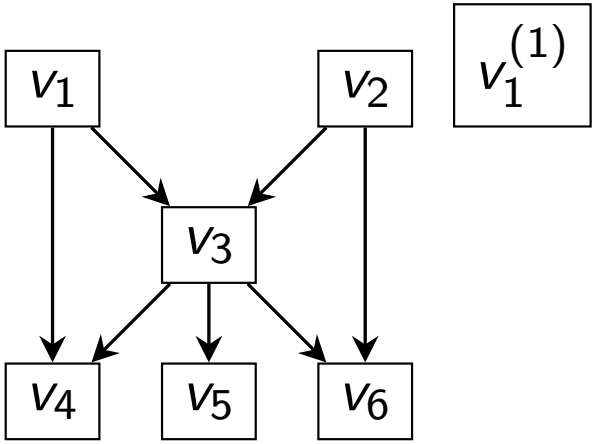
Case 1: DAG

- We can take any unfolding sequence s that is in topological order
- Such an s can be constructed by iteratively removing a vertex a without ancestors from G and appending it to s



Case 1: DAG

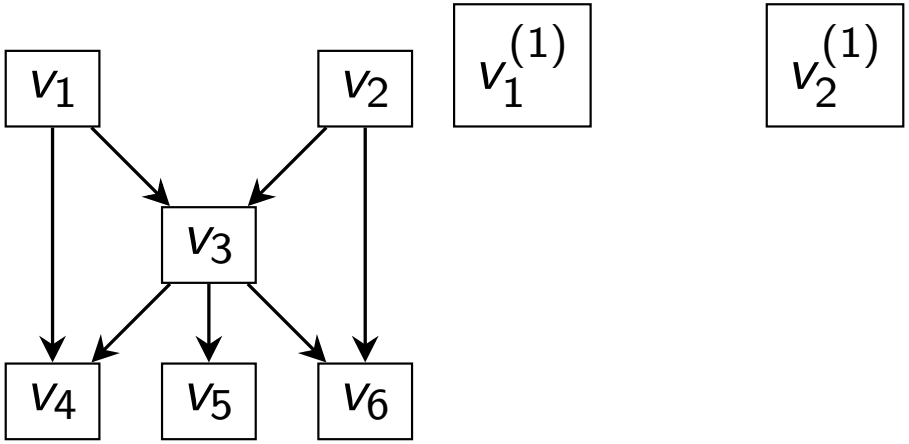
- We can take any unfolding sequence s that is in topological order
- Such an s can be constructed by iteratively removing a vertex a without ancestors from G and appending it to s



Use unfolding sequence $s = v_1$.

Case 1: DAG

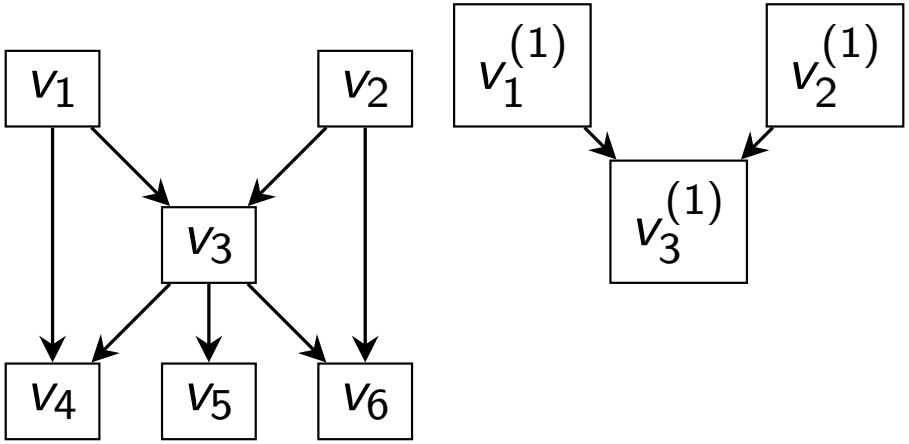
- We can take any unfolding sequence s that is in topological order
- Such an s can be constructed by iteratively removing a vertex a without ancestors from G and appending it to s



Use unfolding sequence $s = v_1 v_2$.

