Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning
Main Track

Treewidth-Aware Cycle Breaking for Algebraic Answer Set Counting

Thomas Eiter , Markus Hecher , Rafael Kiesel
Vienna University of Technology
thomas.eiter@tuwien.ac.at, markus.hecher@tuwien.ac.at, rafael.kiesel@tuwien.ac.at

Abstract

Probabilistic reasoning, parameter learning, and most probable explanation inference for answer set programming have recently received growing attention. They are only some of the problems that can be formulated as *Algebraic Answer Set Counting* (AASC) problems. The latter are however hard to solve, and efficient evaluation techniques are needed. Inspired by Vlasser et al.'s $\mathrm{T}_\mathcal{P}$-compilation (JAR, 2016), we introduce $\mathrm{T}_\mathcal{P}$-unfolding, which employs forward reasoning to break the cycles in the positive dependency graph of a program by *unfolding* them. $\mathrm{T}_\mathcal{P}$-unfolding is defined for any normal answer set program and unfolds programs with respect to *unfolding sequences*, which are akin to elimination orders in SAT-solving. Using "good" unfolding sequences, we can ensure that the increase of the *treewidth* of the unfolded program is small. Treewidth is a measure adhering to a program's tree-likeness, which gives performance guarantees for AASC. We give sufficient conditions for the existence of good unfolding sequences based on the novel notion of *component-boosted backdoor size*, which measures the cyclicity of the positive dependencies in a program. The experimental evaluation of a prototype implementation, the AASC solver aspmc, shows promising results.

1 Introduction

Recently, there has been a rising interest in reasoning problems for Answer Set Programming (ASP) that go beyond the classical reasoning tasks of entailment and consistency. For example, $\mathrm{LP}^{\mathrm{MLN}}$(Lee and Yang 2017), P-log (Baral, Gelfond, and Rushton 2009) and Problog (De Raedt, Kimmig, and Toivonen 2007) allow for probabilistic reasoning. Weight Constraints and more generally the asprin framework (Brewka et al. 2015) consider preferential reasoning over answer sets. Finally, algebraic Problog (Kimmig, Van den Broeck, and De Raedt 2011) and Weighted LARS (Eiter and Kiesel 2020) capture and generalize both of these ideas in *Algebraic Answer Set Counting (AASC)*, i.e. weighted answer set counting over semirings.

While the variety of frameworks that allow the specification of such problems is big, the corresponding choice of solvers is limited: the clingo solver (Gebser et al. 2014) only allows for answer set counting by enumeration, which becomes quickly infeasible once the number of answer sets exceeds a few millions. The dynASP2.5 solver (Fichte et al. 2017) uses the fact that answer set counting is fixed parameter tractable (FPT) in the treewidth of the program. However, already for normal ASP, the best known FPT algorithm for answer set counting is double exponential in the treewidth and therefore only feasible for instances with very low treewidth in practice. Last but not least, Problog reduces AASC to Algebraic Model Counting (AMC) (Kimmig, Van den Broeck, and De Raedt 2017), for which a wider variety of efficient solvers exist (Oztok and Darwiche 2015a; Sang, Beame, and Kautz 2005; Thurley 2006).

We focus on Problog as it is the most promising approach. Here, the basic strategy is the following. First one breaks the cyclic dependencies in the input program, obtaining a tight program, where the answer sets are the models of its Clark completion (Fages 1994). The CNF corresponding to the Clark completion can then be given to a solver that compiles it into an equivalent d-DNNF/SDD representation. On these representations, AMC is possible in linear time (Kimmig, Van den Broeck, and De Raedt 2017).

This approach allows for practically relevant instances to be solved (Vlasselaer et al. 2016). However, it has limitations and weaknesses. Firstly, Problog only accepts programs where negation is limited to a set of intensional atoms or stratified programs, rather than supporting the full syntax of normal ASP. Secondly, Problog only allows so called *factorized* algebraic measures, i.e., AASC instances where the weight of each individual answer set $\mathcal{I}$ is computed as a product of semiring values over the literals in $\mathcal{I}$ rather than allowing expressions where sums and products alternate arbitrarily as in wLARS (Eiter and Kiesel 2020). Last but not least, the way cycles in the positive dependency graph of the program are broken can have a strong negative effect on the treewidth of the instance. This in turn has a negative effect on performance guarantees for the compilation to d-DNNF/SDD: it is merely known that an instance of treewidth k has a d-DNNF/SDD representation of size at most 2^k. Thus an effective way of breaking cycles is needed.

Cycle breaking is an often and well studied topic in the ASP community. Among others, Hecher's (2020) translation from ASP to SAT guarantees that the treewidth of the SAT instance is $\mathcal{O}(k \log(l))$, where k is the original treewidth and l is the minimum of k and the size of the largest strongly connected component of the dependency graph of the program. Unfortunately, this translation does

not preserve models bijectively and is thus not helpful for answer set counting. In contrast, the cycle breaking used primarily in Problog (Mantadelis and Janssens 2010; Vlasselaer et al. 2016) and another (Janhunen 2004) preserve models bijectively but their treewidth bounds are not as strong.

In this work, we tackle all three of the aforementioned issues and make the following main contributions:

- Taking inspiration from $\mathrm{T}_\mathcal{P}$-compilation (Vlasselaer et al. 2016), we introduce $\mathrm{T}_\mathcal{P}$*-unfolding*, which uses the idea of forward reasoning to unfold a cyclic into an acyclic program along an *unfolding sequence* s. Such a sequence lists variables for unfolding steps, where variables can occur multiple times. Intuitively, unfolding sequences are to $\mathrm{T}_\mathcal{P}$-unfolding what elimination orders are to SAT-solving: as we will see, they have a big impact on the treewidth of the unfolded result and thus the performance guarantees of AASC.
- Furthermore, we introduce conditions that guarantee that $\mathrm{T}_\mathcal{P}$-unfolding along an unfolding sequence s returns a program whose answer sets are bijectively preserved. Using these results, we can show that every program of treewidth k can be translated into an acyclic program with treewidth at most $k \cdot \mathrm{cbs}(\mathrm{DEP}(\Pi))$, where $\mathrm{cbs}(\mathrm{DEP}(\Pi))$, is the *component-boosted backdoor size* of the dependency graph of Π; notably, $\mathrm{cbs}(.)$ is a novel parameter on directed graphs that combines backdoor sets and decomposability to measure the cyclicity of a directed graph.
- We show that AASC for any algebraic measure can be reduced in polynomial time to AASC for a *factorized* measure, which entails that an implementation for AASC over factorized measures is sufficient.
- Our prototype implementation aspmc performs AASC not only for Problog programs, but for all normal answer set programs. Our experimental results show that while clingo's extremely fast answer set enumeration is hard to beat when the number of answer sets is small, in the unavoidable case where a program has too many answer sets for clingo, aspmc outperforms both Problog and lp2sat (Janhunen 2004).

After preliminaries in Section 2, we give in Section 3 the definition of algebraic measures and show that they are equivalent in expressive power to *factorized* measures. Next, in Section 4, we introduce $\mathrm{T}_\mathcal{P}$-unfolding and give bounds on the increase of the treewidth caused by $\mathrm{T}_\mathcal{P}$-unfolding in terms of the component-boosted backdoor size of the dependency graph of a program. In Section 5, we describe our implementation and evaluate it in different relevant settings. Finally, in Section 6, we give conclusions and mention issues for further research.

2 Preliminaries

A (normal) *answer set program* Π is a finite set of *rules*

$$r = a \leftarrow b_1, \ldots, b_m, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_n,$$

where a, b_j, c_k are propositional variables. Given such a rule r, we let $H(r) = a$, $B^+(r) = \{b_1, \ldots, b_n\}$ and $B^-(r) = \{c_1, \ldots, c_n\}$. We slightly abuse notation and use $\leftarrow b_1, \ldots, b_m, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_n$ for $\bot \leftarrow b_1, \ldots, b_m, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_n, \mathrm{not}\, \bot$. Also, we allow *choice constraints* $\{a\} \leftarrow B^+(r), B^-(r)$ as a shorthand for the two rules $a \leftarrow B^+(r), B^-(r), \mathrm{not}\, na$ and $na \leftarrow B^+(r), B^-(r), \mathrm{not}\, a$, where na is a fresh propositional variable. We denote by $\mathcal{A}(\Pi)$ the set of propositional variables that occur in Π.

