Artificial Intelligence 330 (2024) 104109

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint

aspmc: New frontiers of algebraic answer set counting

Thomas Eiter [a], Markus Hecher [b], Rafael Kiesel [a,*]

[a] *Institute of Computational Logic, TU Wien, Favoritenstrasse 9-11, Vienna, 1040, Austria*
[b] *Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, 32 Vassar St., Cambridge, MA 02139, USA*

ARTICLE INFO

Keywords:
Answer set programming
Algebraic answer set counting
Treewidth
Cycle breaking
Semirings
Probabilistic inference
Knowledge compilation

ABSTRACT

In the last decade, there has been increasing interest in extensions of answer set programming (ASP) that cater for quantitative information such as weights or probabilities. A wide range of quantitative reasoning tasks for ASP and logic programming, among them probabilistic inference and parameter learning in the neuro-symbolic setting, can be expressed as algebraic answer set counting (AASC) tasks, i.e., weighted model counting for ASP with weights calculated over some semiring, which makes efficient solvers for AASC desirable. In this article, we present aspmc, a new solver for AASC that pushes the limits of efficient solvability. Notably, aspmc provides improved performance compared to the state of the art in probabilistic inference by exploiting three insights gained from thorough theoretical investigations in our work. Namely, we consider the knowledge compilation step in the AASC pipeline, where the underlying logical theory specified by the answer set program is converted into a tractable circuit representation, on which AASC is feasible in polynomial time. First, we provide a detailed comparison of different approaches to knowledge compilation for programs, revealing that translation to propositional formulas followed by compilation to sd-DNNF seems favorable. Second, we study how the translation to propositional formulas should proceed to result in efficient compilation. This leads to the second and third insight, namely a novel way of breaking the positive cyclic dependencies in a program, called $T_{\mathcal{P}}$-Unfolding, and an improvement to the Clark Completion, the procedure used to transform programs without positive cyclic dependencies into propositional formulas. Both improvements are tailored towards efficient knowledge compilation. Our empirical evaluation reveals that while all three advancements contribute to the success of aspmc, $T_{\mathcal{P}}$-Unfolding improves performance significantly by allowing us to handle cyclic instances better.

1. Introduction

In the last decades but especially in recent years, there has been an increasing interest in reasoning frameworks that combine logical and numerical aspects, in order to deal with both qualitative and quantitative uncertainty. This is of particular importance for the integration of machine learning and knowledge representation [1–3], probabilistic planning [4], bio informatics [5,6], natural language processing [7], etc.

The trend of tackling these problems is especially visible in the context of logic programming (LP) [8] and answer set programming (ASP) [9]. There, different approaches for probabilistic reasoning were introduced, among them LP^{MLN} [10], P-log [11],

* Corresponding author.
E-mail addresses: thomas.eiter@tuwien.ac.at (T. Eiter), hecher@mit.edu (M. Hecher), rafael.kiesel@web.de (R. Kiesel).

https://doi.org/10.1016/j.artint.2024.104109
Received 14 April 2023; Received in revised form 21 February 2024; Accepted 1 March 2024

Available online 6 March 2024

ProbLog [12], PITA [13], CP-logic [14], SMProbLog [15], and many others [16–18]. *Neuro-symbolic reasoning*, which builds on top of probabilistic inference for a fixed distribution, is explored by NeurASP [19], SLASH [20], and DeepProbLog [1]. The idea is natural: the semantics for a given set of weights is that of one of the previous probabilistic extensions of LP/ASP but instead of assuming the weights to be given, we learn (to predict) them. Apart from this, also model counting [21,22], optimization [23,24], and other quantitative extensions [25–28] enable reasoning with ASP on problems that go beyond the solution of yes/no questions.

Motivation The plethora of reasoning tasks and settings raises the need to develop a range of algorithms and implementations. This motivates us to consider quantitative reasoning over the set of models in ASP from a general and unifying perspective. Fortunately, while many reasoning tasks in the formalisms above are vastly different at first glance, it was shown that they can be approached in a uniform manner by considering them from an algebraic point of view [29–32]. Specifically, in the *Semiring Paradigm* one uses the abstract algebraic structure of a semiring $\mathcal{R} = (R, \oplus, \otimes, e_\oplus, e_\otimes)$ to capture a multitude of different quantitative semantics by expressions over a set R of elements using an "addition" $\oplus$ and a "multiplication" $\otimes$ that satisfy some axiomes, such as associativity and commutativity. For example, the probabilistic semiring $\mathcal{P} = ([0,1], +, \cdot, 0, 1)$ enables probabilistic inference; the tropical semiring $\mathcal{R}_{\max,+} = (\mathcal{R} \cup \{-\infty\}, \max, +, -\infty, 0)$ corresponds to optimization by choosing the solution with the maximal weight; and Eisner's [33] gradient semiring allows for parameter learning. By performing *algebraic answer set counting (AASC)*, which is to sum up the (semiring) values associated with each answer set of a given program, we then immediately obtain a solution to the original quantitative reasoning task at hand.

Main aim However, solving AASC efficiently poses a challenge. It is well-known that quantitative reasoning may be (much) harder than qualitative reasoning, e.g., lifting in ASP problems from NP-completeness to P^{NP}-completeness or to #P-hardness (see [34] and references therein). This calls for the need of methods and techniques in order to advance the performance of AASC in practice, which is the main goal of our work.

Existing work Since there already exists a wide range of implementations and diverse approaches [35,36,13,37,38,21,39,40], it is suggestive to improve upon an existing idea. However, there is no consensus regarding which evaluation strategy is generally favorable. Thus, it is unclear which approach to choose as the basis of our work. Specifically, in *enumeration*-based approaches [19, 38,37], one enumerates all the answer sets one after the other and sums up their weights one by one. While the idea is simple, it works well for "few" (less than a billion) answer sets. However, it does not scale when there are many answer sets, even for structurally very simple programs. In contrast, *knowledge compilation*-based approaches [36,40,13] are more sophisticated and produce a so called *tractable circuit representation* of a given program. They represent the answer sets of a program, using *knowledge compilation* [41,42], in a manner that can be vastly more succinct than an explicit representation of all the answer sets while still allowing one to perform AASC using linearly many semiring operations in the size of the resulting circuit [30]. Apart from that, there is also the option to perform dynamic programming [21] when the *treewidth* of the program is low enough. Here, treewidth is a parameter that intuitively gives an upper-bound on how complex the structure of a program (or more generally a graph) is by measuring how "treelike" the shape of rule dependencies is, which determines how expensive it is to decompose the problem into smaller ones in a recursive manner.

However, there are not only different basic approaches. For knowledge compilation, for instance, there are different classes of tractable circuits that one can compile to; we may compile some circuits either in a "bottom up" [43,44] or in a "top down" [45–47] manner; and we may either compile the program directly or convert it before compilation into a propositional formula in Conjunctive Normal Form (CNF).

Approach and methods In a first step, we therefore investigate different strategies in depth from a theoretical perspective using recent results from the field of parameterized complexity [48–52]. While these results do not give us general lower bounds for complexity, they do give us upper bounds in terms of structural parameters, such as treewidth or pathwidth, which we can use to derive performance guarantees for the considered strategies. Based on this analysis, we conclude that the most promising strategy is to first translate programs into CNFs and compile them using one of the highly advanced tools from the SAT community [46,45,53,47].

Naturally, this CNF translation should be chosen in such a manner that it leads to good performance of knowledge compilation. From the previous theoretical analysis, we know that small CNFs of low treewidth allow for good worst case guarantees. In fact, there is recent empirical evidence that shows that explicitly exploiting the treewidth of CNFs during knowledge compilation can lead to significantly improved performance [47]. Apart from the size and treewidth, we assume that it is beneficial if the CNF encoding is not very "semantically complex" such that intuitively the knowledge compiler can easily identify parts of the search space that it may discard. This last measure is rather imprecise; nonetheless, we try to use it by drawing intuition from known rules of thumb from CNF encodings for SAT.

The translation of answer set programs to CNFs usually proceeds in two steps. First the cyclic positive dependencies in the program need to be taken care of in *cycle breaking*. Only then, in the second step, Clark's Completion [54] is guaranteed to produce a CNF that correctly captures the semantics of the program. For Clark's Completion itself, we consider it rather unlikely that there is a way to significantly reduce the size or semantic complexity of the resulting CNF. However, previous work by Hecher [55] has shown that we can modify the completion to limit the treewidth increase compared to the *primal treewidth* of the program to a low constant factor. We improve this result to use the *incidence treewidth* of the program [22], which provides bounds that are better or equal.

As for cycle breaking, a wide range of approaches exists [56–59,55], and so we again need a more in-depth analysis. Some of the approaches do not preserve answer sets in a bijective fashion [59,55] and thus have to be discarded, while we examine encoding properties of the rest. Here, we see that the "semantic complexity" of Janhunen and Niemelä's work [56] may due to the use of binary counters prove to be problematic, but the asymptotic guarantees for CNF size and treewidth are far better than those from Mantadelis and Janssens's [58] work. Nevertheless, previous empirical results showed that the asymptotic guarantees do not necessarily translate to better performance, which is presumably due to high constant factors hidden by the Big-O notation and the high "semantic complexity" [36]. We thus introduce a novel strategy for cycle breaking called $\mathrm{T}_{\mathcal{P}}$*-Unfolding*, where we aim to reach a middle ground between the previous approaches. That is, we obtain similarly low "semantic complexity" and constant factors as Mantadelis and Janssens [58] but only moderately higher asymptotic size and treewidth guarantees than Janhunen and Niemelä [56]. In order to achieve this, we resort not only to the number of atomic formulas that are involved in cyclic dependencies, but we also provide a novel, more sensitive measure of the cyclicity of the positive dependencies. This measure, which we call the *component-boosted backdoor size*,[1] is related to the notion of *backdoor size* into acyclicity [60], which intuitively asks how many atoms we must remove to make the (dependency) graph acyclic.

In addition to our theoretical advancements, we implemented our findings in the open source AASC solver aspmc and use it to validate that our theoretical considerations indeed result in an increased performance in practice. To this end, we conducted a thorough empirical evaluation on typical benchmarks from the probabilistic logic programming community, which confirms previous expectations regarding the CNF sizes and treewidths as well as the performance increase.

Main contributions Summing up, we make in this work briefly the following main contributions:

1. Our analysis of different approaches to the knowledge compilation step in AASC highlights possible strengths and weaknesses allowing for a more informed choice of the knowledge compilation method in the AASC pipeline and provide explanations as to why some strategies may work better than others.
2. We introduce an improved version of Clark's Completion [54] that can, in some cases, provide even better treewidth guarantees than Hecher's [55] previous modification with minimal overhead otherwise.
3. By employing a more fine-grained measure of cyclicity in answer set programs, we define $\mathrm{T}_{\mathcal{P}}$-Unfolding, a new approach for cycle breaking that has favorable properties when it comes to knowledge compilation.
4. We provide an open source implementation of our improved and specialized algorithms for Clark's Completion and cycle breaking in the AASC solver aspmc. In addition, aspmc enables the use of recent advancements in knowledge compilation by integrating state of the art implementations [47,46,45,53].
5. We report on an extensive empirical evaluation using our implementation. The results reveals that all three improvements provide significant performance benefits:

   - our modification to Clark's Completion outperforms previous versions in most cases;
   - $\mathrm{T}_{\mathcal{P}}$-Unfolding, results in small CNF encodings with a moderate increase in the treewidth;
   - the overall performance is further boosted by employing state of the art tools for knowledge compilation.

Additional contributions of the extended version Some of the results in this article were presented, in preliminary form, at the KR 2021 conference [61] and the PLP 2021 workshop [62]. The additional contributions of this work, are twofold. First, we significantly improved the implementation. Here, we on the one hand improved efficiency, by incorporating additional specialized tools, e.g., to compute approximations of our measure of cyclicity [63] and for the knowledge compilation step [47,53]. On the other hand, we enabled inference for the full class of probabilistic logic programs, with annotated disjunctions. Second, we incorporated an in-depth analysis of the different approaches to AASC. Here, we not only added a wide ranges of theoretical tractability guarantees but also significantly extended our practical evaluation by comparing a wide range of internal configurations and by including more solvers and benchmarks to the general comparison to the state of the art.