Case 1: DAG

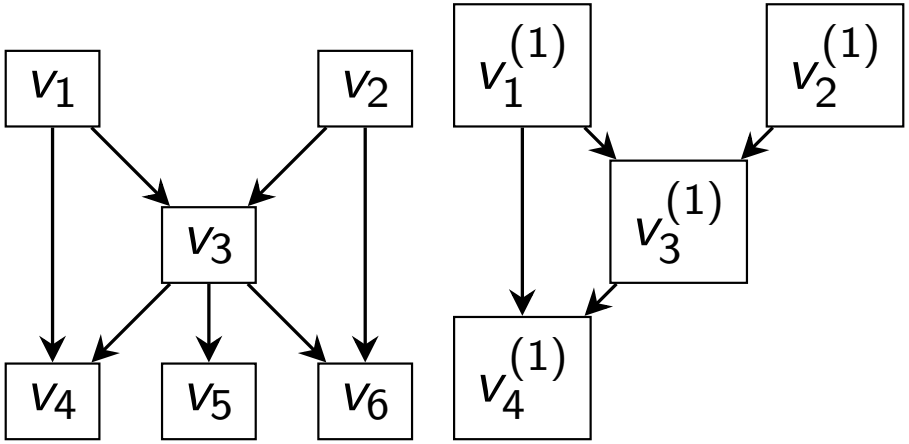
- We can take any unfolding sequence s that is in topological order
- Such an s can be constructed by iteratively removing a vertex a without ancestors from G and appending it to s



Use unfolding sequence $s = v_1 v_2 v_3$.

Case 1: DAG

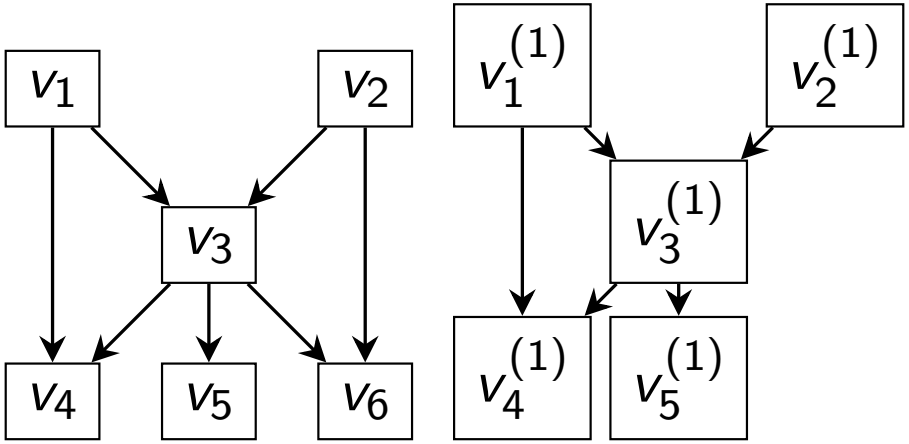
- We can take any unfolding sequence s that is in topological order
- Such an s can be constructed by iteratively removing a vertex a without ancestors from G and appending it to s



Use unfolding sequence $s = v_1 v_2 v_3 v_4$.