An *interpretation* $\mathcal{I} \subseteq \mathcal{A}(\Pi)$ satisfies Π, if for each rule $r \in \Pi$ it holds that $H(r) \in \mathcal{I}$ or there exists $a \in \mathcal{A}(\Pi)$ s.t. $a \in B^+(r) \setminus \mathcal{I}$ or $a \in B^-(r) \cap \mathcal{I}$. Furthermore, $\mathcal{I}$ is an *answer set* of Π if it is a $\subseteq$-minimal satisfying interpretation of the *reduct*[1] $\Pi^\mathcal{I} = \{r \in \Pi \mid B^+(r) \subseteq \mathcal{I}, B^-(r) \cap \mathcal{I} = \emptyset\}$. We denote the set of answer sets of a program Π by $\mathcal{AS}(\Pi)$.

Example 1 (Smokers)**.** *We consider the smokers program*[2]*, which is a standard example from probabilistic logic programming (De Raedt, Kimmig, and Toivonen 2007).*

$$\{\mathrm{stress}(X)\} \leftarrow \mathrm{person}(X)$$
$$\mathrm{smokes}(X) \leftarrow \mathrm{stress}(X)$$
$$\{\mathrm{inf}(X,Y)\} \leftarrow \mathrm{friend}(X,Y)$$
$$\mathrm{smokes}(Y) \leftarrow \mathrm{smokes}(X), \mathrm{inf}(X,Y), \mathrm{smokes}(Y)$$

This encodes that for all persons it is randomly determined whether they are stressed. If they are, they smoke. Furthermore, if one of their friends influences them, which is again random, and smokes, then they also smoke. We shorten $\mathrm{stress}(.)$ *and* $\mathrm{smokes}(.)$ *to* $\mathrm{st}(.)$ *and* $\mathrm{sm}(.)$*, respectively.*

Especially in Section 4 we will make use of graphs and digraphs, using the following notation. The vertex- and edge-sets of a (di)graph G are denoted by $V(G)$ and $E(G)$. For $V \subseteq V(G)$ we let $G[V]$ be the (di)graph obtained by removing all vertices not in V from $V(G)$ (i.e. $V(G[V]) = V(G) \cap V$) and removing all edges that are incident with a vertex not in V (i.e. $E(G[V]) = E(V) \cap V \times V$). Further, we define $G \setminus V$ as $G[V(G) \setminus V]$. The subgraph $C = G[V]$ is *strongly connected* if every vertex in C is reachable from any other vertex in C. We denote by $\mathrm{SCC}(G)$ the set of *strongly connected components (SCC)* of G, which are strongly connected subgraphs $G[V]$, where V is subset maximal.

The (positive) *dependency graph* $\mathrm{DEP}(\Pi)$ of a program Π is the digraph G with $V(G) = \mathcal{A}(\Pi)$ and $(b, a) \in E(G)$ if there is a rule $r \in \Pi$ such that $a \in H(r)$ and $b \in B^+(r)$. The *primal graph* $\mathrm{PRIM}(\Pi)$ of Π is the graph G with $V(G) = \mathcal{A}(\Pi)$ and $\{x, y\} \in E(G)$ if there is a rule $r \in \Pi$ such that $x, y \in \{H(r)\} \cup B^+(r) \cup B^-(r)$.

Example 2 (cont'd)**.** *Given the input data* $\mathrm{person}(i), i = 1, \ldots, 3$ *and* $\mathrm{friend}(i,j), i+1 \equiv j \mod 3$ *we can ground the smokers program to* Π_{sm}

$$\mathrm{sm}(X) \leftarrow \mathrm{st}(X)$$
$$\mathrm{sm}(X) \leftarrow \mathrm{inf}(Y,X), \mathrm{sm}(Y)$$
$$\{\mathrm{st}(x)\} \leftarrow \textit{for } x = 1, \ldots, 3$$
$$\{\mathrm{inf}(y,x)\} \leftarrow \textit{for } x + 1 = y \textit{ mod } 3$$

[1] All our results hold for both the FLP-reduct (Faber, Pfeifer, and Leone 2011) and GL-reduct (Gelfond and Lifschitz 1988)

[2] As usual the used schema rules with variables X, Y are implicitly $\forall$-quantified and their semantics is the grounding semantics.

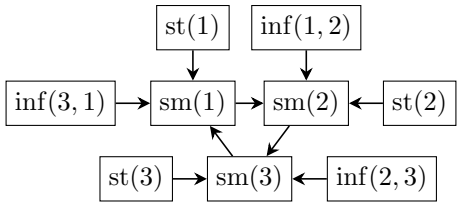

Figure 1: Dependency Graph of Π_{sm}.

The dependency graph of the result is given in Figure 1.

Next, we recall the definition of treewidth.

Definition 1 (Tree decomposition, Treewidth)**.** *Let G be a graph. Then a* tree decomposition *is a pair (T, χ), where T is a tree and χ is a labeling of $V(T)$ by subsets of $V(G)$ s.t.*

- *for all nodes $v \in V(G)$ there is $t \in V(T)$ s.t. $v \in \chi(t)$;*
- *for every edge $\{v_1, v_2\} \in V(E)$ there exists $t \in V(T)$ s.t. $v_1, v_2 \in \chi(t)$;*
- *for all nodes $v \in V(G)$ the set of nodes $\{t \in V(T) \mid v \in \chi(t)\}$ forms a (connected) subtree of T.*

The width of (T, χ) is $\max_{t \in V'} |\chi(t)| - 1$. The treewidth *of a graph is the minimal width of any of its tree decompositions. The treewidth of a program Π is the treewidth of* $\mathrm{PRIM}(\Pi)$.

Definition 2 (Semiring)**.** *A* semiring $\mathcal{R} = (R, \oplus, \otimes, e_\oplus, e_\otimes)$ *consists of a nonempty set R equipped with two binary operations $\oplus$ and $\otimes$, called addition and multiplication, where*

- *$(R, \oplus)$ is a commutative monoid with identity element $e_\oplus$,*
- *$(R, \otimes)$ is a monoid with identity element $e_\otimes$,*
- *multiplication left and right distributes over addition, and*
- *$e_\oplus$ annihilates R, i.e. $\forall r \in R : r \otimes e_\oplus = e_\oplus = e_\oplus \otimes r$.*

A semiring is commutative*, if $(R, \otimes)$ is commutative, and is* idempotent*, if $\forall r \in R : r \oplus r = r$.*

In the following, we restrict ourselves to commutative semirings. Some examples of well-known semirings are

- $\mathbb{F} = (\mathbb{F}, +, \cdot, 0, 1)$, for $\mathbb{F} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$ the semiring of the numbers in $\mathbb{F}$ with addition and multiplication,
- $\mathcal{R}_{\max} = (\mathbb{N} \cup \{-\infty\}, \max, +, -\infty, 0)$, the max-plus semiring,
- $\mathbb{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$, the Boolean semiring,
- $\mathcal{P} = ([0, 1], +, \cdot, 0, 1)$, the probability semiring.

For a more comprehensive list of semirings and applications see (Kimmig, Van den Broeck, and De Raedt 2017).

3 Algebraic Answer Set Counting

We now introduce ASP with algebraic measures, which amounts to the restriction of weighted LARS to ASP (Eiter and Kiesel 2020). This is sufficient, since it was shown that algebraic Problog, Problog, LP$^{\mathrm{MLN}}$and P-log can all be expressed in ASP with algebraic measures.

We use a variant of weighted logics (Droste and Gastin 2007) restricted to propositional formulas.

Definition 3 (Weighted Logic)**.** *Let $\mathcal{R} = (R, \oplus, \otimes, e_\oplus, e_\otimes)$ be a semiring. A* weighted formula *α over $\mathcal{R}$ is of the form*

$$\alpha ::= k \mid v \mid \neg v \mid \alpha + \alpha \mid \alpha * \alpha$$

where $k \in R$ and v is a propositional variable. The semantics of α w.r.t. an interpretation $\mathcal{I}$, denoted $[\![\alpha]\!]_\mathcal{R}(\mathcal{I})$, is

$$[\![k]\!]_\mathcal{R}(\mathcal{I}) = k,$$

$$[\![l]\!]_\mathcal{R}(\mathcal{I}) = \begin{cases} e_\otimes & l = v, v \in \mathcal{I} \text{ or } l = \neg v, v \notin \mathcal{I} \\ e_\oplus & \text{otherwise.} \end{cases},$$

$$[\![\alpha_1 + \alpha_2]\!]_\mathcal{R}(\mathcal{I}) = [\![\alpha_1]\!]_\mathcal{R}(\mathcal{I}) \oplus [\![\alpha_2]\!]_\mathcal{R}(\mathcal{I}),$$

$$[\![\alpha_1 * \alpha_2]\!]_\mathcal{R}(\mathcal{I}) = [\![\alpha_1]\!]_\mathcal{R}(\mathcal{I}) \otimes [\![\alpha_2]\!]_\mathcal{R}(\mathcal{I}).$$

Using weighted formulas, we define algebraic measures.