Relation to AI and usage of our results Algebraic operations and counting solutions are fundamental tasks in mathematics and computer science [64–66], with applications to areas such as mathematical combinatorics [67], learning [68,10], neuro-symbolic reasoning [38,69], and probabilistic inference [70,15,11]. Given this strong basis, AI researchers and developers can benefit from the results of our work in different regards. On the foundational side, researchers can acquire new insights in properties of knowledge compilation methods. This is particularly interesting also for propositional formulas that encode properties like transitive closure, acyclicity, or are based on cyclic derivations, e.g., reachability in a graphical network. Our work also implies how to overcome these challenging barriers without destroying structural properties that are essential for efficiently employing algebraic (semiring) operations. On the practical side, researchers can directly take advantage of our new notions and methods, in particular, the improved Clark's Completion, component-boosted backdoors, and the $\mathrm{T}_{\mathcal{P}}$-Unfolding for developing novel reasoning engines or advancing existing engines. Furthermore, on the application side, developers may employ aspmc as a back-end solver or library for easy off-the-shelf evaluation. Indeed, our excessive empirical evaluation underlines the strengths of aspmc, paving the way to future applications in systems utilizing counting, quantitative reasoning, or neurosymbolic learning and reasoning.

[1] Unsurprisingly, like other parameters it is NP-hard to compute in general, however we find that approximations are both efficient and precise enough to be useful.

Organization The remainder of this article is organized as follows. We first provide the necessary preliminaries in Section 2 and then introduce AASC in Section 3. After this, we shortly go over some instantiations of AASC highlighting its broad applicability for quantitative inference in Section 4, before directing our attention to AASC evaluation in Section 5, where we give special attention to different approaches for the knowledge compilation step. We finish the theoretical considerations with a study of Clark's Completion and cycle breakings, in Section 6 and Section 7, respectively. Based on this, we outline the practical realization in our solver aspmc, in Section 8, which we evaluate empirically, in Section 9. Finally, we conclude and highlight possible further points of attack for improved performance in Section 10.

2. Preliminaries

We begin by giving the background on ASP, graph representations of programs, their associated structural parameters, and the algebraic structure of semirings.

2.1. Logic programming

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

$$a \leftarrow b_1, \dots, b_m, \text{not}\, c_1, \dots, \text{not}\, c_n,$$

where a and all b_j, c_k are propositional atoms. Given such a rule r, we let

$$H(r) = a, \qquad B^+(r) = \{b_1, \dots, b_n\},$$
$$B^-(r) = \{c_1, \dots, c_n\}, \qquad B(r) = \{b_1, \dots, b_n, \text{not}\, c_1, \dots, \text{not}\, c_n\}.$$

We slightly abuse notation and write

$$\leftarrow b_1, \dots, b_m, \text{not}\, c_1, \dots, \text{not}\, c_n$$

for

$$\bot \leftarrow b_1, \dots, b_m, \text{not}\, c_1, \dots, \text{not}\, c_n, \text{not}\, \bot,$$

where $\bot$ is a propositional atom that otherwise does not occur in Π.

Furthermore, we allow *choice rules* $\{a\} \leftarrow B^+(r), B^-(r)$ as a shorthand for the two rules $a \leftarrow B^+(r), B^-(r), \text{not}\, na$ and $na \leftarrow B^+(r), B^-(r), \text{not}\, a$, where na is a fresh propositional atom. We denote by $\mathcal{A}(\Pi)$ the set of propositional atoms that occur in Π.

An *interpretation*, denoted $\mathcal{I}$, is a subset of $\mathcal{A}(\Pi)$; it *satisfies* an atom $a \in \mathcal{A}(\Pi)$ (resp. $\text{not}\, a$ for $a \in \mathcal{A}(\Pi)$), written $\mathcal{I} \models a$ (resp. $\mathcal{I} \models \text{not}\, a$), if $a \in \mathcal{I}$ (resp. $a \notin \mathcal{I}$). It satisfies Π (is a *model* of Π), if for each rule $r \in \Pi$ it holds that either $\mathcal{I} \models H(r)$ or $\mathcal{I} \not\models B(r)$, i.e. there exists some $l \in B(r)$ such that $\mathcal{I} \not\models l$. Furthermore, $\mathcal{I}$ is an *answer set* of Π if it is a $\subseteq$-minimal model of the *reduct*[2] $\Pi^{\mathcal{I}} = \{r \in \Pi \mid B^+(r) \subseteq \mathcal{I}, B^-(r) \cap \mathcal{I} = \emptyset\}$, of Π with respect to $\mathcal{I}$, i.e. the set of rules r in Π where $B(r)$ is satisfied. We denote the set of answer sets of a program Π by $\mathcal{AS}(\Pi)$.

As usual the schematic rules with variables $X, Y, \dots$ are implicitly universally quantified and their semantics is given by grounding (instantiation) with concrete values (constants).

Example 1 *(Smokers).* We consider the smokers program, which is a standard example from probabilistic logic programming [12].

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

This encodes that for each person it is randomly determined whether they are stressed. Stressed persons smoke may influence friends, which is again random, to also smoke. We shall abbreviate stress(.) and smokes(.) by st(.) and sm(.), respectively.

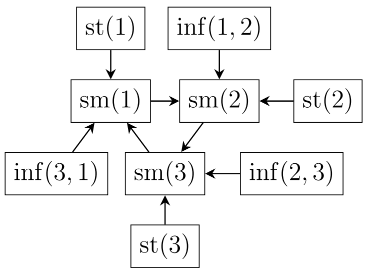
2.2. Propositional logic

We aim to translate programs into formulas of propositional logic.

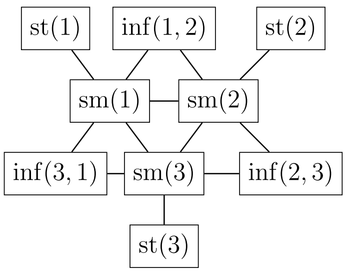
We use propositional formulas in Conjunctive Normal Form (CNF). A CNF C, defined for a set V of variables, is a finite conjunction of *clauses* C_i, where each clause consists of a finite disjunction of *literals* $l \in \{v, \neg v\}$ for some $v \in V$.

We use the standard satisfaction relation and call an interpretation $\mathcal{I} \subseteq V$ that satisfies a formula a *model*.

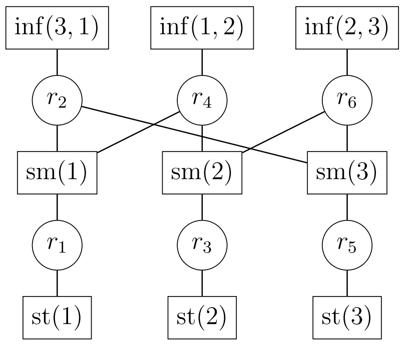
[2] All our results hold for both the FLP-reduct [71] and GL-reduct [72].



1a. Dependency graph of Π_{sm} restricted to non-choice rules.



1b. Primal graph of Π_{sm} restricted to non-choice rules.



1c. Incidence graph of Π_{sm} restricted to non-choice rules. Vertices for rules are circles, vertices for atoms are rectangles

Fig. 1. Different graphs associated with the running example program Π_{sm} for simplicity restricted to non-choice rules (i.e. with rules of the form $\{\text{stress}(x)\} \leftarrow$ etc.)

2.3. Graphs and digraphs

Especially in Section 7.4 we will consider graphs and digraphs, using the following notation. The vertex- and edge-set of a (di)graph $G = (V, E)$ is denoted by $V(G)$ and $E(G)$, respectively. We denote the undirected edge between vertices $a, b \in V(G)$ of a graph G by $\{a, b\}$, whereas for the directed edge from a to b of a digraph, we use the notation (a, b). 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 which use 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 $\text{SCC}(G)$ 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* $\text{DEP}(\Pi)$ 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* $\text{PRIM}(\Pi)$ 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 $x, y \in \{H(r)\} \cup B^+(r) \cup B^-(r)$. The *incidence graph* $\text{INC}(\Pi)$ of Π is the graph G with $V(G) = \mathcal{A}(\Pi) \cup \Pi$ and $\{a, r\} \in E(G)$ if there is a rule $r \in \Pi$ such that $a \in \{H(r)\} \cup B^+(r) \cup B^-(r)$.

Intuitively, all three graphs represent that the truth values of atoms (and rules in the case of the incidence graph) connected via an edge directly depend on one another. More specifically, the dependency graph is related to whether an atom could positively occur on a derivation of another one and the primal graph is related to whether satisfaction of one atom may directly limit the possible truth values of another atom in models of a program. The incidence graph is similar to the primal graph but allows for a more fine grained analysis by including not only atoms but also rules.

Example 2 *(cont'd)*. Given the input data person(i) for $i = 1, \dots, 3$ as well as $\texttt{friend}(i, j)$ for $(i, j) = (1, 2), (2, 3), (3, 1)$, we can ground and reduce (by omitting facts and removing them from the bodies of rules, etc.) the smokers program to Π_{sm}

$$\{\text{st}(1)\} \leftarrow \qquad \{\text{st}(2)\} \leftarrow \qquad \{\text{st}(3)\} \leftarrow$$

$$\{\text{inf}(3,1)\} \leftarrow \qquad \{\text{inf}(1,2)\} \leftarrow \qquad \{\text{inf}(2,3)\} \leftarrow$$

$$r_1 = \text{sm}(1) \leftarrow \text{st}(1) \qquad r_2 = \text{sm}(1) \leftarrow \text{inf}(3,1), \text{sm}(3)$$

$$r_3 = \text{sm}(2) \leftarrow \text{st}(2) \qquad r_4 = \text{sm}(2) \leftarrow \text{inf}(1,2), \text{sm}(1)$$

$$r_5 = \text{sm}(3) \leftarrow \text{st}(3) \qquad r_6 = \text{sm}(3) \leftarrow \text{inf}(2,3), \text{sm}(2)$$

The dependency, primal, and incidence graph of the resulting program Π_{sm} without choice rules is given in Fig. 1a, Fig. 1b, and Fig. 1c, respectively.

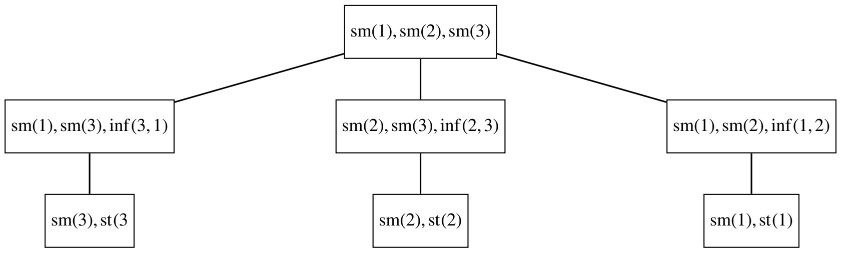


Fig. 2. An optimal tree decomposition of the graph in Fig. 1b. Each vertex is labeled by the vertices in the corresponding bag.

Next, we recall the definition of treewidth.

Definition 1 *(Tree Decomposition, Treewidth).* A *tree decomposition* of a graph G 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.

Intuitively, treewidth is a measure of the distance of a graph from a tree. It is motivated by the fact that many computationally hard problems are tractable on trees. Correspondingly, trees are the only graphs that have treewidth 1. Given low treewidth it is then often possible to generalize tractability results by decomposing problems recursively into smaller subproblems using a tree decomposition witnessing the low width.

A more restricted parameter than treewidth is pathwidth.

Definition 2 *(Path Decomposition, Pathwidth).* Let G be a graph. Then, a *path decomposition* is a tree decomposition (T, χ), where T is a path.

The *pathwidth* of a graph is the minimal width of any of its path decompositions.

Pathwidth is more restricted than treewidth but has similar properties, i.e. it allows for the generalization of tractability results on graphs that are paths. Notably, when the treewidth of an n-vertex graph G is k, then the pathwidth of G is in $\mathcal{O}(k \cdot \log(n))$ [73].

Then, the primal (resp. incidence) treewidth of a program is the treewidth of its primal (resp. incidence) graph. The same applies to pathwidth.

Example 3 *(cont'd).* The treewidth of the graph in Fig. 1b is 2, as its tree decomposition in Fig. 2 which has width 2 shows. A smaller width is not possible here, since the graph contains a clique over three vertices (e.g. $\mathrm{sm}(1), \mathrm{sm}(2), \mathrm{sm}(3)$). The treewidth of the graph in Fig. 1c is also 2, since it is not a tree as the cycle $r_2 \rightarrow \mathrm{sm}(3) \rightarrow r_6 \rightarrow \mathrm{sm}(2) \rightarrow r_4 \rightarrow \mathrm{sm}(1) \rightarrow r_2$ shows.

Apart from the logical aspects specified using the program, we also need semirings to specify the quantitative aspects.

2.4. Semirings

A *semiring* $\mathcal{R} = (R, \oplus, \otimes, e_\oplus, e_\otimes)$ consists of a non-empty 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.

All of the aforementioned semirings are commutative.

For a more comprehensive list of semirings and applications see [30].