Case 1: DAG

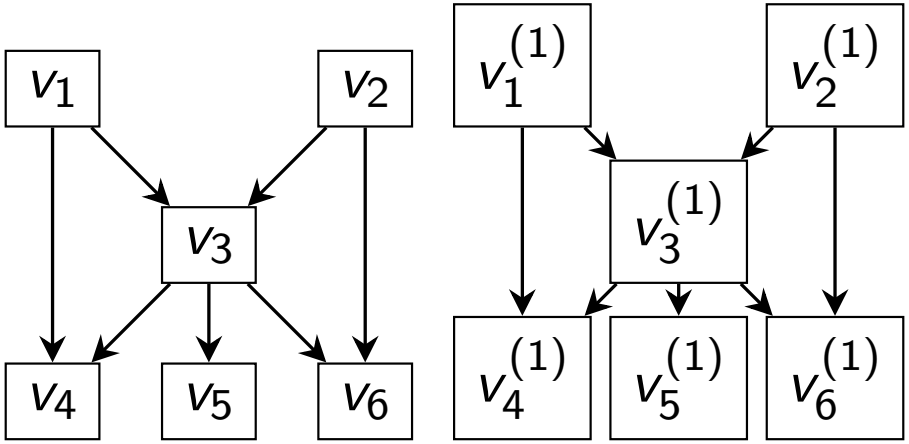
- We can take any unfolding sequence s that is in topological order
- Such an s can be constructed by iteratively removing a vertex a without ancestors from G and appending it to s



Use unfolding sequence $s = v_1 v_2 v_3 v_4 v_5$.

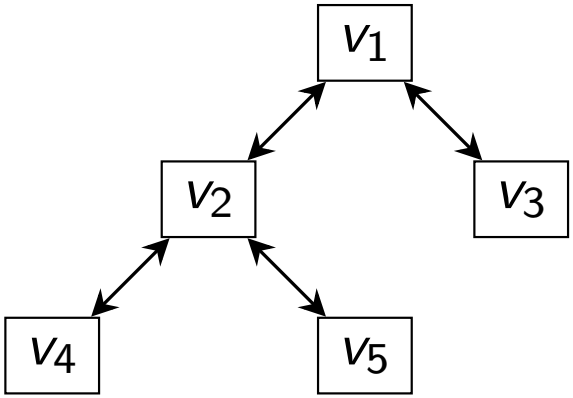
Case 1: DAG

- We can take any unfolding sequence s that is in topological order
- Such an s can be constructed by iteratively removing a vertex a without ancestors from G and appending it to s



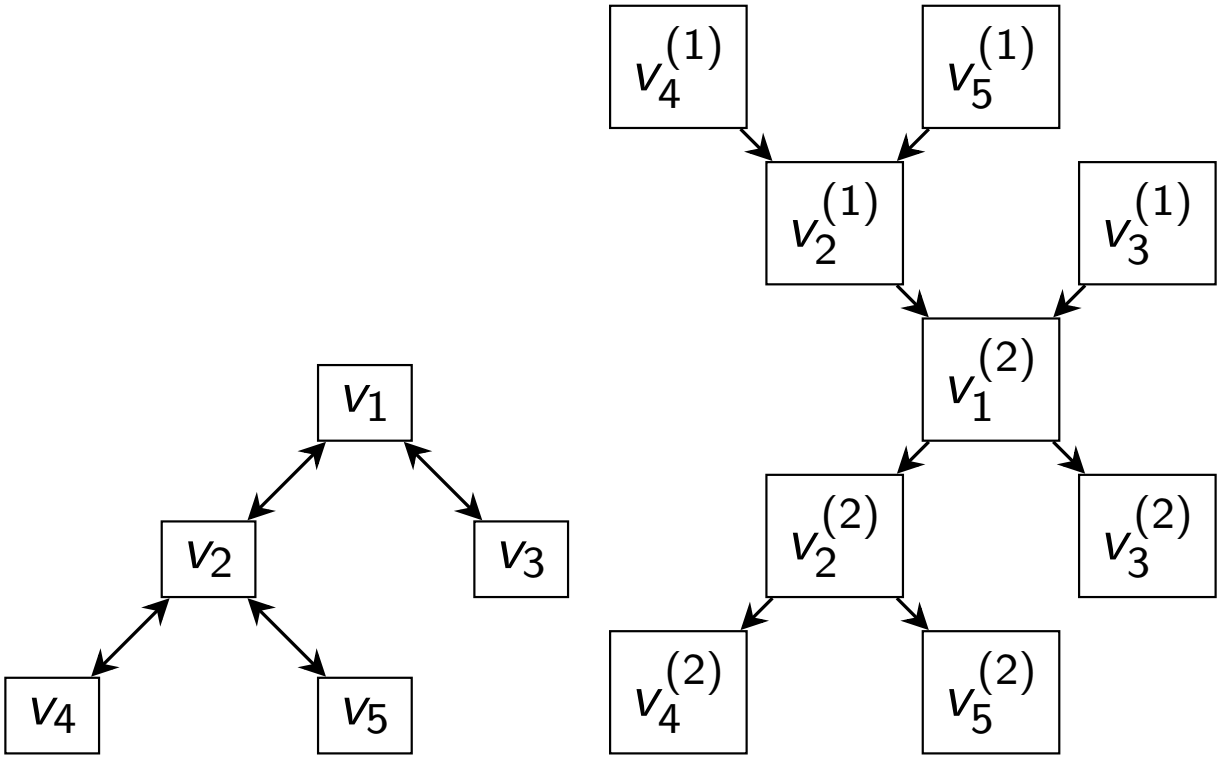
Use unfolding sequence $s = v_1 v_2 v_3 v_4 v_5 v_6$.