Definition 4 (Algebraic Measure)**.** *An* algebraic measure *$\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$ consists of an answer set program Π, a weighted formula α, and a semiring $\mathcal{R}$. The weight of an answer set $\mathcal{I} \in \mathcal{AS}(\Pi)$ under μ is*

$$\mu(\mathcal{I}) := [\![\alpha]\!]_\mathcal{R}(\mathcal{I}),$$

and a query *$\mu(a)$ for $a \in \mathcal{A}(\Pi)$ has result*

$$\mu(a) = \bigoplus_{\mathcal{I} \in \mathcal{AS}(\Pi), a \in \mathcal{I}} \mu(\mathcal{I}).$$

Example 3 (cont'd)**.** *As mentioned before, the smokers program is a typical example from the probabilistic domain. Using algebraic measures we can introduce probabilities. We define the measure $\mu_{sm} = \langle \Pi_{sm}, \alpha, \mathcal{P} \rangle$, where*

$$\alpha = \Pi_{i=1}^{3} \mathrm{st}(i) * 0.4 + \neg \mathrm{st}(i) * 0.6$$
$$* \Pi_{i,j=1,2,3,i+1 \equiv j \bmod 3} \inf(i, j) * 0.3 + \neg \inf(i, j) * 0.7.$$

This means that the probability of a person being stressed is 0.4 and the probability that a person influences their friend is 0.3. The answer set $\mathcal{I} = \{\mathrm{st}(1), \mathrm{sm}(1)\}$ has weight $\mu_{sm}(\mathcal{I}) = 0.4 \cdot 0.6^2 \cdot 0.7^3$. The query $\mu(\mathrm{sm}(1))$ corresponds to the probability that $\mathrm{sm}(1)$ holds. To evaluate it we need to perform AASC, i.e. sum up the probabilities of all answer sets s.t. $\mathrm{sm}(1)$ holds.

Following the conventions of (Belle and De Raedt 2020), we also introduce factorized measures.

Definition 5 (Factorized Measure)**.** *Let $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$ be an algebraic measure and $F \subseteq \mathcal{A}(\Pi)$. Then μ is* factorized *w.r.t. F, if there is a weight function $\beta : F \cup \{\neg f \mid f \in F\} \to R$ s.t. for all $\mathcal{I} \in \mathcal{AS}(\Pi)$ it holds that*

$$\mu(\mathcal{I}) = \bigotimes_{f \in F \cap \mathcal{I}} \beta(f) \otimes \bigotimes_{f \in F \setminus \mathcal{I}} \beta(\neg f).$$

Example 4 (cont'd)**.** *The measure μ_{sm} is factorized over $\mathrm{st}(i), i = 1, \ldots, 3$ and $\inf(i, j), i + 1 \equiv j \bmod 3$, by letting $\beta(\mathrm{st}(i)) = 0.4, \beta(\neg \mathrm{st}(i)) = 0.6$ and $\beta(\inf(i, j)) = 0.3, \beta(\neg \mathrm{st}(i, j)) = 0.7$.*

Example 5 (Non-factorized)**.** *For an example of a non-factorized measure, consider the measure $\mu_w = \langle \Pi, a + b + (-1 * \neg a * \neg b), \mathbb{Z} \rangle$. It has value 2 if both a and b hold, 1 if one them holds and -1 otherwise. This measure is not factorized over $F = \{a, b\}$ as there are no values $\beta(a), \beta(b), \beta(\neg a), \beta(\neg b) \in \mathbb{Z}$, s.t.*

$$\beta(a) \cdot \beta(b) = 2 \qquad \beta(a) \cdot \beta(\neg b) = 1$$
$$\beta(\neg a) \cdot \beta(b) = 1 \qquad \beta(\neg a) \cdot \beta(\neg b) = -1$$

Note that there are also no such values in $\mathbb{R}$.

While not every algebraic measure is factorized, there always exists a factorized measure that preserves weights of queries. To establish this, we need some notation for sets of indexed subformulas of a weighted formula α.

Definition 6 (Subformulas)**.** *Let α a weighted formula. Then $\mathcal{S}(\alpha)$ is the set of pairs (i, β), where β is a subformula of α indexed by position-string $i \in \{0,1\}^*$. That is:*

- *For $\alpha \in \{p(\vec{x}), \neg p(\vec{x}), k\}$ we let $\mathcal{S}(\alpha) = \{(\epsilon, \alpha)\}$.*
- *For $\alpha \in \{\alpha_1 + \alpha_2, \alpha_1 * \alpha_2\}$ we let $\mathcal{S}(\alpha) = \{(\epsilon, \alpha)\} \cup \{(0r, \beta) \mid (r, \beta) \in \mathcal{S}(\alpha_1)\} \cup \{(1r, \beta) \mid (r, \beta) \in \mathcal{S}(\alpha_2)\}$.*

Theorem 7 (Factorization)**.** *Let $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$ be an algebraic measure. Then we can construct a factorized algebraic measure $\mu' = \langle \Pi', \alpha', \mathcal{R} \rangle$ s.t. for $a \in \mathcal{A}(\Pi) : \mu(a) = \mu'(a)$ in linear time.*

Proof (sketch). Intuitively, we implicitly imply the distributive law. For this, we introduce a new atom α_i for every subformula $(i, \beta) \in S(\alpha)$, which is true if the value of the subformula β at index i is included in the current product. To implement this, we let $\Pi' = \Pi \cup \Pi_{\text{root}} \cup \Pi_* \cup \Pi_+ \cup \Pi_{\text{leaf}}$. Here, $\Pi_{root} = \{\leftarrow \text{not } \alpha_0\}$ ensures that the value of the formula at the root is included.

$$\Pi_* = \left\{ \begin{array}{c} \alpha_i \leftarrow \alpha_{i.0}, \alpha_{i.1} \\ \leftarrow \alpha_{i.0}, \text{not } \alpha_{i.1} \\ \leftarrow \text{not } \alpha_{i.0}, \alpha_{i.1} \end{array} \middle| \; (i, \beta_1 * \beta_2) \in \mathcal{S}(\alpha) \right\}$$

ensures that a subformula that uses multiplication is only included if both subformulas are included. Further, we ensure that either both or none of the subformulas are included.

$$\Pi_+ = \left\{ \begin{array}{c} \alpha_i \leftarrow \alpha_{i.1} \\ \alpha_i \leftarrow \alpha_{i.0} \\ \leftarrow \alpha_{i.0}, \alpha_{i.1} \end{array} \middle| \; (i, \beta_1 + \beta_2) \in \mathcal{S}(\alpha) \right\}$$

ensures that a subformula that uses addition is only included if one of the subformulas is included. Further, we ensure that at most one of the subformulas is included.

$$\begin{aligned} \Pi_{\text{leaf}} = &\{\{\alpha_i\} \leftarrow a \mid (i, a) \in \mathcal{S}(\alpha)\} \cup \\ &\{\{\alpha_i\} \leftarrow \text{not } a \mid (i, \neg a) \in \mathcal{S}(\alpha)\} \cup \\ &\{\{\alpha_i\} \leftarrow \mid (i, k) \in \mathcal{S}(\alpha), k \in R\}. \end{aligned}$$

Formally, we define $\alpha' = \Pi_{(i,k) \in \mathcal{S}(\alpha), k \in R} \alpha_i * k + \neg \alpha_i$. Then $\mu' = \langle \Pi', \alpha', \mathcal{R} \rangle$ is factorized, by choosing $F = \{\alpha_i \mid (i, k) \in \mathcal{S}(\alpha), k \in R\}$ and $\beta(\alpha_i) = k, \beta(\neg \alpha_i) = e_{\otimes}$.

Further, along a similar line of reasoning as in (Eiter and Kiesel 2021) it follows that $\mu(\Pi) = \mu'(\Pi')$. □

Hence, any AASC instance given as an algebraic measure can be reduced to AASC for a factorized measure. Thus, we can focus on factorized measures for which AASC can be performed on a tractable circuit representation like d-DNNFs or SDDs.