3. Algebraic answer set counting

We now introduce ASP with algebraic measures as a basic formalism for Algebraic Answer Set Counting (AASC). Technically, ASP with algebraic measures extends ASP with capabilities for quantitative reasoning over the set of answer sets such that algebraic Prolog, ProbLog, LP$^{\mathrm{MLN}}$ and P-log can all be expressed.

We use a variant of weighted logics [74] 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 atom. The semantics of α w.r.t. an interpretation $\mathcal{I}$, denoted $[\![\alpha]\!]_{\mathcal{R}}(\mathcal{I})$, is

$$\begin{aligned}
[\![k]\!]_{\mathcal{R}}(\mathcal{I}) &= k,\\
[\![v]\!]_{\mathcal{R}}(\mathcal{I}) &= \begin{cases} e_\otimes & v \in \mathcal{I} \\ e_\oplus & \text{otherwise} \end{cases},\\
[\![\neg v]\!]_{\mathcal{R}}(\mathcal{I}) &= \begin{cases} e_\oplus & v \in \mathcal{I} \\ e_\otimes & \text{otherwise} \end{cases},\\
[\![\alpha_1 + \alpha_2]\!]_{\mathcal{R}}(\mathcal{I}) &= [\![\alpha_1]\!]_{\mathcal{R}}(\mathcal{I}) \oplus [\![\alpha_2]\!]_{\mathcal{R}}(\mathcal{I}),\\
[\![\alpha_1 * \alpha_2]\!]_{\mathcal{R}}(\mathcal{I}) &= [\![\alpha_1]\!]_{\mathcal{R}}(\mathcal{I}) \otimes [\![\alpha_2]\!]_{\mathcal{R}}(\mathcal{I}).
\end{aligned}$$

We can use weighted formulas to express calculations over a semiring, depending on the truth values of atoms with respect to an interpretation.

Example 4 *(cont'd).* As mentioned before, the smokers program is a typical example from the probabilistic domain. Using weighted formulas, we can introduce probabilities. We define

$$\alpha = \Pi_{i=1}^{3} \mathrm{st}(i) * 0.4 + \neg \mathrm{st}(i) * 0.6 \tag{1}$$

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

Line (1) tells us that every person i is either stressed, with probability 0.4, or not stressed, with probability 0.6. With line (2) we capture that for $(i,j) = (1,2)$, $(2,3)$, $(3,1)$ person i is influenced by person j with probability 0.3 and not influenced with probability 0.7.

Using weighted formulas, we define algebraic measures as follows:

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 $\mathcal{R}$. Then, the weight of an answer set $\mathcal{I} \in \mathcal{AS}(\Pi)$ under μ is defined by

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

Additionally, the result of an *(atomic) query* for an atom $a \in \mathcal{A}(\Pi)$ is given by

$$\mu(a) = \bigoplus\nolimits_{\mathcal{I} \in \mathcal{AS}(\Pi), a \in \mathcal{I}} \mu(\mathcal{I}),$$

and the result of the *overall weight query* of Π is

$$\mu(\Pi) = \bigoplus\nolimits_{\mathcal{I} \in \mathcal{AS}(\Pi)} \mu(\mathcal{I}).$$

Example 5 *(cont'd).* Using algebraic measures, we perform probabilistic reasoning. We combine the weighted formula α, which handles the probabilities, and the program Π_{sm}, which handles the logical background theory, in the measure $\mu_{sm} = \langle \Pi_{sm}, \alpha, \mathcal{P} \rangle$. Then, 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. Since Π_{sm} has 2^6 answer sets out of which $2^5 + 2^3 + 2$ include $\mathrm{sm}(1)$, we refrain from computing $\mu(\mathrm{sm}(1))$ naïvely by considering all the answer sets separately. Instead, we consider the following three disjoint cases in which $\mathrm{sm}(1)$ holds:

1. $\mathrm{st}(1)$ is true and the other probabilistic atoms take an arbitrary truth value;

2. st(1) is false, inf(3, 1) and st(3) are true and the other probabilistic atoms take an arbitrary truth value;
3. st(1) and st(3) are false, inf(3, 1), inf(2, 3) and st(2) are true, and inf(1, 2) takes an arbitrary truth value.

These three cases are exclusive, i.e. no answer set falls in any two of the cases, and furthermore, they cover all the answer sets in which sm(1) is true. Thus, we can compute the probability of sm(1) as a sum of the probabilities of the three cases. These are $0.4, 0.6 * 0.3 * 0.4 (= 0.072)$ and $0.6 * 0.6 * 0.3 * 0.3 * 0.4 (= 0.01296)$, respectively, since it is enough to take the product of the probabilities of the atoms whose truth values we assert as the other atoms that are left arbitrary only contribute a factor of 1. Thus, $\mu(\mathrm{sm}(1)) = 0.48496$.

Following the conventions of [75], 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

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

The difference between factorized and non-factorized measures is intuitively how complicated the weight function is. The former must be expressible as a product of weights of literals that are true in a given interpretation, whereas the latter allow complex arithmetic expressions using both the addition and multiplication of weights in dependence on the true literals in the interpretation.

The motivation behind considering factorized measures is that current frameworks, such as ProbLog and algebraic Prolog only use factorized measures. This raises the question of whether this covers everything we need.

Example 6 *(cont'd).* The measure μ_{sm} is factorized over $\mathrm{st}(i), i = 1, 2, 3$ and $\mathrm{inf}(i, j), i + 1 \equiv j \bmod 3$, by letting $\beta(\mathrm{st}(i)) = 0.4, \beta(\neg\mathrm{st}(i)) = 0.6$ and $\beta(\mathrm{inf}(i, j)) = 0.3$, $\beta(\neg\mathrm{st}(i, j)) = 0.7$.

Not every measure is factorized however.

Example 7 *(Non-factorized).* For an example of a non-factorized measure, consider the following problem: Given a program Π, whose answer sets correspond to feasible applicants for a software development position and which include whether the selected applicant knows C++ (denoted by $\mathrm{knows}(c)$) and python (denoted by $\mathrm{knows}(p)$). We aim to give candidates an additional score of 2, if they know both languages, a score of 1 of they know one of them, and a score of -1 if they know neither language.

We implement this using the measure

$$\mu_w = \langle \Pi, \mathrm{knows}(c) + \mathrm{knows}(p) + (-1 * \neg\mathrm{knows}(c) * \mathrm{knows}(p)), \mathbb{Z} \rangle.$$

This measure is not factorized over $F = \{\mathrm{knows}(c), \mathrm{knows}(p)\}$ as there are no values $\beta(\mathrm{knows}(c)), \beta(\mathrm{knows}(p)), \beta(\neg\mathrm{knows}(c))$, $\beta(\neg\mathrm{knows}(p)) \in \mathbb{Z}$, s.t.

$$\beta(\mathrm{knows}(c)) \cdot \beta(\mathrm{knows}(p)) = 2 \qquad \beta(\mathrm{knows}(c)) \cdot \beta(\neg\mathrm{knows}(p)) = 1$$
$$\beta(\neg\mathrm{knows}(c)) \cdot \beta(\mathrm{knows}(p)) = 1 \qquad \beta(\neg\mathrm{knows}(c)) \cdot \beta(\neg\mathrm{knows}(p)) = -1$$

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 be a weighted formula. Then, $\mathcal{S}(\alpha)$ is the set of pairs (i, β), where β is a subformula of α indexed by a position-string $i \in \{0, 1\}^*$ in the syntax-tree of α, 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 apply the distributive law. For this, we introduce a new atom α_i for every indexed subformula $(i, \beta) \in \mathcal{S}(\alpha)$, which is true if the value of the subformula β at index i is included in the current product. To implement this, we let $\Pi' = \Pi \cup \Pi_{\mathrm{root}} \cup \Pi_* \cup \Pi_+ \cup \Pi_{\mathrm{leaf}}$, where $\Pi_{root} = \{\leftarrow \mathrm{not}\ \alpha_0\}$ ensures that the value of the formula at the root is included.

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

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.

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

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

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

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 $\beta(\alpha_i) = k, \beta(\neg \alpha_i) = e_\otimes$.

Furthermore, along a similar line of reasoning as in [34] it follows that $\mu(\Pi) = \mu'(\Pi')$. $\square$

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 during evaluation but allow for the specification of measures in the general form.

4. Applications of AASC

We covered the introductory theory necessary when dealing with AASC and algebraic measures. In the following, we shortly reiterate some of the prominent applications of AASC via instantiation with fixed semirings.

The idea behind AASC is ubiquitous in quantitative reasoning: one takes a Boolean logic and turns it into a weighted logic, by replacing conjunctions by multiplications and disjunctions by additions over a semiring. Other such semiring reasoning frameworks are for example Semiring-based Constraint Satisfaction Problems [76], provenance semantics for datalog [77], and Algebraic Model Counting (AMC) [30], to name only a few. Due to this, a wide range of applications of the semiring semantics are already known. Kimmig et al. [30] give an extensive list of different reasoning tasks that can be performed using the semiring semantics and discuss the appropriate semirings for it.

We therefore do not extensively list all the different possibilities here but focus on some that relate to existing extensions of ASP and, thus, have higher relevance for our setting. Namely, we focus on two different classes of problems and touch upon a third. First, we consider problems that are #P-hard, i.e. at least as hard as counting the number of satisfying assignments of a propositional formula. For these problems knowledge compilation is a well-established and competitive technique [46,45]. Second, when we work over idempotent semirings, algebraic measure evaluation is often equivalent to OPTP-hard problems. Here, at least in the area of Weighted MaxSAT, a prototypical OPTP-hard problem, knowledge compilation does not seem to be a good strategy, given that none of the solvers of the recent MaxSAT Evaluation [78] is knowledge compilation based. Thus, while we relate AASC to OPTP-hard problems here and cover their solution also with our implementation (which works for general commutative semirings), we do not expect performance improvements compared to specialized tools such as clingo [79] here.

We note that there are relevant problems that are in a complexity class above #P. Recent work [32] showed that many such problems can be solved by AASC over two semirings but not over one of them. Here, the problems are typically harder than AASC over one semiring, since they remain NP-hard on the usual tractable circuit representations [80]. Note that, as with the OPTP-hard problems, an implementation that works for general semirings is available but not further considered here, since we focus on the evaluation of problems that are #P-hard but solvable on typical tractable circuit representations. For details we refer the interested reader to [32].

4.1. #P-hard problems

Probabilistic reasoning The most common application that involves weighted model counting over the answer sets, in the typical sense, is probabilistic inference as in ProbLog, LP$^{\text{MLN}}$, or P-log. The specification and semantics of probabilistic programs as well as the allowed class of programs varies between the languages. In fact, there is also *Credal Semantics* for probabilistic programs [81], which we however leave untreated in this work.

The main difference between the treated languages is that ProbLog uses Sato's distribution semantics [82], which assumes that the only non-determinism in the program is due to a set of probabilistic facts and that the program has no constraints. This means that any given truth assignment to the probabilistic facts can be *uniquely* extended to a stable model of the program. The benefit of this restriction is that the measure specified by the program directly satisfies the axioms of a probability distribution, e.g. the weights of all answer sets add up to one. On the other hand, LP^{MLN} and P-log do not impose such a restriction, thus allowing for a broader class of programs. Here, in order to obtain a probability measure, the probability of an answer set is not its weight but its *normalized* weight. That is, its weight is divided by the sum of the weights of all answer sets.

Apart from that, LP^{MLN} features another property. Namely, while ProbLog and P-log only allow for weights of atoms or rules that are between 0 and 1, LP^{MLN} allows for any real number as a weight.

For probabilistic inference by means of AASC, both of these differences are not significant. We can perform probabilistic inference in any of the three languages by evaluating an algebraic measure over an answer set program. For ProbLog programs, it is sufficient to evaluate $\mu(a)$ for the algebraic measure μ corresponding to the program to obtain the probability that the query atom a is true according to the program. In the ProbLog case, we can use the probabilistic semiring $\mathcal{P}$ since only weights in $[0,1]$ occur. For LP^{MLN} and P-log programs, we can compute the probability of a query atom a by evaluating $\mu(a)$ and $\mu(\text{not } a)$ over the semiring $\mathbb{R}$ of the real numbers. Then, $Pr(a)$, the probability of a, is $^{\mu(a)}/_{\mu(a)+\mu(\text{not } a)}$. For a closer look at the relationship of the different probabilistic logic programming languages we refer to [83,10,38].