Case 2: Polytree



Use unfolding sequence $s = s_{post} s_{pre}$, where
$s_{post} = v_4 v_5 v_2 v_3 v_1$, $s_{pre} = v_3 v_2 v_5 v_4$.

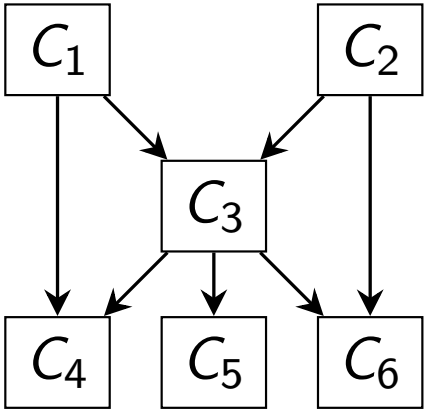
Case 2: Polytree



Use unfolding sequence $s = s_{post} s_{pre}$, where
$s_{post} = v_4 v_5 v_2 v_3 v_1$, $s_{pre} = v_3 v_2 v_5 v_4$.

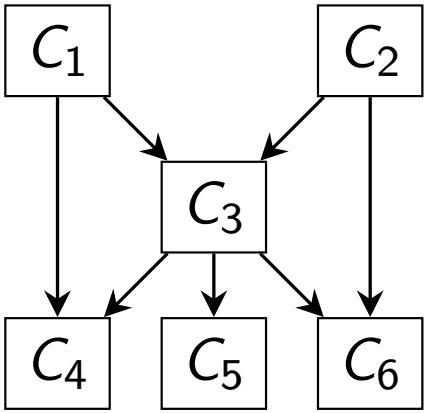
Case 3: Cyclic but not Strongly Connected

- We can handle each strongly connected component $C_i, i = 1, \ldots, k$ of G separately and combine them