4 Cycle Breaking

In order to obtain a tractable circuit representation for a program Π using a standard knowledge compiler, we translate Π into a propositional formula ϕ such that the answer sets of Π are in a one-to-one correspondence with the models of ϕ. To this end, we will prove the following as our main result of this section:

Theorem 8. *For any factorized measure $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$, there exists $\mu' = \langle \Pi', \alpha, \mathcal{R} \rangle$ with an acyclic program Π' s.t.*

- *for all $a \in \mathcal{A}(\Pi)$ it holds that $\mu(a) = \mu'(a)$,*
- *the treewidth of Π' is bounded by $k \cdot \text{cbs}(\text{DEP}(\Pi))$, where k is the treewidth of Π.*

There are multiple aspects to consider in order to appreciate the usefulness of this Theorem. Firstly, it is important that Π' is acyclic, as it is well-known that when the dependency graph $\text{DEP}(\Pi)$ of Π is acyclic, then $\text{Clark}(\Pi)$, the Clark-completion (Fages 1994) of Π, is a propositional formula whose models exactly correspond to the answer sets of Π. The removal of cycles may significantly increase the treewidth, which in turn is assumed to have negative effects on the efficiency of model counting. However, according to the above Theorem, we can bound the treewidth overhead using $\text{cbs}(\text{DEP}(()\Pi))$, i.e., a parameter that only depends on the dependency graph of Π. Here, $cbs(.)$ is the *component-boosted backdoor size*, a new digraph-parameter that measures cyclicity.

4.1 $\text{T}_\mathcal{P}$-Unfolding

A current state of the art approach to AASC is $\text{T}_\mathcal{P}$-compilation (Vlasselaer et al. 2016). Intuitively, the idea is to use forward reasoning to iteratively compile an SDD that captures the truth of variables in a program in terms of probabilistic input variables. We also use forward reasoning, but in a different way: namely, to break the cycles in the dependency graph of a program by *unfolding* it along paths that forward reasoning could take. Considering $\text{T}_\mathcal{P}$-unfolding allows us to unfold all normal answer set programs while $\text{T}_\mathcal{P}$-compilation is only defined for Problog programs and normal answer set programs with stratified negation.

Furthermore, whereas $\text{T}_\mathcal{P}$-compilation relies on performing equivalence checks in order to check whether it is faithful, we unfold programs along an *unfolding sequence* $s \in \mathcal{A}(\Pi)^*$ and give conditions that ensure that the unfolding is faithful.

Our procedure is described in Algorithm 1. Intuitively, where the immediate consequence operator $\text{T}_\mathcal{P}$ (Van Emden and Kowalski 1976) checks if an atom a follows from previously derived atoms, $\text{T}_\mathcal{P}$-unfolding introduces copies of all rules that derive a from previously considered atoms. For this, we iterate over the unfolding sequence considering at each step the variable s_i (line 4). As the head atom of the rule-copies we take a new copy $s_i^{cnt(s_i)+1}$ or the original atom s_i depending on whether s_i is the last occurrence of s_i in s (see lines 5-8). The positive body atoms are replaced by the last copy made of them (line 10) and the negative atoms in $B^-(r)$ are left as they are. After copying all rules with s_i in the head, we update the last copy of s_i and increase the counter storing the number of copies (lines 12, 13). The output of $\text{T}_\mathcal{P}$-$\text{UNFOLD}(\Pi, s)$ is always an acyclic program. We further want to preserve models bijectively.

Definition 9 (Faithfulness)**.** *We say $s \in \mathcal{A}(\Pi)^*$ is* faithful *(for Π), if:*

- $|\mathcal{AS}(\Pi)| = |\mathcal{AS}(\text{T}_\mathcal{P}\text{-UNFOLD}(\Pi, s))|$ *and*
- $\mathcal{AS}(\Pi) = \{\mathcal{I} \cap \mathcal{A}(\Pi) \mid \mathcal{I} \in \mathcal{AS}(\text{T}_\mathcal{P}\text{-UNFOLD}(\Pi, s))\}$.

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'.

1: last = $\{a \mapsto \bot \mid a \in \mathcal{A}(\Pi)\}$
2: cnt = $\{a \mapsto 0 \mid a \in \mathcal{A}(\Pi)\}$
3: $\Pi' = \{r \in \Pi, H(r) = \bot\}$
4: **for** $i = 1, \ldots, \mathrm{len}(s)$ **do**
5: **if** ISLASTOCCURRENCE(s_i, i, s) **then**
6: head = s_i
7: **else**
8: head = $s_i^{\mathrm{cnt}(s_i)+1}$
9: **for** $r \in \Pi, s_i = H(r)$ **do**
10: B^+_{new} = {last$(b) \mid b \in B^+(r)$}
11: $\Pi' = \Pi' \cup$ {head $\leftarrow B^+_{new}, B^-(r)$}
12: last(s_i) = head
13: cnt(s_i) = cnt(s_i) + 1
14: **return** Π'

Faithfulness is important for us, as it guarantees us that we can perform AASC over the unfolded program without changing the result.

Lemma 10. *Let $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$ be a measure, s a faithful unfolding sequence and $\mu' = \langle \mathrm{T}_\mathcal{P}\text{-UNFOLD}(\Pi, s), \alpha, \mathcal{R} \rangle$. Then for all $a \in \mathcal{A}(\Pi)$, $\mu(a) = \mu'(a)$.*

Example 6 (cont'd)**.** *We compute $\mathrm{T}_\mathcal{P}$-UNFOLD(Π_{sm}, s) using the unfolding sequence*[3] $s = \mathrm{st}(1)\mathrm{nst}(1)\ldots\mathrm{st}(3)\mathrm{nst}(3)$ $\mathrm{inf}(3,1)\mathrm{ninf}(3,1)\ldots\mathrm{inf}(2,3)\mathrm{ninf}(2,3)\mathrm{sm}(1)\mathrm{sm}(2)\mathrm{sm}(3)$ $\mathrm{sm}(1)\mathrm{sm}(2)$ *and obtain:*

$$\{\mathrm{st}(x)\} \leftarrow \textit{ for } x = 1, \ldots, 3$$
$$\{\mathrm{inf}(y, x)\} \leftarrow \textit{ for } x + 1 = y \textit{ mod } 3$$

$$\mathrm{sm}(1)^1 \leftarrow \mathrm{st}(1) \qquad \mathrm{sm}(1)^1 \leftarrow \mathrm{inf}(3,1), \bot$$
$$\mathrm{sm}(2)^1 \leftarrow \mathrm{st}(2) \qquad \mathrm{sm}(2)^1 \leftarrow \mathrm{inf}(1,2), \mathrm{sm}(1)^1$$
$$\mathrm{sm}(3) \leftarrow \mathrm{st}(3) \qquad \mathrm{sm}(3) \leftarrow \mathrm{inf}(2,3), \mathrm{sm}(2)^1$$
$$\mathrm{sm}(1) \leftarrow \mathrm{st}(1) \qquad \mathrm{sm}(1) \leftarrow \mathrm{inf}(3,1), \mathrm{sm}(3)$$
$$\mathrm{sm}(2) \leftarrow \mathrm{st}(2) \qquad \mathrm{sm}(2) \leftarrow \mathrm{inf}(1,2), \mathrm{sm}(1)$$

We observe that s faithful for Π_{sm}.

In general, it is enough to iterate over all the variables $n = |\mathcal{A}(\Pi)| + 1$ times, since every derivation in Π can only take n steps. However, as we have seen in the previous example, it can suffice to use much fewer steps. This is because the number of times a variable needs to be considered in an unfolding sequence depends on the positive dependencies it takes part in: e.g. $\mathrm{st}(i), i = 1, \ldots, 3$ does not positively depend on any variable and can therefore be considered once before all other variables and never again afterwards.

We give a sufficient condition for faithfulness that abstracts away the actual program Π and is instead based on a structural property of the *digraph unfolding* of $\mathrm{DEP}(\Pi)$.

Definition 11 (Digraph Unfolding)**.** *Let G be a digraph and $s \in V(G)^*$ be an unfolding sequence. Then the unfolding $\mathrm{UF}(G, s)$ of G w.r.t. s is the digraph U such that*

[3]Recall that choice constraints $\{a\} \leftarrow$ are a shorthand for $a \leftarrow \mathrm{not}\ na$ and $na \leftarrow \mathrm{not}\ a$.