Neuro-symbolic reasoning Neuro-symbolic reasoning, as for example in DeepProbLog [1], takes the idea of probabilistic reasoning with programs one step further. Here, instead of assuming that the probabilities of rules and facts are given, they are determined by a machine learning model that predicts the probability of a fact or rule for a given ground instantiation of it. Instead of having the same probability of stress for each person

$$0.4 :: \text{stress}(X) : \mp \text{ person}(X),$$

we might have a *neural predicate* $\text{nn}{\mp}\text{stress}(Age, Employment)$ that models the probability of a person being stressed based on age and employment. In the DeepProbLog language, this would be expressed as

$$\text{nn}{\mp}\text{stress}(A, E) :: \text{stress} : \mp \text{ person}(X), \text{age}(A, X), \text{employment}(E, X).$$

We consider two tasks in neuro-symbolic reasoning that need to be solved.

(T1) Determine the probability of a query, given a neuro-symbolic program.

For DeepProbLog, this again corresponds to a query using an algebraic measure: when the machine learning model that predicts the probabilities of facts and rules is *fixed* we can see DeepProbLog programs as ProbLog programs and, thus, evaluate probabilistic queries using algebraic measures. Here, we can again use the probabilistic semiring $\mathcal{P}$.

(T2) Train the model to predict the correct probabilities.

Usually, this is done using gradient descent to minimize some error function *err* that depends on the output of the neural networks. In order to use gradient descent, it is necessary to compute the derivative of *err*. In the DeepProbLog setting, this can be rather complicated, as *err* is not limited to measuring whether the predicted value was correct. Instead, the error can also be determined by the probability of a derived atom. E.g. in the smokers setting, we may have data about whether a person smokes but not about a persons stress. Thus, applying gradient descent to the error function of the smokers model involves obtaining its derivative in terms of the output of the neural network.

Manhaeve et al. [1] showed that we can compute this gradient by weighted model counting over the *gradient semiring* $GRAD$. Therefore, we can also model this task using an algebraic measure over

$$GRAD = ([0,1] \times \mathbb{R}, \oplus, \otimes, (0,0), (1,0)),$$

where

$$(p_1, d_1) \oplus (p_2, d_2) = (p_1 + p_2, d_1 + d_2)$$
$$(p_1, d_1) \otimes (p_2, d_2) = (p_1 p_2, d_1 p_2 + d_2 p_1).$$

Intuitively, the first entry corresponds to the probability, as before, whereas the second entry computes the derivative of the first with respect to some parameter.

Knowledge compilation Using the AASC approach we may further obtain a tractable circuit representation of the program. This may seem counterintuitive, since we said in the Introduction that we use knowledge compilation to perform AASC and not the other way around. However, it shows that when we can do weighted model counting over semirings, then we can also do knowledge compilation.

For knowledge compilation, we need a somewhat non-standard semiring similar to the provenance semirings of Green et al. [77]. Given a semiring $\mathcal{R}$ and a set of propositional atoms $\mathcal{V}$, we define the knowledge compilation semiring of algebraic circuits over $\mathcal{V}$ and $\mathcal{R}$ as

$$\mathcal{KC}(\mathcal{V},\mathcal{R}) = (\mathcal{AC}(\mathcal{V},\mathcal{R}), +, *, \bot, \top)$$

$$\mathcal{AC}(\mathcal{V},\mathcal{R}) = \{\alpha \mid \alpha \text{ is an algebraic circuit over } \mathcal{V} \text{ and } \mathcal{R}\}$$

$$\alpha + \beta = \alpha + \beta$$

$$\alpha * \beta = \alpha * \beta$$

Here, an algebraic circuit over $\mathcal{V}$ and $\mathcal{R}$ is a directed acyclic graph (DAG), where each vertex is labeled by either $+, *$, a literal $l \in \mathrm{Lit}(\mathcal{V})$ with $\mathrm{Lit}(\mathcal{V}) = \mathcal{V} \cup \{\neg v \mid v \in \mathcal{V}\}$, or a semiring value $r \in R$, in such a manner that vertices without incoming edges are either labeled by a literal or a semiring value and vertices with incoming edges are either labeled by $+$ or $*$. Furthermore, there must be exactly one vertex that does not have outgoing edges, which we refer to as the *output*.

Clearly, using the standard equality in the sense of syntactical equality is inadequate, since $\mathcal{KC}(\mathcal{V},\mathcal{R})$ is not commutative:

$$a + b \neq_{\mathrm{syntax}} b + a.$$

We instead use semantic equality, i.e. two algebraic circuits $\alpha, \beta \in \mathcal{AC}(\mathcal{V},\mathcal{R})$ are *semantically equal* if for every weight assignment $\sigma : \mathrm{Lit}(\mathcal{V}) \to R$ it holds that

$$[\![\alpha[\sigma]]\!]_{\mathcal{R}} = [\![\beta[\sigma]]\!]_{\mathcal{R}}.$$

Here, $\alpha[\sigma]$ denotes that we replace every literal l in α by $\sigma(l)$ its semiring value under σ. Then, $[\![\alpha[\sigma]]\!]_{\mathcal{R}}$ corresponds to evaluating the circuit by inductively defining the value of a node with a semiring label r as r and the value of a node labeled by $+$ (resp. $*$) with incoming nodes with values $r_1, \dots, r_n$ as $\bigoplus_{i=1}^n r_i$ (resp. $\bigotimes_{i=1}^n r_i$. The final value of $[\![\alpha[\sigma]]\!]_{\mathcal{R}}$ is then given by the value of the output of the circuit.

Note that this idea is not new: Kimmig et al. [29] introduced a BDD semiring, where conjunction and disjunction correspond to the respective operation on BDDs, thus, allowing knowledge compilation to a BDD. However, in contrast to the BDD semiring, our semiring allows any kind of algebraic circuit, even those that classically do not correspond to a tractable circuit class such as BDD, SDD, or d-DNNF.

Apart from this, we may obtain syntactically different results depending on how we evaluate a query over $\mathcal{KC}(\mathcal{V},\mathcal{R})$. This, however, is of no concern to us. We can show that given any such syntactic representation of the result, we can evaluate meaningful queries.

Lemma 8 *(Semiring Knowledge Compilation). Let $\mu = \langle \Pi, \alpha, \mathcal{R}\rangle$ be a factorized algebraic measure with weight function β that factorizes μ. Furthermore, let*

$$\mu_{KC} = \langle \Pi, \alpha_{KC}, \mathcal{KC}(\mathcal{A}(\Pi), \mathcal{R})\rangle$$

$$\alpha_{KC} = \Pi_{v \in \mathcal{A}(\Pi)}(v * \mathbf{v} + \neg v * \neg \mathbf{v}),$$

where we write atoms in italics v and semiring values in boldface $\mathbf{v}$.

Then, given any algebraic circuit γ that represents the result of $\mu_{KC}(\Pi)$, we have that $\mu(\Pi) = [\![\gamma[\sigma_\beta]]\!]_{\mathcal{R}}(\emptyset)$, where $\sigma_\beta : \mathrm{Lit}(\mathcal{A}(\Pi) \to R$ is given by $\sigma_\beta(l) = \beta(l)$.

Thus, given any algebraic circuit γ that corresponds to the result of the knowledge compilation query $\mu_{KC}(\Pi)$, we can evaluate any factorized measure query over $\mathcal{R}$ using γ in linearly many semiring operations in the size of γ. Thus, the key is to compute a small circuit γ and the latter efficiently. It is well-known that such arithmetic circuits can easily be obtained from BDDs, SDDs, or d-DNNFs [26], by replacing disjunctions and conjunctions by additions and multiplications, respectively.

For us the algebraic circuit semirings are not only of theoretical but also of practical interest. Namely, as we describe in more detail later in the section on implementation, we use SHARPSAT-TD [47], which enables weighted model counting over semirings. Thus, we can use it for knowledge compilation using the idea of the algebraic circuit semirings. This again shows the power of the semiring paradigm. Since SHARPSAT-TD is implemented in such a manner that it works for general commutative semirings, we can solve a variety of different interesting tasks by performing a minor modification. In fact, the adaptation of SHARPSAT-TD to enable knowledge compilation to smooth d-DNNF was possible in less than 500 lines of code, including the parsing of additional input arguments and the adaptation of the preprocessor to handle weights over general semirings. Details and an independent evaluation can be found in the corresponding conference paper [84].

4.2. OptP-hard problems

The applications discussed above share the fact that they are Weighted Model Counting problems in the strict sense, i.e. they are #P-hard. We thus expect that they are practically harder to solve using current implementations than other AASC problems which are not #P-hard. For example, AASC over the Boolean semiring $\mathbb{B}$ corresponds to checking the consistency of an answer set program and is therefore NP-complete. Arguably, it makes sense to consider such problems which are practically easier to solve separately.

Another class of problems that are affected by this reasoning are *optimization problems*. There is a broad variety of extensions of ASP that introduce different ways to add preferences to programs [23,24]. We focus instead on the Most Probable Explanation (MPE) task from the probabilistic logic programming literature [85], which also fits into the context of optimization problems.

Most probable explanation The Most Probable Explanation (MPE) task is to determine the most likely probabilistic choices that led to a set of observed facts. We consider it in the setting of ProbLog programs.

Definition 9 *(MPE).* Given a ProbLog program Π and a set $E = \{e_1, \dots, e_n\}$ of atoms called evidence, MPE is to determine the most likely assignment to the probabilistic facts such that E is entailed. That is to compute

$$p^* = \operatorname*{argmax}_{\mathcal{I} \in \mathcal{AS}(\Pi), E \subseteq \mathcal{I}} \prod_{p::f \in \Pi, f \in \mathcal{I}} p \cdot \prod_{p::f \in \Pi, f \notin \mathcal{I}} (1-p). \tag{3}$$

As we have seen before, we can define a measure μ over the probabilistic semiring $\mathcal{P}$ such that

$$\mu(\mathcal{I}) = \prod_{p::f \in \Pi, f \in \mathcal{I}} p \cdot \prod_{p::f \in \Pi, f \notin \mathcal{I}} (1-p).$$

Naturally, we could use μ as a function to optimize for modeling MPE. However, it would be desirable to have an algebraic measure whose overall weight corresponds to the solution of MPE.

Indeed, we can construct such a measure based on the max-times semiring $\mathcal{R}_{\max,\cdot}$. Recall that the multiplication of $\mathcal{R}_{\max,\cdot}$ corresponds to the usual product and that the addition corresponds to taking the maximum. Then, we can use the following formula α_L over $\mathcal{R}_{\max}$ to compute the likelihood, i.e. the logarithm of the probability of a given assignment to the probabilistic facts:

$$\Pi_{p::f \in \Pi}(f * p + \neg f * (1-p)).$$

Lemma 10 *(MPE as AASC). Given a ProbLog program* Π *and evidence* E*, the overall weight of the algebraic measure* $\mu = \langle \Pi \cup \{ \ : \mp \text{ not } e \mid e \in E\}, \alpha_L, \mathcal{R}_{\max,\cdot} \rangle$ *is equal to* p^* *in Equation* (3)*, the maximum likelihood of any assignment to the probabilistic facts in* Π *such that* E *is entailed.*

Notably, this does not solve MPE strictly speaking, since we do not know the assignment that leads to the maximum probability. However, we can extend the max-times semiring $\mathcal{R}_{\max,\cdot}$ to perform bookkeeping that tracks which interpretation was used. For this, we use pairs $(p, \mathcal{I})$, where $p \in \mathbb{R}$ and $\mathcal{I}$ is a partial interpretation. Then, addition of two such pairs $(p_i, \mathcal{I}_i), i = 1, 2$ takes that pair, where p_i is maximal, breaking ties with an arbitrary but fixed order on the partial interpretations. Multiplication of two such pairs results in $(p_1 \cdot p_2, \mathcal{I}_1 \cup \mathcal{I}_2)$. By changing the probabilistic facts f with weight p to have weight $(p, \{f\})$ instead, we obtain the desired output.

4.3. Harder problems

There are also problems harder than #P, i.e. problems that stay hard on d-DNNFs without additional restrictions. Such problems frequently occur in probabilistic logic programming. Some examples are Maximum A Posteriori (MAP) problems, probabilistic inference using the semantics for general answer set programs of Totis et al. [15] and Skryagin et al. [20], and Maximum Expected Utility (MEU) problems [26,25].