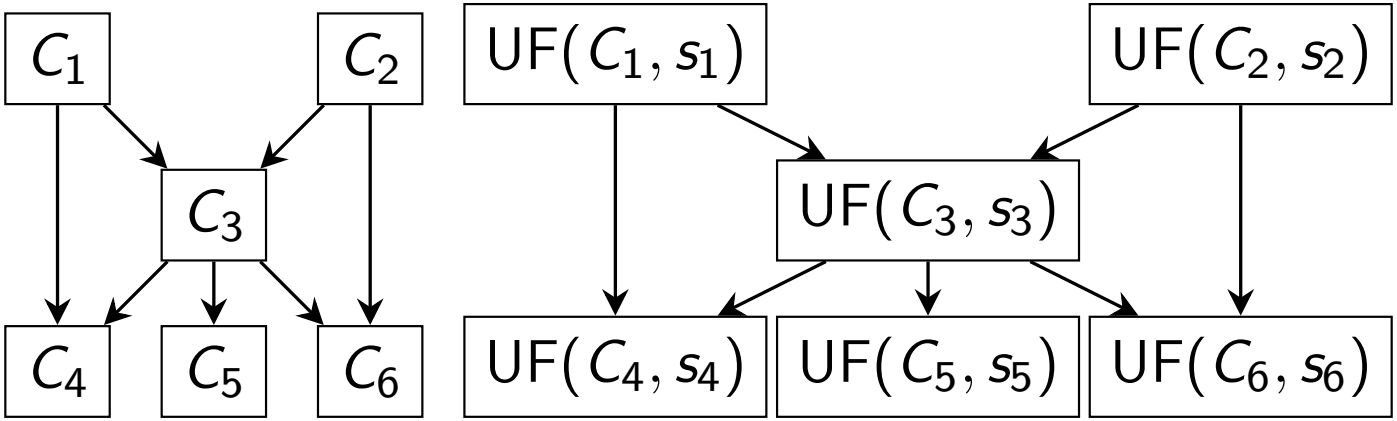
Case 3: Cyclic but not Strongly Connected

- We can handle each strongly connected component $C_i, i = 1, \ldots, k$ of G separately and combine them
- Take $s = s_{i_1} \ldots s_{i_k}$ (in topological order of the SCCs), where s_{i_j} is a path-preserving $\mathrm{cbs}(C_{i_j})$-unfolding sequence for C_{i_j}



Case 3: Cyclic but not Strongly Connected

- We can handle each strongly connected component $C_i, i = 1, \ldots, k$ of G separately and combine them
- Take $s = s_{i_1} \ldots s_{i_k}$ (in topological order of the SCCs), where s_{i_j} is a path-preserving $\mathrm{cbs}(C_{i_j})$-unfolding sequence for C_{i_j}



Use unfolding sequence $s = s_1 s_2 s_3 s_4 s_5 s_6$.

Case 4: Strongly Connected but not Polytree

- G is strongly connected but not a polytree

Case 4: Strongly Connected but not Polytree

- G is strongly connected but not a polytree
- Cut out $S = \{a_1, \ldots, a_n\} \subseteq V(G)$

Case 4: Strongly Connected but not Polytree

- G is strongly connected but not a polytree
- Cut out $S = \{a_1, \ldots, a_n\} \subseteq V(G)$
- Obtain a path-preserving $\mathrm{cbs}(G \setminus S)$-unfolding sequence s_r for $G \setminus S$

Case 4: Strongly Connected but not Polytree

- G is strongly connected but not a polytree
- Cut out $S = \{a_1, \ldots, a_n\} \subseteq V(G)$
- Obtain a path-preserving $\mathrm{cbs}(G \setminus S)$-unfolding sequence s_r for $G \setminus S$
- Use $s = (s_r s_S)^{|S|} s_r$, were $s_S = a_1 \ldots a_n$

Case 4: Strongly Connected but not Polytree

- G is strongly connected but not a polytree
- Cut out $S = \{a_1, \ldots, a_n\} \subseteq V(G)$
- Obtain a path-preserving $\mathrm{cbs}(G \setminus S)$-unfolding sequence s_r for $G \setminus S$
- Use $s = (s_r s_S)^{|S|} s_r$, were $s_S = a_1 \ldots a_n$
- s is a path-preserving $\mathrm{cbs}(G \setminus S) \cdot (|S| + 1)$-unfolding sequence for G

Case 4: Strongly Connected but not Polytree

- G is strongly connected but not a polytree
- Cut out $S = \{a_1, \ldots, a_n\} \subseteq V(G)$
- Obtain a path-preserving $\mathrm{cbs}(G \setminus S)$-unfolding sequence s_r for $G \setminus S$
- Use $s = (s_r s_S)^{|S|} s_r$, were $s_S = a_1 \ldots a_n$
- s is a path-preserving $\mathrm{cbs}(G \setminus S) \cdot (|S| + 1)$-unfolding sequence for G