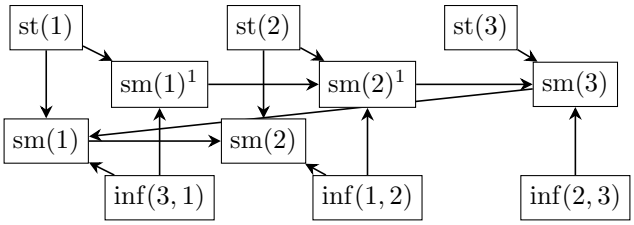

Figure 2: Dependency Graph of $\mathrm{T}_\mathcal{P}$-UNFOLD(Π_{sm}, s).

- $V(U) = \{a^i \mid 1 \leq i \leq |\{s_j \mid s_j = a\}|\}$,
- $(b^i, a^j) \in E(U)$ *if* $(b, a) \in E(G)$, $s_k = a$ *for some* k *and* $s_1, \ldots, s_k$ *contains* j *many* a*'s and* $i > 0$ *many* b*'s.*

The idea is that the digraph unfolding of $\mathrm{DEP}(\Pi)$ w.r.t. $s \in \mathcal{A}(\Pi)^*$ is the dependency graph of the unfolded program $\mathrm{T}_\mathcal{P}$-UNFOLD(Π, s). Therefore, the vertices are the copies of the atoms (see lines 6,8) and there is an edge (b^i, a^j) if b^i is the last copy of an atom that is used to derive a^j, i.e. the j-th occurrence of a in s (see lines 10,11). Formally:

Lemma 12. *Let Π be an answer set program and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. Then* $\mathrm{UF}(\mathrm{DEP}(\Pi), s) = \mathrm{DEP}(\mathrm{T}_\mathcal{P}\text{-UNFOLD}(\Pi, s))$ *(when identifying a with $a^{cnt(a)}$, where $cnt(a) = |\{i \mid s_i = a\}|$, for each $a \in \mathcal{A}(\Pi)$).*

Example 7 (cont'd)**.** *The dependency graph of $\mathrm{T}_\mathcal{P}$-UNFOLD(Π_{sm}, s) is given in Figure 2. As we can see it is acyclic and corresponds to* $\mathrm{UF}(\mathrm{DEP}(\Pi_{sm}), s)$ *as expected.*

Using the above Lemma, we can provide a sufficient condition for faithfulness.

Theorem 13. *Let Π be an answer set program and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. 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* $\mathrm{UF}(\mathrm{DEP}(\Pi), s)$*, then s is faithful.*

Proof (sketch). Let Π be some answer set program, s an unfolding sequence that satisfies the precondition of the theorem and $\mathcal{I} \subseteq \mathcal{A}(\Pi)$. We know that, regardless of s, the reduct $\mathrm{T}_\mathcal{P}$-UNFOLD$(\Pi, s)^{\mathcal{I}_{ext}}$ for $\mathcal{I}_{ext} \cap \mathcal{A}(\Pi) = \mathcal{I}$ is $\mathrm{T}_\mathcal{P}$-UNFOLD$(\Pi, s)^\mathcal{I}$ because the rules that are added in line 12 use the original negative body $B^-(r)$, which only uses atoms from $\mathcal{A}(\Pi)$. Therefore, we can consider $\mathrm{T}_\mathcal{P}$-UNFOLD$(\Pi, s)^\mathcal{I}$, which has a unique minimal model. We see that if there is an answer set $\mathcal{I}_{ext}$ of $\mathrm{T}_\mathcal{P}$-UNFOLD(Π, s) that is equal to $\mathcal{I}$ on $\mathcal{A}(\Pi)$, then it is the only such answer set.

By the same argument we see that taking the reduct w.r.t. $\mathcal{I}$ and $\mathrm{T}_\mathcal{P}$-compilation commute: $\mathrm{T}_\mathcal{P}$-UNFOLD$(\Pi, s)^\mathcal{I} = \mathrm{T}_\mathcal{P}$-UNFOLD$(\Pi^\mathcal{I}, s)$. Since $\mathcal{I}$ is an answer set iff it is a minimal model of the reduct $\Pi^\mathcal{I}$, it remains to show that $a \in \mathcal{A}(\Pi)$ is derivable from $\Pi^\mathcal{I}$ iff it is derivable from $\mathrm{T}_\mathcal{P}$-UNFOLD$(\Pi^\mathcal{I}, s)$. Since both programs are positive, a is derivable iff it has an SLD tree. However, we know that s preserves all simple paths and since the paths in every SLD tree correspond to paths in $\mathrm{DEP}(\Pi)$, we know there exists a corresponding SLD-tree in $\mathrm{T}_\mathcal{P}$-UNFOLD$(\Pi^\mathcal{I}, s)$. □

Note that a similar theorem can be proven for $\mathrm{T}_\mathcal{P}$-compilation, which reaches a fixed point iff s is faithful.

We introduce the component-boosted backdoor size of a digraph, which intuitively measures cyclicity. When this parameter is low, there exists a faithful unfolding sequence in which each variable occurs only a few times.

Definition 14 ($\mathrm{cbs}(G)$)**.** *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*

As the name suggests, *cbs* adds *component-boosting* to a specific variant of *backdoors*, which in general have already been considered in the context of ASP (Fichte and Szeider 2015). The most related notion of a *backdoor* for a digraph G is a vertex set S, such that $G \setminus S$ is a polytree, polyforest or dag, where the *backdoor size* of G is the minimum size of a backdoor for G plus 1. Note that component-boosted backdoor size additionally takes into account that $G \setminus S$ may consist of separate SCCs that can be handled recursively and is therefore always at most as high as backdoor size.

Example 8 (cont'd)**.** *Consider the dependency graph* $\mathrm{DEP}(\Pi_{sm})$ *in Figure 1. It is strongly connected and not a polytree, therefore* $\mathrm{cbs}(\mathrm{DEP}(\Pi_{sm}))$ *is given by* $\min\{\mathrm{cbs}(\mathrm{DEP}(\Pi_{sm}) \setminus S)(|S|+1) \mid S \subseteq V(\mathrm{DEP}(\Pi_{sm}))\}$*. We see that if we take away any $S_i = \{\mathrm{sm}(i)\}, i = 1, \ldots, 3$, then* $\mathrm{DEP}(\Pi_{sm}) \setminus S_i$ *is acyclic. Therefore, we see that* $\mathrm{cbs}(\mathrm{DEP}(\Pi_{sm})) \leq \mathrm{cbs}(\mathrm{DEP}(\Pi_{sm}) \setminus S_i) \cdot (|S_i| + 1) = 2$*.*

In this example backdoor size and component-boosted backdoor size align. However, for larger, more complex graphs $cbs(.)$ can be much smaller than backdoor size.

The following is the main result of this section.

Theorem 8. *For every answer set program Π, there exists a faithful unfolding sequence $s \in \mathcal{A}(\Pi)^*$ such that the treewidth of $\mathrm{T}_\mathcal{P}$-$\textsc{Unfold}(\Pi, s)$ is less or equal to $k \cdot \mathrm{cbs}(\mathrm{DEP}(\Pi))$, where k is the treewidth of Π.*

The theorem is based on the following fact.

Lemma 15. *Let Π be an answer set program with treewidth k and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. If every variable $a \in \mathcal{A}(\Pi)$ only occurs m times in s, then the treewidth of $\mathrm{T}_\mathcal{P}$-$\textsc{Unfold}(\Pi, s)$ is less or equal to $k \cdot m$.*

Proof (sketch). We know that during unfolding we introduce at most $m - 1$ copies $a_1, \ldots, a_{m-1}$ of a variable $a \in \mathcal{A}(\Pi)$. Now, let (T, χ) be a tree decomposition for $\mathrm{PRIM}(\Pi)$ of width k. Then (T, χ'), where

$$\chi'(t) = \chi(t) \cup \{a_j \mid 1 \leq j \leq m-1, a \in \chi(t)\}$$

is a tree decomposition of $\mathrm{T}_\mathcal{P}$-$\textsc{Unfold}(\Pi, s)$. Further, $|\chi'(t)| \leq |\chi(t)| \cdot m \leq k \cdot m$. □

We remark that the converse of this Lemma does not hold.

Motivated by this Lemma and Theorem 13, we say that s is a *path-preserving m-unfolding sequence* (for digraph G), if for every simple (directed) path $\pi = (a_1, \ldots, a_n)$ in G there is a (directed) path $\pi_c = (a_1^{c_1}, \ldots, a_n^{c_n})$ in $\mathrm{UF}(G, s)$ and every variable $a \in V(G)$ occurs at most m times in s.