What all these problems share is that they come with a partition X_O, X_I of the variables and two semirings $\mathcal{R}_O, \mathcal{R}_I$ such that their final result can be expressed as

$$\bigoplus\nolimits^{O}_{\mathcal{I}_O \subseteq X_O} \bigotimes\nolimits^{O}_{l \in \text{lit}(X_O), \mathcal{I}_O \models l} \alpha_O(l) \otimes^O t\left(AASC(\mathcal{I}_O)\right), \text{ where} \tag{4}$$

$$AASC(\mathcal{I}_O) = \bigoplus\nolimits^{I}_{\mathcal{I}_I \subseteq X_I, \mathcal{I}_O \cup \mathcal{I}_I \in \mathcal{AS}(\Pi)} \bigotimes\nolimits^{I}_{l \in \text{lit}(X_I), \mathcal{I} \models l} \alpha_I(l), \tag{5}$$

for functions α_O and α_I that assign the literals over X_O and X_I a weight from $\mathcal{R}_O$ and $\mathcal{R}_I$, respectively, and a transformation function t that maps semiring values from $\mathcal{R}_I$ to $\mathcal{R}_O$. Intuitively, this means that we have to solve one AASC instance $AASC(\mathcal{I}_O)$ for each assignment $\mathcal{I}_O$ to the outer variables X_O and sum up the results, i.e. we need to perform nested AASC.

Consider for example the MAP problem. It is similar to MPE in the sense that both problems we ask for the most likely assignment to a set of variables X_O, given some evidence. However, while for MPE the probability of the most likely assignment is determined as the product of the probabilities of the assignments to the variables in X_O, in MAP this product is multiplied by the marginal probability over the remaining probabilistic variables in the set X_I. Thus, for MAP the expression in (5) corresponds to

$$\begin{aligned} &\operatorname*{argmax}_{\mathcal{I}_O \subseteq X_O} \prod_{l \in \text{lit}(X_O), \mathcal{I} \models l} \alpha_O(l) \cdot \sum_{\mathcal{I}_I \subseteq X_I, \mathcal{I}_O \cup \mathcal{I}_I \in \mathcal{AS}(\Pi)} \prod_{l \in \text{lit}(X_I), \mathcal{I} \models l} \alpha_I(l) \\ &= \operatorname*{argmax}_{\mathcal{I}_O \subseteq X_O} Pr(\mathcal{I}_O) \cdot Pr(E \mid \mathcal{I}_O), \end{aligned}$$

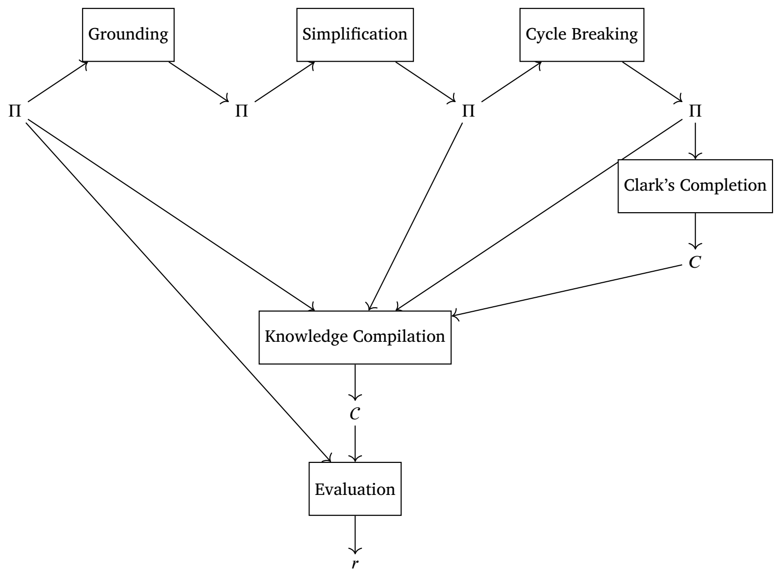


Fig. 3. Schema of the overall workflow of existing solvers for the evaluation of AASC problems.

where $Pr(\mathcal{I}_O)$ and $Pr(E \mid \mathcal{I}_O)$ denote the probability of $\mathcal{I}_O$ and the probability of the evidence E given $\mathcal{I}_O$, respectively. The inner sum computes $Pr(E \mid \mathcal{I}_O)$, the conditional probability of E given the assignment $\mathcal{I}_O$ to the outer variables X_O and the outer sum takes the maximum over all assignments to the outer variables. Here, we assume that Π includes the constraints that the evidence should hold, as in Lemma 10.

Since all these problems are also definable algebraically and can be solved by compilation to *constrained* tractable circuits instead of arbitrary tractable circuits, the results of the rest of the paper are also relevant for this setting. Again, we see the flexibility of the algebraic approach that allows us to consider a large variety in the same manner. For more details, including the adaptations that are necessary in the knowledge compilation step to allow for *constrained* knowledge compilation we refer to [32].

5. Solving AASC problems

The main goal of our work is to solve AASC problems more efficiently. In order to do so, we need to aggregate the values associated with each answer set of a program by the weighted formula. Already computing one answer set is NP-hard and computing an aggregate of over all of them can be even harder, depending on the chosen aggregate, which in our setting corresponds to the addition of the semiring. There has, however, been a lot of research on different approaches that can be used to overcome this hardness in an effort to provide an efficient implementation [36,35,58,39,13]. While these approaches differ significantly in their details, they usually follow a similar overall workflow.

In this section, we give a broad overview of both the shared workflow and the differences between the approach to arrive at a solution. Especially, we investigate the conditions under which each of the approaches is likely to succeed. This is interesting, on the one hand, from a user perspective, as it can allow one to make an informed choice for a tool, depending on the conditions satisfied by the instances at hand. On the other hand, it is useful for us since we can select the most promising approach as the basis for our work.

5.1. Overall workflow

A schema of the general pipeline that is usually shared between different solvers is shown in Fig. 3. Many steps are optional and different solvers use different paths through this pipeline. Usually, solvers accept possibly non-ground programs Π with some additional annotations for the algebraic measure. Often, this program is then grounded, resulting in a propositional program, which is simplified in the next step. Afterwards, the actual AASC starts, by converting the program into an equivalent tractable circuit representation C via some form of *knowledge compilation* (KC). In the last step, the result is obtained by evaluation over the circuit C. Before we go into more details, we would like to note that in some solvers these steps are intertwined instead of being executed one after the other.

We focus on the knowledge compilation step. First, there has already been a lot of work on the other steps of the pipeline: Fierens et al. [35] showed that for probabilistic inference with ProbLog programs, it is sufficient to restrict oneself to the relevant ground part of the program, which can significantly reduce the size of the problem. Tsamoura et al. [86] went even further and used the magic set technique [87] to identify the relevant part during grounding, thus, eliminating the need to even produce the remaining irrelevant part in this phase. Additionally, Shterionov [88, Chapter 3] thoroughly investigated a wide range of simplifications for probabilistic logic programs that were shown to lead to significantly improved solving times on many benchmark sets.

Table 1
Different possibilities for the knowledge compilation step accompanied by a tractability guarantee in terms of an appropriate parameter. Here, ptw is short for primal treewidth, $pdtw$ (resp. $pdpw$) are short for the treewidth (resp. pathwidth) of the primal graph that additionally includes an edge between any two atoms that are in a positive cyclic dependency. Furthermore, C denotes a CNF and Π a program.

KC Step	Tractability Guarantee	Implemented in
program to MODS	$\Theta(\lvert\mathcal{AS}(\Pi)\rvert \cdot \lvert\mathcal{A}(\Pi)\rvert)$	LP$^{\text{MLN}}$, plingo
CNF to sd-DNNF	$\mathcal{O}(2^{ptw}\lvert C\rvert)$	ProbLog
acyclic program to SDD	$\mathcal{O}\left(2^{2^{(ptw+2)2^{(ptw+1)}+1}+1}\lvert\Pi\rvert\right)$	ProbLog
acyclic program to OBDD	$\mathcal{O}\left(2^{(ppw+2)2^{(ppw+2)}}\lvert\Pi\rvert\right)$	PITA
program to sd-DNNF	-	(ProbLog)
dynamic programming	$\mathcal{O}\left(2^{2^{ptw+2}}\lvert\Pi\rvert\right)$	dynasp

Second, we argue that the knowledge compilation step has the largest potential for improvement and is likely to lead to the biggest effect. Due to these reasons, we focus on the knowledge compilation step, as we assume its investigation to be the most fruitful.

5.2. Knowledge compilation

The idea behind knowledge compilation is the following: while certain problems, such as AASC, are hard on general logical theories, there are so called *tractable circuit representations*, where this is not the case. Here, we shortly summarize the different theoretical guarantees associated with compiling a given representation of programs to prominent circuit classes in addition to implementations that support them. For a more in-depth introduction and discussion, we refer the interested reader to Appendix B.

A summary is given in Table 1. We see that compiling to MODS leads to a circuit of size linear in the number of answer sets. Thus, this strategy is tempting, when there are "few" of them, which is however rarely the case for probabilistic logic programs, where the number of answer sets grows exponentially in the number of probabilistic facts. For the compilation of CNFs and acyclic programs as well as the explicit dynamic programming approach [21], we see that the guarantees strongly depend on the structural parameters, i.e., treewidth or pathwidth of associated structures. Here, the guarantee for CNFs seems to be the most desirable, due to the single exponential dependency on treewidth compared to the at least double exponential dependency for the other approaches. It is, however, important to note that (1) these are only upper bounds, rather than lower bounds, and (2) Amarilli et al. [52] were able to obtain an upper bound of $\mathcal{O}\left(2^{5(k+1)} \cdot \lvert\text{Vars}(C)\rvert\right)$ by slightly loosening the requirements for SDDs. Therefore, we may obtain much better performance in practice as the guarantees suggest.

For the direct compilation of programs to sd-DNNF as given by Aziz et al. [89], no guarantees are known. It is interesting, since contrary to CNF and acyclic program compilation, here, the requirement for stability of models is taken care of during compilation rather than beforehand. Taking care of it in advance during cycle breaking requires transformations that may increase the treewidth and size of programs. Here, this is solved by adding so called *loop formulas* [90], which are clauses that prohibit cyclic derivations, during compilation. However, loop formulas as introduced by Lin and Zhao [90] may span a whole SCC of the dependency graph of the program.[3] This can destroy previously low treewidth and be devastating for the compilation performance. Apart from that, not only the treewidth may suffer: we may end up with exponentially many additional clauses in the size of the largest SCC [91]. This version of DSHARP was removed from the ProbLog implementation, presumably for the above reasons. While there are strategies to reduce the number of necessary loop formulas [92,93], their adaptation would not circumvent the exponential worst case lower bound on the number of added clauses.

Summary Summing up, we see that there is a wide range of different approaches to knowledge compilation or more generally solving AASC problems, coming with different theoretical guarantees. Mostly, the guarantees are exponential in a structural parameter associated with the formula that is to be compiled. The only exception is the approach of the solvers LP$^{\text{MLN}}$ [10] and plingo [38] that enumerate all the models.

The most promising theoretical results are known for the compilation of CNFs to sd-DNNFs. Furthermore, the recent work of Korhonen and Järvisalo [47] showed that one can achieve performance improvements by using the latter explicitly. This begs the question whether and how we can convert programs of low treewidth into CNFs of low treewidth, in order to fruitfully exploit these recent advancements.

6. Clark's completion

A crucial step in the translation of programs to CNF formulas is the well-known Clark's Completion [54]. While it is defined on general programs, it is only guaranteed to lead to an equivalent CNF, when the program it was applied to does not have cycles in

[3] They usually contain even more variables, however already this is devastating for knowledge compilation.

its positive dependence graph. The traditional, most well-known version of Clark's Completion does not allow for bounds on the treewidth of the resulting CNF in terms of the treewidth of the primal graph of the original program. There has, however, already been work by Hecher that does imply such bounds. In the following, we first discuss the original version of Clark's Completion and its limitations as well as the modifications by Hecher. After that we show that even Hecher's version allows for further improvements and introduce an extended version for which we can obtain better bounds.

The idea behind Clark's Completion is the following: if we have an atom a that is in the head of the rules $r_1, \dots, r_n$, then a is only included in a stable model if we are forced to derive it due to one of these rules. More formally, the definition is as follows:

Definition 11 *(Clark's Completion).* Let Π be a normal answer set program. Then, Clark's Completion of Π is defined as the propositional formula

$$\mathrm{Clark}_{\mathrm{Prop}}(\Pi) = \bigwedge_{a \in \mathcal{A}(\Pi)} a \leftrightarrow \bigvee_{r \in \Pi_a} \bigwedge_{l \in B(r)} l,$$

where for an atom $a \in \mathcal{A}(\Pi)$, we let $\Pi_a = \{r \in \Pi \mid H(r) = a\}$.

For convenience reasons, we also introduce $\mathrm{Clark}(\Pi)$ as the CNF

$$\bigwedge_{a \in \mathcal{A}(\Pi)} \left(\neg a \vee \bigvee_{r \in \Pi_a} \mathit{forced}_r \wedge \bigwedge_{r \in \Pi_a} a \vee \neg \mathit{forced}_r \right) \wedge$$
$$\bigwedge_{r \in \Pi} \left(\mathit{forced}_r \vee \bigvee_{l \in B(r)} \neg l \wedge \bigwedge_{l \in B(r)} \neg \mathit{forced}_r \vee l \right),$$

where forced_r for $r \in \Pi$ is an auxiliary variable such that intuitively forced_r is true iff every literal in the body of r is satisfied, which is ensured by the second line.