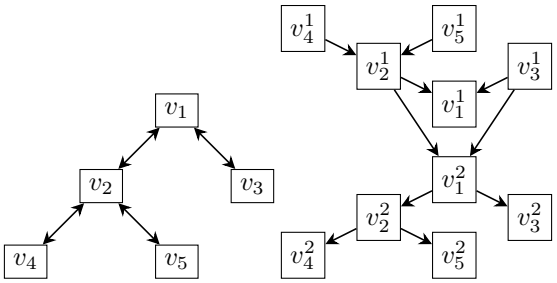

Figure 3: A polytree G (left) and the unfolding $\mathrm{UF}(G, s_{post}s_{pre})$ for $s_{post} = v_4v_5v_2v_3v_1$, $s_{pre} = v_1v_3v_2v_5v_4$ (right).

We prove Theorem 8 by showing that there exists a path-preserving $\mathrm{cbs}(\mathrm{DEP}(\Pi))$-unfolding sequence for $\mathrm{DEP}(\Pi)$ using structural induction on the definition of $\mathrm{cbs}(.)$.

Lemma 16. *Let G be an acyclic digraph (DAG). Then there exists a path-preserving 1-unfolding sequence.*

Proof. Let s be an unfolding sequence, where every $a \in V(G)$ occurs exactly once and which obeys a topological ordering of G. Then $\mathrm{UF}(G, s)$ is equal to G (modulo variable renaming) and therefore path-preserving. □

Next, we consider the second case, where G is a polytree.

Lemma 17. *For every polytree G there exists a path-preserving 2-unfolding sequence s.*

Proof. We know that G is a polytree. We take G^{tree}, the corresponding undirected graph, which is a tree with some arbitrarily chosen root. Let $s_{post}, s_{pre} \in V(G)^*$ be sequences such that every vertex occurs in s_{post} and s_{pre} after all its descendants and ancestors in G^{tree}, respectively. Then $s_{post}s_{pre}$, the concatenation of s_{post} and s_{pre}, is a path-preserving 2-unfolding sequence of G, as depicted in Figure 3. □

In the third case, which is the first recursive one, we assume G is cyclic but not strongly connected. Here, we divide the problem into one subproblem for each SCC of G and obtain a global solution by combining the solutions for the subproblems.

Lemma 18. *Let G be a cyclic but not strongly connected digraph, and for each $C \in \mathrm{SCC}(G)$ let $s_C \in V(C)^*$ be a path-preserving m_C-unfolding sequence for C. Then some path-preserving $\max_{C \in \mathrm{SCC}(G)} m_C$-unfolding sequence for G exists.*

Proof. Let G^{con} be the condensation of G, i.e. $V(G^{con}) = \mathrm{SCC}(G)$ and $(C, C') \in E(G^{con})$ if there exist $v \in V(C), v' \in V(C')$ such that $(v, v') \in E(G)$. Since G^{con} is acyclic we can assume a topological order $(C_1, \ldots, C_n)$ of G^{con} to be given. Consider, $s = s_{C_1} \ldots s_{C_n}$, the concatenation of the unfolding sequences for the SCCs in the chosen topological order. It is a path-preserving unfolding

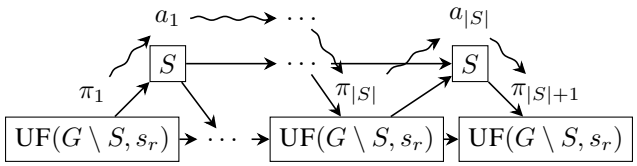

Figure 4: Sketch of UF(G, s) and a path π through it, for the second recursive case, as in the proof of Lemma 19.

sequence for G since for every directed simple path in G that contains $a, b \in V(G)$ it holds that if $a \in C_i$ and $b \in C_j$ such that $i < j$ then a must occur after b. Therefore, as the sequences s_{C_i} per component are path-preserving, we know that the whole sequence is path-preserving. Further, since $V(C_i) \cap V(C_j) = \emptyset$ for $i \neq j$ and $s_{C_i} \in V(C_i)^*$, it is clear that the maximum number of times a vertex $a \in V(G)$ occurs in s is bounded by $\max_{C \in \mathrm{SCC}(G)} m_C$. □

Last but not least, we consider the second recursive case. Here, G is strongly connected but not a polytree. We remove a set $S \subseteq V(G)$ of "problematic" vertices such that the component-boosted backdoor size of the rest, i.e., $\mathrm{cbs}(G \setminus S)$, is small and handle S and $G \setminus S$ separately.

Lemma 19. *Let G be a strongly connected digraph, $S \subseteq V(G)$ and $s_r \in V(G \setminus S)^*$ a path-preserving m_r-unfolding sequence. Then there exists a path-preserving $m_r(|S|+1)$-unfolding sequence for G.*

Proof. Let $S = \{a_1, \ldots, s_{|S|}\}$. We define $s_S = a_1 \ldots a_{|S|}$ and $s = (s_r s_S)^{|S|} s_r$, i.e., $s \in V(G)^*$ is the sequence obtained by concatenating the sequence $s_r s_S$ with itself $|S|-1$ times and then concatenating s_r. Then s is a $m_r(|S|+1)$-unfolding sequence, as every $a \in S$ occurs exactly $|S| \leq m_r(|S|+1)$ times, and every $a \in V(G \setminus S)$ occurs at most m_r times in s_r and at most $m_r(|S|+1)$ times in general.

Furthermore, s is path-preserving: every simple directed path π in G uses $k \leq |S|$ vertices from S and thus $\pi = \pi_1, a_{i_1}, \pi_2, a_{i_2}, \ldots, a_{i_k}, \pi_k$, where π_i is a simple directed path in $G \setminus S$. Consider Figure 4, which sketches UF(G, s) and the path π. As s_r is path-preserving for $G \setminus S$, we know that we can walk π_1 in UF$(G \setminus S, s_r)$, then go to $a_1 \in S$, walk the path π_2 in UF$(G \setminus S, s_r)$ and so forth. □

5 Implementation & Experiments

Our prototypical implementation aspmc[4], written in Python3, currently allows for two main settings: (algebraic) *Problog mode*, where the accepted inputs are Problog programs Π and aspmc computes the answers to probabilistic (resp. algebraic) queries in Π, and *ASP mode*, which accepts any normal program Π and computes the number of answer sets of Π. Our implementation aspmc proceeds as follows:

1. We parse the input program Π, using the Python API of clingo (Gebser et al. 2014) for normal programs and our own parser for Problog programs.
2. We compute an initial treewidth upper bound for the primal graph of Π using htd (Abseher, Musliu, and Woltran 2017), which is a highly efficient tool for computing tree decompositions via heuristics. On all our benchmark instances this required at most 60 seconds.
3. Next, we start the actual cycle breaking. While $\mathrm{cbs}(.)$ is hard to compute exactly, we can compute a backdoor, i.e. a subset of the vertices such that the dependency graph of the program without them is a polytree or polyforest. These backdoors are estimated by running clingo on an ASP encoding of the problem and using the best result found after 30s. Using the backdoor computed by clingo we extract an unfolding sequence using Lemma 16 to 19 and apply $\mathrm{T}_\mathcal{P}$-unfolding.
4. The resulting cycle-free program Π' is then converted to a propositional formula $\mathrm{Clark}(\Pi')$, by using a treewidth-aware Clark completion due to (Hecher 2020).
5. Last but not least we perform weighted/algebraic model counting (Problog mode) or model counting (ASP mode) on $\mathrm{Clark}(\Pi')$ using a knowledge compiler/model counter.

Benchmark Setting. In order to evaluate the performance of aspmc, we considered the following 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

Scenario S1 aims at improving the evaluation of Problog programs by employing $\mathrm{T}_\mathcal{P}$-unfolding. Scenario S2 tackles an important #P-complete problem (Valiant 1979). Scenario S3 is motivated by the recent decision of the biennial competition in abstract argumentation (Mailly et al. 2021) to also include dedicated counting tracks.

For each of these three scenarios, we state corresponding hypotheses that we want to study and verify in the course of our experimental analysis.

H1 Treewidth-aware cycle breaking as presented in this work has a significant impact on solving reasoning tasks based on cyclic logic programs.

H2 For real-world settings as in counting problems originating on networks of practical relevance, like public transport networks, $\mathrm{T}_\mathcal{P}$-unfolding is beneficial as long as the treewidth is not excessively large.