Formally, we obtain:

Lemma 12 *(Folklore). Let Π be an answer set program. Then, (i) every model $\mathcal{I}$ of $\mathrm{Clark}_{\mathrm{Prop}}(\Pi)$ can be uniquely extended to a model $\mathcal{I}'$ of $\mathrm{Clark}(\Pi)$ by assigning forced_r to true, if $\mathcal{I}' \models B(r)$ and to false, otherwise, and (ii) for every model $\mathcal{I}$ of $\mathrm{Clark}(\Pi)$ the interpretation $\mathcal{I}' = \mathcal{I} \cap \mathcal{A}(\Pi)$ is a model of $\mathrm{Clark}_{\mathrm{Prop}}(\Pi)$.*

The way we use Clark's Completion is to translate programs into equivalent propositional formulas. The following gives a sufficient condition for this to work.

Theorem 13 *([54]). Let Π be a normal answer set program. If the dependency graph $\mathrm{DEP}(\Pi)$ of Π is acyclic, then every model of $\mathrm{Clark}_{\mathrm{Prop}}(\Pi)$ is an answer set of Π and vice versa.*

6.1. *Primal tree decomposition guidance*

For now, we disregard the acyclicity requirement of Clark's Completion and consider only the change in treewidth as a result of applying Clark's Completion. Unfortunately, as the following example shows, we cannot give treewidth guarantees for $\mathrm{Clark}(\Pi)$ in terms of the treewidth of Π.

Example 8 *(Treewidth Bounds I).* Consider the program Π_n consisting of rules

$$r_1 : \ a \leftarrow b_1, \dots, r_n : \ a \leftarrow b_n$$

The treewidth of $\mathrm{PRIM}(\Pi_n)$ is 1, since the only edges are for the form (a, b_i), which means that $\mathrm{PRIM}(\Pi_n)$ is a tree. On the other hand, the CNF version of Clark's Completion of Π_n contains the following clauses to model the truth of the atom a:

$$\neg a \vee \bigvee_{j=1}^{n} \mathit{forced}_{r_j}, \qquad a \vee \neg \mathit{forced}_{r_i}, \qquad \text{for } i = 1, \dots, n$$
$$\mathit{forced}_{r_i} \vee \neg b_i, \qquad \neg \mathit{forced}_{r_i} \vee b_i, \qquad \text{for } i = 1, \dots, n.$$

Due to the clause $\neg a \vee \bigvee_{i=1}^{n} \mathit{forced}_{r_i}$, the primal graph of $\mathrm{Clark}(\Pi_n)$ contains a clique of size $n + 1$. Therefore, the treewidth of $\mathrm{Clark}(\Pi_n)$ is n.

This is a problem, if we want to use the treewidth of the CNF we obtain from Clark's Completion to bound the time needed to solve an AASC instance. However, this problem can be avoided by introducing further auxiliary variables that intuitively save partial results and thus limit the dependence of the variables among each other.

Example 9 *(cont.)*. Consider instead of the set of clauses from Example 8 the following alternative clauses:

$$\neg a \vee part_n, \qquad a \vee \neg part_n \tag{6}$$

$$\neg part_i \vee b_i \vee part_{i-1}, \qquad part_i \vee \neg b_i, \qquad part_i \vee \neg part_{i-1} \qquad \text{for } i = 1, \ldots, n \tag{7}$$

$$\neg part_0. \tag{8}$$

Here, $part_i$ for $i = 0, 1, \ldots, n$ are auxiliary variables. Intuitively, $part_i$ is responsible for storing a partial result. Namely, $part_i$ is true iff one of $b_1, \ldots, b_i$ is true and consequently there is a reason to derive a from one of the first i rules. This is ensured by the clauses in (8) and (7). Indeed, as $part_0$ is false, so $part_1 \leftrightarrow b_1$ holds as well as $part_i \leftrightarrow part_{i-1} \vee b_i$ for $i = 1, \ldots, n$. Finally, the clauses in (6) ensure that a is true iff any of the (first n) rules fires by ensuring that it a is equivalent to $part_n$.

While the primal graph of this CNF has treewidth 2 which is higher than the one of $\mathrm{PRIM}(\Pi_n)$ that has treewidth 1, it is independent of n, in contrast to the treewidth of $\mathrm{Clark}(\Pi_n)$.

The idea of the previous example is formalized as $\mathrm{PClark}(\Pi, \mathcal{T}, t_r)$ a revised definition of $\mathrm{Clark}_{\mathrm{Prop}}(\Pi)$ given by Hecher [55] that uses a tree decomposition $\mathcal{T}$ with root t_r of the primal graph of Π for guiding the Clark Completion. It leads to the following guarantee:

Theorem 14 *([55]). Given a normal answer set program* Π *and a tree decomposition* $\mathcal{T} = (T, \chi)$ *of* $\mathrm{PRIM}(\Pi)$ *with width* k *and root* t_r*, the CNF* $\mathrm{PClark}(\Pi, \mathcal{T}, t_r)$ *can be constructed in time linear in* $|\Pi| + |T|$ *and satisfies that*

(i) *every model of* $\mathrm{Clark}_{\mathrm{Prop}}(\Pi)$ *can be uniquely extended to a model of* $\mathrm{PClark}(\Pi, \mathcal{T}, t_r)$*,*
(ii) *the size of* $\mathrm{PClark}(\Pi, \mathcal{T})$ *is in* $\mathcal{O}(k|\Pi|)$*, and*
(iii) *the treewidth of* $\mathrm{PRIM}(\mathrm{PClark}(\Pi, \mathcal{T}, t_r))$ *is at most* $3(k+1)$.

Intuitively, we can avoid a higher increase in treewidth by avoiding long clauses introduced by large disjunctions $a_1 \vee \cdots \vee a_n$. For this, we apply the Tseitin transformation to obtain multiple shorter clauses that correspond to $a_1 \vee a_2 \vee part_1, part_1 \leftrightarrow a_3 \vee part_2, \ldots, part_{n-3} \leftrightarrow a_{n-1} \vee a_n$.

We do not spell out PClark(.), as we will give a further improved version. Notably, Clark's Completion may also introduce large conjunctions, which we also wish to avoid if they artificially increase the treewidth.

6.2. Incidence tree decomposition guidance

We have seen that we can use a tree decomposition of the primal graph of a program to limit the increase in the treewidth during translation to CNF that is caused by many rules that have the same atom in the head. Another factor that can lead to artificially high treewidth are long rules:

Example 10 *(Treewidth Bounds II)*. Consider the program Π_n with the single rule

$$r: \; a \leftarrow b_1, \ldots, b_n.$$

The treewidth of $\mathrm{PRIM}(\Pi_n)$ is n, since it is a clique on $n+1$ vertices. Thus, Theorem 14 guarantees us a CNF of treewidth at most $3(n+1)$. In this case, this upper bound is not helpful, since already the usual version of Clark's Completion results in the following clauses to model the truth of atom a:

$$a \vee \neg forced_r, \qquad \neg a \vee forced_r,$$

$$forced_r \vee \bigvee_{i=1}^{n} \neg b_i, \qquad \bigwedge_{i=1}^{n} \left(\neg forced_r \vee b_i\right).$$

The primal graph of this CNF also has treewidth n, which is better than our upper bound.

We can avoid the long clauses by using the following propositional formula instead:

$$a \leftrightarrow b_1 \wedge part_1,$$

$$part_i \leftrightarrow b_{i+1} \wedge part_{i+1}, \qquad \text{for } i = 1, \ldots, n-3,$$

$$part_{n-2} \leftrightarrow b_{n-1} \wedge b_n.$$

The treewidth of the primal graph of the corresponding CNF is 2 and therefore independent of n.

As previously, we see that we can decrease the treewidth of the CNF resulting from Clark's Completion by introducing auxiliary variables. The difference here is that we introduce them for conjunctions instead of disjunctions. Unfortunately, as we have also seen

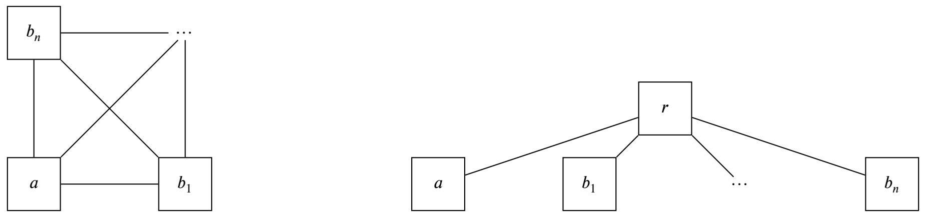


Fig. 4. The primal graph (left) and the incidence graph of the program Π_n from Example 11.

in the example, a tree decomposition of the primal graph of the program is unlikely to help us: it also assigns a high treewidth to the program even if has a simple representation.

We overcome this problem by considering the incidence graph of the program instead of the primal graph.

Example 11 *(cont.)*. The incidence graph of Π_n in Example 10 is shown in Fig. 4; as it is a tree, it has treewidth 1.

Definition 15 *(Incidence Tree Decomposition-guided Clark's Completion)*. Let Π be a normal answer set program and $\mathcal{T} = (T, \chi)$ a tree decomposition of $\mathrm{INC}(\Pi)$ with root t_r. Then, $\mathrm{IClark}(\Pi, \mathcal{T}, t_r)$ the *Incidence Tree Decomposition-guided* Clark's Completion of Π with respect to (T, χ) and root t_r is defined as the propositional formula

$$upto_t^a \leftrightarrow \bigvee_{t' \in children(t), a \in \chi(t')} upto_{t'}^a \vee \bigvee_{r \in \chi(t), H(r)=a} forced_r \qquad \text{for } t \in T, a \in \chi(t) \tag{9}$$

$$a \leftrightarrow upto_{t'}^a \qquad \text{for } t, t' \in T, a \in \chi(t') \setminus \chi(t), t' \in children(t) \tag{10}$$

$$a \leftrightarrow upto_{t_r}^a \qquad \text{for } a \in \chi(t_r) \tag{11}$$

$$upto_t^r \leftrightarrow \bigwedge_{t' \in children(t), r \in \chi(t')} upto_{t'}^r \wedge \bigwedge_{b \in \chi(t), b \in B(r)} b \qquad \text{for } t \in T, r \in \chi(t) \tag{12}$$

$$forced_r \leftrightarrow upto_{t'}^r \qquad \text{for } t, t' \in T, r \in \chi(t') \setminus \chi(t), t' \in children(t) \tag{13}$$

$$forced_r \leftrightarrow upto_{t_r}^r \qquad \text{for } r \in \chi(t_r) \tag{14}$$

The basic intuition here is the same as with the original CNF version of Clark's Completion, i.e. we introduce additional variables that are defined as equivalent to subformulas. The variables of the form $upto_t^a$ for $t \in T$ and $a \in \mathcal{A}(\Pi)$ intuitively capture whether a is derived by any of the rules that occur in $\chi(t)$ or $\chi(t')$ for a descendant t' of t. This is ensured in (9). Then, in (10) and (11) we ensure that the atom a holds if it was derived by any rule, by checking whether it was derived in the bag of the tree decomposition that contains it and is closest to the root (which may be the root). Similarly, the atoms of the form $upto_t^r$ for $t \in T$ and $r \in \Pi$ intuitively capture whether all the body atoms of r that occur in $\chi(t)$ or $\chi(t')$ for all descendants t' of t are satisfied due to (12). Accordingly, in (13) and (14) we ensure that $forced_r$ holds iff all the body atoms of r are satisfied, by checking whether $upto_t^r$ holds in the unique bag $t \in T$ such that $r \in \chi(t)$ and t does not have any ancestors that contain r.

This leads to the following result as desired:

Theorem 16. *Given a normal answer set program Π and a tree decomposition $\mathcal{T} = (T, \chi)$ of $\mathrm{INC}(\Pi)$ with width k and root t_r, the CNF $\mathrm{IClark}(\Pi, \mathcal{T}, t_r)$ can be constructed in time linear in $|\Pi| + |T|$ and satisfies that*

(i) *every model of $\mathrm{Clark}_{\mathrm{Prop}}(\Pi)$ can be uniquely extended to a model of $\mathrm{IClark}(\Pi, \mathcal{T}, t_r)$,*
(ii) *the size of $\mathrm{IClark}(\Pi, \mathcal{T})$ is in $\mathcal{O}(k|\Pi|)$, and*
(iii) *the treewidth of $\mathrm{PRIM}(\mathrm{IClark}(\Pi, \mathcal{T}, t_r))$ is at most $3(k+1)$.*

Proof (sketch). Roughly, we construct a new tree decomposition by adding for each node $t \in T$ with vertices $\chi(t)$ the atoms $upto_t^x$ for each $x \in \chi(t)$ and $forced_r$ for $r \in \chi(t)$. Then, we are interested in the cardinality of the set

$$(\chi(t) \setminus \{r \mid r \in \chi(t) \cap \Pi\}) \cup \{forced_r \mid r \in \chi(t) \cap \Pi\} \cup \{upto_t^x \mid x \in \chi(t)\},$$

which is $2(k+1)$. However, this does not cover the clauses in (12) and (9). We can assume w.l.o.g. that every node $t \in T$ has at most 2 children (e.g. by introducing duplicates of t) and add $upto_{t'}^x$ for every child t' of t to naively achieve an upper bound of $4(k+1)$. By instead using in the new tree decomposition copies $(t, 1), \dots, (t, |\chi(t)| + 1)$ of t for each $t \in T$ and processing the clauses in (12) and (9) one by one, we achieve the upper bound of $3(k+1)$. Assuming $\chi(t) = \{x_1, \dots, x_{|\chi(t)|}\}$, we start with the bag $(t, |\chi(t)| + 1)$ that contains $upto_t^{x_{|\chi(t)|}}, x, upto_{t_1}^x$, and $upto_{t_2}^x$ for the children t_1, t_2 of t and all $x \in \chi(t)$. Then, we define $\chi'(t, i)$ for $i = 1, \dots, |\chi(t)|$ as

$$\{upto_t^{x_{i-1}}\} \cup \chi'(t, i+1) \setminus \{upto_t^{x_i}, upto_{t_1}^{x_i}, upto_{t_2}^{x_i}\}.$$

This means that we start with bag $(t, |\chi(t)| + 1)$ of size $3(k+1)+1$ and decrease the size of the following bags by 2 in each step, resulting in a width of at most $3(k+1)$. □

Recall that a graph of primal treewidth k has incidence treewidth less or equal to $k+1$. Thus, in the worst case the guarantees of Theorem 16 only differ from those of Theorem 14 by a constant factor. However, since incidence treewidth may be arbitrarily smaller than primal treewidth, the guarantees with respect to incidence treewidth can be much better. The latter can also be seen in Example 11, where we consider a family of programs with constant incidence treewidth but unbounded primal treewidth. Thus, Theorem 16 broadens the range of cases, where we can fruitfully use CNF-translation followed by algebraic model counting on the CNF for algebraic answer set counting.

In order to solve AASC with knowledge compilation via a version of Clark's Completion, we however first need to ensure that the resulting CNF preserves the models of the original program bijectively.

7. Cycle breaking

Recall that Clark's Completion is only guaranteed to capture the answer sets of a program if the program is acyclic. Thus, for a program with a cyclic dependency graph, we first need to make some further effort to ensure that Clark's Completion will be correct. This process is commonly referred to as cycle breaking and transforms a program Π into a program $C(\Pi)$.

There has been a lot of work on cycle breaking both in the ASP community [59,57,55] for translations to SAT as well in the probabilistic reasoning community [58] specifically for weighted model counting. The basic idea of cycle breaking is to modify the program in such a way that afterwards the models of the Clark Completion and the modified program are the same. In the context of general AASC, it is moreover important that the answer sets before and after cycle breaking are in a specific one-to-one relationship. Formally, the cycle breaking needs to be faithful:

Definition 17 *(Faithfulness).* A cycle breaking $C(.)$ is *faithful* (for Π), if:

(i) $|\mathcal{AS}(\Pi)| = |\mathcal{AS}(C(\Pi))|$, and
(ii) $\mathcal{AS}(\Pi) = \{\mathcal{I} \cap \mathcal{A}(\Pi) \mid \mathcal{I} \in \mathcal{AS}(C(\Pi))\}$.

As expected, faithfulness guarantees that we can perform AASC over the program obtained by cycle breaking without changing the result.

Lemma 18 *(Faithfulness Implies Query Invariance). Let $\mu = \langle \Pi, \alpha, \mathcal{R}\rangle$ be a measure and let $C(.)$ be a faithful cycle breaking for Π. Then, for $\mu' = \langle C(\Pi), \alpha, \mathcal{R}\rangle$ it holds that $\mu(a) = \mu'(a)$ for every $a \in \mathcal{A}(\Pi)$.*

The translations of Lin and Zhao [59] and Hecher [55] do not satisfy this requirement for all program, as they serve to check the existence of an answer set.

In contrast, the two cycle breakings of Janhunen and Niemelä [56] and Mantadelis and Janssens [58] are faithful for all programs. In fact, both have been considered in the context of probabilistic reasoning. Janhunen and Niemelä's was considered for the implementation of Problog (cf. [36]) and Mantadelis and Janssens's is still part of the standard Problog implementation.

In this section, our aim is to find a cycle breaking that will result in good AASC-performance. As mentioned before, our main criterion here is whether we can bound the increase of the treewidth. However, of course also the increase in program size and insights regarding the semantic complexity are relevant for us. In the following, we thus first analyze the state of the art of faithful cycle breakings. Afterwards we introduce a novel cycle breaking called $\mathrm{T}_{\mathcal{P}}$-Unfolding, which we use in our implementation, and argue that it has favorable properties.

7.1. Necessity of cycle breaking

Before we consider existing cycle breakings, we first motivate the need for it by giving an example program, where the models of Clark's Completion do not align with the stable models.

Example 12. Consider again the program Π_{sm} from the running example.

$$\begin{array}{lll} \{\mathrm{st}(1)\} \leftarrow & \{\mathrm{st}(2)\} \leftarrow & \{\mathrm{st}(3)\} \leftarrow \\ \{\mathrm{inf}(3,1)\} \leftarrow & \{\mathrm{inf}(1,2)\} \leftarrow & \{\mathrm{inf}(2,3)\} \leftarrow \end{array}$$

$$\begin{array}{ll} \mathrm{sm}(1) \leftarrow \mathrm{st}(1) & \mathrm{sm}(1) \leftarrow \mathrm{inf}(3,1), \mathrm{sm}(3) \\ \mathrm{sm}(2) \leftarrow \mathrm{st}(2) & \mathrm{sm}(2) \leftarrow \mathrm{inf}(1,2), \mathrm{sm}(1) \\ \mathrm{sm}(3) \leftarrow \mathrm{st}(3) & \mathrm{sm}(3) \leftarrow \mathrm{inf}(2,3), \mathrm{sm}(2) \end{array}$$

Recall that $\{a\} \leftarrow$ is a shorthand for the two rules $a \leftarrow \mathrm{not}\ na$ and $na \leftarrow \mathrm{not}\ a$. Thus, each of the stress and influence atoms contribute the following equivalences to Clark's Completion, respectively:

$$\mathrm{st}(i) \leftrightarrow \neg\mathrm{nst}(i) \qquad \mathrm{ninf}(i,j) \leftrightarrow \neg\mathrm{inf}(i,j)$$
$$\mathrm{nst}(i) \leftrightarrow \neg\mathrm{st}(i) \qquad \mathrm{inf}(i,j) \leftrightarrow \neg\mathrm{ninf}(i,j)$$

Without any additional constraints, this means that we can choose the truth values of $\mathrm{st}(i)$ and $\mathrm{inf}(i,j)$ arbitrarily as long we assign $\mathrm{nst}(i)$ and $\mathrm{ninf}(i,j)$ the negation, respectively. Since this is the intended behavior of choice constraints, we see that Clark's Completion works as expected here. On the other hand, for the smokes atoms, we obtain the following equivalences:

$$\mathrm{sm}(i) \leftrightarrow \mathrm{st}(i) \vee (\mathrm{inf}(j,i) \wedge \mathrm{sm}(j)),$$

where $i+1 \equiv j \mod 3$. I.e. i smokes iff i is stressed or j influences i and j smokes. Under the stable model semantics, we can only derive that any person smokes if at least one person is stressed. However, for Clark's Completion there is a model, where every person smokes but nobody is stressed, namely:

$$\{\mathrm{sm}(1), \mathrm{sm}(2), \mathrm{sm}(3), \mathrm{nst}(1), \mathrm{nst}(2), \mathrm{nst}(3), \mathrm{inf}(3,1), \mathrm{inf}(1,2), \mathrm{inf}(2,3)\}.$$

While this is also a model of Π_{sm} it is not stable, since the strict subset

$$\{\mathrm{nst}(1), \mathrm{nst}(2), \mathrm{nst}(3), \mathrm{inf}(3,1), \mathrm{inf}(1,2), \mathrm{inf}(2,3)\}$$

is also a model of Π_{sm}.

The previous example clearly shows that the problem with Clark's Completion on general problem is caused by cyclic derivations. Cyclic derivations are those that use that an atom a is true to derive that a is true.

7.2. *The* MJ(.) *cycle breaking [58]*

The strategy of Mantadelis and Janssens's [58] cycle breaking MJ(.) to avoid such cyc'lic derivations is to introduce copies a_F of an atom a for $F \subseteq \mathcal{A}(\Pi)$ that intuitively capture derivations of a that do not positively use any atom from F. Then, we can use the atom $a_{\{b\}}$ in a derivation of b from a, even if a and b are in a cyclic dependency. Naturally, introducing a copy a_F for every subset $F \subseteq \mathcal{A}(\Pi)$ is undesirable as it would lead to an exponential number of atoms.

In order to avoid this whenever possible, the cycle breaking MJ(.) makes use of two ideas. Firstly, it uses the fact that sometimes the derivations for a that do not use atoms from F or F' align. Then, it does not make sense to introduce two separate atoms $a_F, a_{F'}$ and one can use one of them for both cases. Second, if every derivation for b uses the atoms in $F \subsetneq \mathcal{A}(\Pi)$ before using the atom a, then no atom $a_{\{b\}}$ will be created but only an atom $a_{F\cup\{b\}}$.

We illustrate the idea behind MJ(.) on our running example.

Example 13 *(cont.)*. First consider the rules for the guesses of the stressed and influences predicate.

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

Here, we do not need to change anything, since these atoms are not involved in any cycles. Next, we consider the remaining atoms in the rules

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

Here, $\mathrm{sm}(1), \mathrm{sm}(2)$ and $\mathrm{sm}(3)$ are in a cyclic dependency. We first consider $\mathrm{sm}(1)$. Due to the cycle, we cannot use all derivations of $\mathrm{sm}(3)$ to derive $\mathrm{sm}(1)$ using the rule

$$\mathrm{sm}(1) \leftarrow \mathrm{inf}(3,1), \mathrm{sm}(3).$$

Instead, we introduce a copy $\mathrm{sm}(3)_{\{\mathrm{sm}(1)\}}$ of the atom $\mathrm{sm}(3)$ that captures all derivations of $\mathrm{sm}(3)$ that do not make use of $\mathrm{sm}(1)$:

$$\mathrm{sm}(1) \leftarrow \mathrm{st}(1) \qquad \mathrm{sm}(1) \leftarrow \mathrm{inf}(3,1), \mathrm{sm}(3)_{\{\mathrm{sm}(1)\}}$$

Similarly, in order to capture all derivations of $\mathrm{sm}(3)$ from $\mathrm{inf}(2,3), \mathrm{sm}(2)$ using the rule

$$\mathrm{sm}(3) \leftarrow \mathrm{inf}(2,3), \mathrm{sm}(2)$$

we create a copy $\mathrm{sm}(2)_{\{\mathrm{sm}(1),\mathrm{sm}(3)\}}$ of the atom $\mathrm{sm}(2)$ that captures all derivations of $\mathrm{sm}(2)$ that use neither $\mathrm{sm}(1)$ nor $\mathrm{sm}(3)$:

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

Finally, to capture the derivations of $\mathrm{sm}(2)$ that captures all derivations of $\mathrm{sm}(2)$ that use neither $\mathrm{sm}(1)$ nor $\mathrm{sm}(3)$ we use the rule

$$\mathrm{sm}(2)_{\{\mathrm{sm}(1),\mathrm{sm}(3)\}} \leftarrow \mathrm{st}(2).$$

Here, we cannot use the rule

$$\mathrm{sm}(2) \leftarrow \mathrm{inf}(1,2), \mathrm{sm}(1)$$

since it would make use of $\mathrm{sm}(1)$.