H3.1 Counting by enumeration is outperformed by counting via compilation for instances with many solutions.

H3.2 Even in an acyclic setting, like S3, our treewidth-aware version of Clark completion improves counting performance.

Compared Solvers. In our experiments, we mainly compare the performance of the following configurations.

- Problog: we use version 2.1.0.42 of Problog run with the arguments "-k sdd", which proved to be the best overall.

[4] available at github.com/raki123/aspmc (open source).

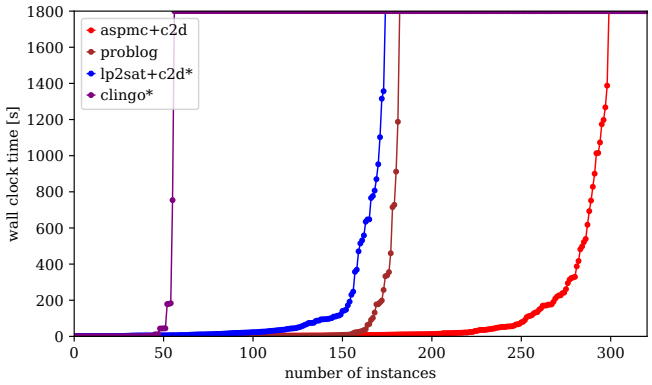

Figure 5: Runtime of the compared solver configurations for Scenario S1. The x-axis shows the number of instances and the y-axis depicts runtimes in seconds. Instances are ordered for each configuration individually in ascending order of their runtimes. The plot legend lists solvers from best to worst ("right" to "left" in the plot).

solver configuration	∑	tw ranges 0-3	4-6	>6	unique	time[h]
aspmc+c2d	**299**	**184**	**113**	2	**116**	**133.44**
Problog	182	168	12	2	0	187.41
lp2sat+c2d*	174	167	3	**4**	2	194.42
clingo*	56	56	0	0	0	248.92

Table 1: Detailed results for Scenario S1: "Σ" is the number of solved instances in total; "tw ranges" gives more fine grained insights showing the number of solved instances grouped by treewidth upper bounds; "unique" refers to the number of instances solved only by that configuration. Finally, "time[h]" is the total runtime over all instances in hours (unsolved instances count as timeout, i.e., 1800s). Configurations marked with an asterisk ("*") refer to counting solutions instead of probabilistic reasoning.

- clingo: this configuration is based on version 5.4.0 and we used arguments "-q -n 0" in order to count answer sets.
- lp2sat+c2d: instances are translated (Bomanson 2017) to CNFs by lp2normal 2.18 in combination with lp2atomic 1.17 and lp2sat 1.24, in order to preserve answer set counts. Then, the answer sets are counted via counting satisfying assignments of the resulting formula, by using c2d version 2.2 (Darwiche 2004). Overall, c2d with arguments "-smooth_all -reduce -in_memory -count" seemed to provide the best results on our instances, even faster results than, e.g., minic2d (Oztok and Darwiche 2015b).
- aspmc+c2d: this configuration uses aspmc to break the cycles and performs AASC on a tractable circuit representation of the constructed CNF, which is obtained via c2d 2.2 with arguments "-smooth_all -reduce".

For scenario S1, we mainly compare aspmc+c2d with Problog, since both solvers are able to solve the reasoning tasks of S1. However, for S1 we still depict results of lp2sat+c2d and clingo when counting answer sets, since counting can be seen as the basis for solving these reasoning tasks. Scenarios S2 and S3 focus on counting answer sets, where we compare aspmc+c2d with clingo and lp2sat+c2d.

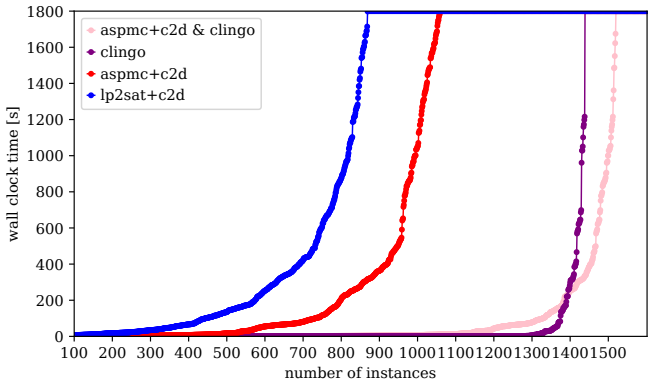

Figure 6: Runtime of the compared solver configuration for Scenario S2. The plot is of the same type as in Figure 5.

solver configuration	∑	tw ranges 0-5	6-10	>10	unique	time[h]
aspmc+c2d & clingo	**1520**	**767**	**355**	**398**	0	**824.08**
clingo	1439	686	**355**	**398**	**4**	846.07
aspmc+c2d	1059	**767**	224	68	3	1083.11
lp2sat+c2d	870	609	202	59	1	1182.7

Table 2: Detailed results over instances of Scenario S2.

Benchmark Instances. The instances used in Scenario S1 are the benchmarks from (Tsamoura, Gutiérrez-Basulto, and Kimmig 2020) and were kindly provided to us by the authors via personal communication. They adhere to typical benchmark domains consisting of 490 instances of the standard smoker's example, 50 instances of the gene's problem (Ourfali et al. 2007) and 63 web knowledge base instances (Davis and Domingos 2009). For Scenario S2 we used real-world graphs, more specifically, public transport networks of several transport agencies over the world (Fichte 2016) as instances. They were also used in the so-called PACE challenge competitions 2016 and 2017 (Dell et al. 2017). In total these instances amount to 561 graph networks and 2553 subgraphs with a focus on different transportation modes. For each instance, we assume the station with the smallest and largest index to be the start and end stations, respectively.

For Scenario S3 we took the instances of the abstract argumentation competition (Mailly et al. 2021) from 2019, since at the time of submission the instances for 2021 were not available. The ASP encoding for conflict-free extensions was taken from the ASPARTIX suite (Dvorák et al. 2020).

Note that all our instances[5] as well as raw data are available at github.com/raki123/aspmc/tree/results/results.

Benchmark Platform. All our solvers ran on a cluster consisting of 12 nodes. Each node of the cluster is equipped with two Intel Xeon E5-2650 CPUs, where each of these 12 physical cores runs at 2.2 GHz clock speed and has access to 256 GB shared RAM. Results are gathered on Ubuntu 16.04.1 LTS powered on kernel 4.4.0-139 with hyperthreading disabled using version 3.7.6 of Python3.

[5]With the exception of the Problog instances (Tsamoura, Gutiérrez-Basulto, and Kimmig 2020) that were kindly provided to us in personal communication and are not yet published.

Experimental Results & Discussion. In our evaluation, we mainly compare total wall clock time and number of solved instances. Concerning benchmark limits, we consider a timeout of 1800 seconds and an 8 GB RAM limit per instance and solver.

Figure 5 shows a plot over all instances of Scenario S1, which indicates that aspmc+c2d solves more instances faster than any of the other configurations that we benchmarked. The detailed results in Table 1 indicate that cycle breaking works well for probabilistic reasoning, thus confirming H1.

Notably, for the instances from S1, we also benchmarked aspmc+c2d and Problog for AASC over other semirings like the gradient semiring from (Eisner 2002) that is used for parameter learning in probabilistic settings. Here, we observed almost identical results, implying a benefit not only for probabilistic reasoning but for AASC over general semirings.

For Scenario S2 we observed that since the number of solutions is on the smaller side, clingo can outperform compilation-based solvers by enumerating all solutions, as indicated in Figure 6. Note, however, that aspmc+c2d still outperforms lp2sat+c2d, the other compilation-based approach. Furthermore, Table 2 reveals that there are instances of small treewidth that aspmc+c2d solved but clingo did not solve. Therefore, for instances of small treewidth aspmc+c2d is still the configuration one would choose, thereby confirming H2. Indeed, the decomposer htd allows us to estimate whether an instance has a small treewidth upper bound in a matter of seconds. Consequently, we can construct an interesting portfolio, referred to by "aspmc+c2d & clingo", that chooses aspmc+c2d for instances of treewidth smaller than six and clingo for the rest. This configuration is also included both in Figure 6 and Table 2. While it does not provide the best performance on every instance, it solves about 80 instances more in less total time over all instances than the second best configuration clingo.

Finally, Scenario S3 supports H3.1, the hypothesis that compilation outperforms enumeration on instances with many solutions, as both lp2sat+c2d and aspmc+c2d solve more instances than clingo. On top of that, we confirm H3.2 by the results of Table 3: even though the instances are cycle free, thus requiring no cycle-breaking but only translation to CNF using Clark completion, aspmc+c2d solves almost 60 more instances than the second best configuration lp2sat+c2d. The improvement is especially visible in Figure 7, which depicts a so-called scatter plot comparing the performance of instances one by one. Indeed, aspmc+c2d outperforms lp2sat+c2d on almost every instance.

By comparing results among different scenarios, we observe that clingo is extremely fast at enumerating solutions. Thus, on instances with not that many solutions, clingo is faster than any of the compilation-based approaches we considered. On the other hand, when there are more solutions, the compilation-based approaches lp2sat+c2d, aspmc+c2d and Problog (where applicable) solve significantly more instances. Thus, our results suggest that overall, one is forced to choose between (a) high performance on instances with few solutions at the cost of not being able to obtain a result otherwise, and (b) an overhead cost on instances with few solutions, which in turn allows for more instances to be solved, given that their treewidth does not become too large. This is especially visible in Figure 5 and Table 1, which correspond to our targeted application domain of probabilistic reasoning. Here, clingo finishes fastest on about 30 instances but solves less than one-fifth of the instances aspmc solves.

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

Table 3: Detailed results over instances of Scenario S3.

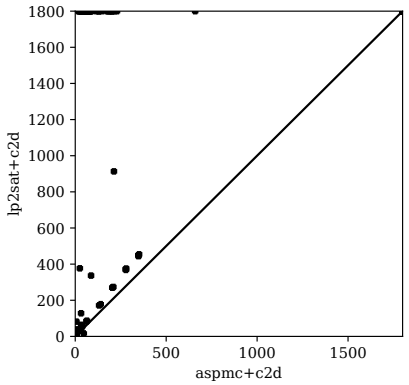

Figure 7: Scatter plot of aspmc+c2d and lp2sat+c2d for Scenario S3, comparing the runtime of instances one by one.

6 Conclusion and Future Work

We have presented $\mathrm{T}_{\mathcal{P}}$-unfolding, which uses the idea of forward reasoning to unfold any normal program into an acyclic program. The combination with unfolding sequences, which guide the unfolding, ensures that we can obtain a treewidth upper bound for the unfolded program. For any program Π, the treewidth can be at most $\mathrm{cbs}(\mathrm{DEP}(\Pi))$ times bigger, where $\mathrm{cbs}(.)$ is a novel parameter that measures the cyclicity of directed graphs. The bound on the treewidth provides us in turn with worst case guarantees for the knowledge compilation step in AASC. Our experimental evaluation of the prototype implementation aspmc shows that this idea does not only provide interesting theoretical results but provides a significant speedup on standard Problog benchmarks compared to other solvers. In our other benchmark settings, we saw that while clingo's enumeration based approach for answer set counting is hard to beat for instances with few answer sets, it can be very beneficial to use aspmc instead of clingo or lp2sat when this is not the case. This applies even to programs that are already acyclic due to the treewidth-aware Clark completion employed by aspmc.

For future work it would be interesting to find better approximation algorithms for $\mathrm{cbs}(.)$, to consider the combination of $\mathrm{T}_{\mathcal{P}}$-compilation with unfolding sequences, and to minimize input programs before cycle breaking.

Acknowledgements

This work has been supported by the Austrian Science Fund (FWF), Grants P32830, Y698 and W1255-N23, as well as the Vienna Science and Technology Fund, Grant WWTF ICT19-065. Hecher is also affiliated with the University of Potsdam, Germany; part of the work was carried out while he was visiting the Simons Institute for the Theory of Computing.

References

Abseher, M.; Musliu, N.; and Woltran, S. 2017. htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, volume 10335 of *Lecture Notes in Computer Science*, 376–386. Springer.

Baral, C.; Gelfond, M.; and Rushton, N. 2009. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming* 9(1):57–144.

Belle, V., and De Raedt, L. 2020. Semiring programming: A semantic framework for generalized sum product problems. *International Journal of Approximate Reasoning* 126:181–201.

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

Brewka, G.; Delgrande, J.; Romero, J.; and Schaub, T. 2015. asprin: Customizing answer set preferences without a headache. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

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

Davis, J., and Domingos, P. 2009. Deep transfer via second-order markov logic. In *Proceedings of the 26th annual international conference on machine learning*, 217–224.

De Raedt, L.; Kimmig, A.; and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, 2462–2467. Hyderabad.

Dell, H.; Komusiewicz, C.; Talmon, N.; and Weller, M. 2017. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In *International Symposium on Parameterized and Exact Computation (IPEC)*, Leibniz International Proceedings in Informatics, 30:1—30:13. Dagstuhl.

Droste, M., and Gastin, P. 2007. Weighted automata and weighted logics. *Theoretical Computer Science* 380(1):69.

Dvorák, W.; Gaggl, S. A.; Rapberger, A.; Wallner, J. P.; and Woltran, S. 2020. The ASPARTIX system suite. In Prakken, H.; Bistarelli, S.; Santini, F.; and Taticchi, C., eds., *Computational Models of Argument (COMMA)*, volume 326 of *Frontiers in Artificial Intelligence and Applications*, 461–462. IOS Press.

Eisner, J. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 1–8.

Eiter, T., and Kiesel, R. 2020. Weighted LARS for quantitative stream reasoning. In *ECAI 2020*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, 729–736. IOS Press.

Eiter, T., and Kiesel, R. 2021. On the complexity of sum-of-products problems over semirings. In *AAAI 2021*, 6304–6311. AAAI Press.

Faber, W.; Pfeifer, G.; and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence* 175(1):278–298.

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

Fichte, J. K., and Szeider, S. 2015. Backdoors to tractable answer set programming. *Artif. Intell.* 220:64–103.

Fichte, J. K.; Hecher, M.; Morak, M.; and Woltran, S. 2017. Dynasp2. 5: Dynamic programming on tree decompositions in action. In *International Symposium on Parameterized and Exact Computation (IPEC)*, volume 89 of *LIPIcs*, 17:1–17:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Fichte, J. K. 2016. daajoe/gtfs2graphs – a GTFS transit feed to graph format converter. https://github.com/daajoe/gtfs2graphs at commit 219944893f874b365de1ed87fc265fd5d19d5972.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo= ASP+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, 1070–1080.

Hecher, M. 2020. Treewidth-aware Reductions of Normal ASP to SAT - Is Normal ASP Harder than SAT after All? In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, 485–495.

Janhunen, T. 2004. Representing normal programs with clauses. In *ECAI*, volume 16, 358. Citeseer.

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

Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2017. Algebraic model counting. *Journal of Applied Logic* 22:46–62.

Lee, J., and Yang, Z. 2017. LPMLN, weak constraints, and P-log. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Mailly, J.-G.; Lonca, E.; Lagniez, J.-M.; and Rossit, J. 2021. International Competition on Computational Models of Argumentation 2021. Website: http://argumentationcompetition.org/2021/rules.html.

Mantadelis, T., and Janssens, G. 2010. Dedicated tabling for a probabilistic setting. In *Technical Communications of the 26th International Conference on Logic Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Ourfali, O.; Shlomi, T.; Ideker, T.; Ruppin, E.; and Sharan, R. 2007. Spine: a framework for signaling-regulatory

pathway inference from cause-effect experiments. *Bioinformatics* 23(13):i359–i366.

Oztok, U., and Darwiche, A. 2015a. A top-down compiler for sentential decision diagrams. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Oztok, U., and Darwiche, A. 2015b. A top-down compiler for sentential decision diagrams. In *IJCAI*, 3141–3148. AAAI Press.

Sang, T.; Beame, P.; and Kautz, H. A. 2005. Performing bayesian inference by weighted model counting. In *AAAI*, volume 5, 475–481.

Thurley, M. 2006. sharpSAT–counting models with advanced component caching and implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*, 424–429. Springer.

Tsamoura, E.; Gutiérrez-Basulto, V.; and Kimmig, A. 2020. Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs. In *Conference on Artificial Intelligence (AAAI)*, 10284–10291. AAAI Press.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM Journal of Computing* 8(3):410–421.

Van Emden, M. H., and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)* 23(4):733–742.

Vlasselaer, J.; Van den Broeck, G.; Kimmig, A.; Meert, W.; and De Raedt, L. 2016. Tp-compilation for inference in probabilistic logic programs. *International Journal of Approximate Reasoning* 78:15–32.