The derivations for $\mathrm{sm}(2)$ and $\mathrm{sm}(3)$ are handled analogously to those of $\mathrm{sm}(1)$ using the following rules:

$$\begin{aligned}
\mathrm{sm}(2) &\leftarrow \mathrm{st}(2) & \mathrm{sm}(2) &\leftarrow \mathrm{inf}(1,2), \mathrm{sm}(1)_{\{\mathrm{sm}(2)\}}\\
\mathrm{sm}(1)_{\{\mathrm{sm}(2)\}} &\leftarrow \mathrm{st}(1) & \mathrm{sm}(1)_{\{\mathrm{sm}(2)\}} &\leftarrow \mathrm{inf}(3,1), \mathrm{sm}(3)_{\{\mathrm{sm}(2),\mathrm{sm}(1)\}}\\
\mathrm{sm}(3)_{\{\mathrm{sm}(2),\mathrm{sm}(1)\}} &\leftarrow \mathrm{st}(3) & &\\
\mathrm{sm}(3) &\leftarrow \mathrm{st}(3) & \mathrm{sm}(3) &\leftarrow \mathrm{inf}(2,3), \mathrm{sm}(2)_{\{\mathrm{sm}(3)\}}\\
\mathrm{sm}(2)_{\{\mathrm{sm}(3)\}} &\leftarrow \mathrm{st}(2) & \mathrm{sm}(2)_{\{\mathrm{sm}(3)\}} &\leftarrow \mathrm{inf}(1,2), \mathrm{sm}(1)_{\{\mathrm{sm}(3),\mathrm{sm}(2)\}}\\
\mathrm{sm}(1)_{\{\mathrm{sm}(3),\mathrm{sm}(2)\}} &\leftarrow \mathrm{st}(1) & &
\end{aligned}$$

Note that here we only used the second idea of Mantadelis and Janssens's cycle breaking, e.g. by not creating an atom $\mathrm{sm}(1)_{\{\mathrm{sm}(3)\}}$ since every derivation of $\mathrm{sm}(3)$ that uses $\mathrm{sm}(1)$ also uses $\mathrm{sm}(2)$. The first idea does not apply here, since the derivations for $\mathrm{sm}(1)$ without $\{\mathrm{sm}(2)\}$ and without $\{\mathrm{sm}(3), \mathrm{sm}(2)\}$ differ (and similarly for the derivations of $\mathrm{sm}(2), \mathrm{sm}(3)$).

This brings us to the question of how the treewidth changes when we apply this cycle breaking. The strategy in Theorem 16 was intuitively to add every copy of an atom to every bag in a tree decomposition that contains that atom. If we do the same with every copy a_F for an atom a and $F \subseteq \mathcal{A}(\Pi)$, we only get an upper-bound that is exponential in the number of atoms of Π. We may do slightly but not significantly better as the following two theorems show.

We first recall that a *simple cycle* in a directed graph G is a (directed) path in G from a vertex $v \in V(G)$ to v such that the only vertex that is visited twice is v.

Theorem 19. *Let Π be a normal answer set program of treewidth k such that the largest strongly connected component of $\mathrm{DEP}(\Pi)$ has size s and any strongly connected component of $\mathrm{DEP}(\Pi)$ has at most c directed simple cycles. Then, the treewidth of $\mathrm{MJ}(\Pi)$ is in $\mathcal{O}(k \cdot s \cdot c)$.*

As in Theorem 16, we add all copies of a to every bag that contains atom a to obtain the desired result.

Proof. Let (T, χ) be a tree decomposition of $\mathrm{PRIM}(\Pi)$. Then, (T, χ'), where

$$\chi'(t) = \chi(t) \cup \{a_F \mid F \subseteq \mathcal{A}(\Pi), a_F \in \mathcal{A}(\mathrm{MJ}(\Pi))\}$$

is a tree decomposition of $\mathrm{MJ}(\Pi)$. This follows from the fact that (T, χ) is a tree decomposition of the original program and since for every rule

$$r_c = a_F \leftarrow b_{1F_1}, \dots, b_{nF_n}, \mathrm{not}\ c_1, \dots, \mathrm{not}\ c_m$$

in $\mathrm{MJ}(\Pi)$ there is a rule

$$r = a \leftarrow b_1, \dots, b_n, \mathrm{not}\ c_1, \dots, \mathrm{not}\ c_m$$

in Π. From the definition of χ' it follows that $r_c \in \chi'(t)$ iff $r \in \chi(t)$. Therefore, every rule is contained completely in some bag as required. $\square$

This upper bound is not very helpful, since the strongly connected components of $\mathrm{DEP}(\Pi)$ can be large and can have exponentially many directed simple cycles in their size. Unfortunately, we cannot do much better.

Theorem 20. *There is a family of programs $(\Pi_n)_{n\in\mathbb{N}}$ such that*

1. *$\mathrm{DEP}(\Pi_n)$ has exactly one simple cycle,*
2. *the treewidth of Π_n is bounded by a constant (independent of n),*
3. *the number of atoms and rules of Π_n is linear in n, and*
4. *the treewidth of $\mathrm{MJ}(\Pi_n)$ grows linearly with n.*

Proof (Sketch). We define a family of programs as follows

$$\begin{aligned}\Pi_n = &\{\{v(i)\} \leftarrow \mid i = 1,\dots,n\} \cup \\ &\{\{e(i,j)\} \leftarrow \mid i,j = 1,\dots,n, i+1 \equiv j \mod n\} \cup \\ &\{in(i) \leftarrow v(i) \mid i = 1,\dots,n\} \cup \\ &\{in(i) \leftarrow e(i,j), in(j) \mid i,j = 1,\dots,n, i+1 \equiv j \mod n\}.\end{aligned}$$

Intuitively, Π_n takes the directed graph over n vertices with arcs

$$(1,2),(2,3),\dots,(n-1,n),(n,1),$$

thus inducing exactly one cycle $1,2,\dots,n,1$. Then, it guesses a random subset of its vertices ($v(i)$) and arcs ($e(i,j)$). All vertices are kept such that $v(i)$ holds or the edge of the predecessor is present $e(i,j)$ and the vertex j was kept. In other words, the latter means that all vertices on maximal paths of guessed edges, whose starting vertex was guessed are also kept.

1.-3. can be easily verified. To show 4., we show that $\mathrm{PRIM}(\mathrm{MJ}(\Pi_n))$ has a graph minor such that every vertex has at least degree n. From this it follows that the treewidth of $\mathrm{PRIM}(\mathrm{MJ}(\Pi_n))$ is at least n by employing standard results from the treewidth literature [94,73]. □

We see that the cycle breaking MJ(.) does not have desirable properties when it comes to treewidth.

7.3. The JN(.) *cycle breaking [56]*

We continue with the second cycle breaking JN(.) due to Janhunen and Niemelä [56]. The previous strategy was to capture partial derivations explicitly by introducing auxiliary atoms leading to an acyclic and answer set preserving program. The cycle breaking JN(.) introduced in this work instead leaves the original program untouched but adds constraints that ensure each true atom can be derived. For this, it makes use of level rankings [95]. Intuitively, the levels of atoms specify in which order they can be used in a derivation, i.e. when the level of atom a is lower than that of atom b, then we can use a rule that has b in the head and a in the body to derive b. It was shown that it is possible to characterize the answer sets of a program in terms of Clark's Completion in combination with level ranking constraints. Furthermore, by adding constraints on the level ranking we obtain a one-to-one correspondence [95].

We illustrate the idea behind JN(.) on our running example and refer the interested reader to [56] for details.

Example 14 *(cont.).* As already mentioned, we keep the original set of rules untouched. That is $\mathrm{JN}(\Pi) = \Pi \cup \Pi'$, where Π' contains additional rules that specify level ranking constraints.

The level ranking constraints for Π_{sm} are as follows. As before, the guesses of the stressed and influences predicates do not require additional rules. More generally, atoms a that can only be derived from rules whose positive body atoms are not in a cycle with a do not require level ranking constraints.

Thus, we only need to take care of the following rules:

$$\begin{aligned}&\mathrm{sm}(1) \leftarrow \mathrm{st}(1) &\qquad &\mathrm{sm}(1) \leftarrow \mathrm{inf}(3,1), \mathrm{sm}(3)\\ &\mathrm{sm}(2) \leftarrow \mathrm{st}(2) && \mathrm{sm}(2) \leftarrow \mathrm{inf}(1,2), \mathrm{sm}(1)\\ &\mathrm{sm}(3) \leftarrow \mathrm{st}(3) && \mathrm{sm}(3) \leftarrow \mathrm{inf}(2,3), \mathrm{sm}(2)\end{aligned}$$

For sm(1), this leads to the following additional rules:

$$\mathbf{just}(\mathrm{sm}(1)) \leftarrow \mathrm{st}(1) \tag{15}$$

$$\mathbf{just}(\mathrm{sm}(1)) \leftarrow \mathrm{inf}(3,1), \mathrm{sm}(3), \mathbf{lt}(\mathrm{sm}(3), \mathrm{sm}(1)) \tag{16}$$

$$\leftarrow \mathrm{sm}(1), \mathrm{not}\, \mathbf{just}(\mathrm{sm}(1)) \tag{17}$$

$$\mathbf{next}(\mathrm{sm}(1)) \leftarrow \mathrm{st}(1) \tag{18}$$

$$\mathbf{next}(\mathrm{sm}(1)) \leftarrow \mathrm{inf}(3,1), \mathrm{sm}(3), \mathbf{succ}(\mathrm{sm}(3), \mathrm{sm}(1)) \tag{19}$$

$$\leftarrow \mathrm{sm}(1), \mathrm{not}\, \mathbf{next}(\mathrm{sm}(1)) \tag{20}$$

Here, we introduce the following new atoms in order to model the level ranking constraints:

- $\mathbf{just}(\mathrm{sm}(1))$ means that sm(1) is justified, i.e. there is a rule that is applicable also when taking into account level rankings.
- $\mathbf{lt}(\mathrm{sm}(3), \mathrm{sm}(1))$ means that sm(3) is at a lower level than sm(1). In this case sm(3) can be used in a derivation of sm(1).
- $\mathbf{next}(\mathrm{sm}(1))$ means that sm(1) is at the next level compared to some atom that is used to derive it. i.e. there is a rule with sm(1) in the head and an atom in the body that is exactly one level lower than that of sm(1). E.g., in (19) we derive $\mathbf{next}(\mathrm{sm}(1))$ when we can derive sm(1) from inf(3, 1) and sm(3) and the level of sm(1) is equal to 1 plus the level of sm(3).

- **succ**(sm(3), sm(1)) means that the level of sm(3) is exactly one level lower than the level of sm(1).

The atoms of the form **just**(a) and **next**(a) are defined by the given rules. For the other auxiliary atoms of the form **succ**(b,a) and **lt**(b,a) we still need to add definitions based on the levels associated with the atoms a and b. To model the level of an atom a, we use atoms that act as a binary counter that represents the level. i.e. if the SCC of the dependency graph of the program that contains a has size n, we introduce atoms **bin**(a,i) that are guessed via rules $\{\mathbf{bin}(a,i)\} \leftarrow$ for $i = 0, \dots, \lceil \log_2(n) \rceil$. Then, for a given interpretation $\mathcal{I}$ of these atoms the level of a is given by

$$\sum_{i=0}^{\lceil \log_2(n) \rceil} \begin{cases} 2^i & \text{if } \mathbf{bin}(a,i) \in \mathcal{I}, \\ 0 & \text{otherwise.} \end{cases}$$

Using this correspondence, we can define the atoms **succ**(b,a) and **lt**(b,a) in terms of the binary counter atoms of a and b. Note that in order to obtain a faithful translation, we additionally need to ensure that the level of atoms is minimal, as otherwise it is not unique. For details, we refer to the original paper [56].

This brings us to the question of treewidth guarantees that we can give for JN(Π). In contrast to MJ(Π), we do not need information about the number of simple cycles for this.

Theorem 21. *Let Π be a normal answer set program of treewidth k such that the largest strongly connected component of* DEP(Π) *has size s. Then, the treewidth of* JN(Π) *is in $\mathcal{O}(k^2 + k\log_2(s))$.*

As with Theorem 16 the idea here is to take an existing tree decomposition of optimal width k and add to each bag all auxiliary atoms that are related to the atoms in the bag.

We point out that the quadratic dependency on the treewidth k can actually be avoided by modifying Janhunen and Niemelä's definition of cycle breaking. For this, observe that the quadratic dependency stems from the inclusion of **lt**(b,a) and **succ**(b,a) for every $a, b \in \chi(t)$ in the new bag $\chi'(t)$. If we instead define one atom **lt**(b,a,r) and **succ**(b,a,r) per rule r such that $head(r) = a$ and $b \in body(r)$, we do not need to include all such atoms in the same bag but can handle them in different bags. It follows that there is a cycle breaking with a treewidth upper bound in $\mathcal{O}(k\log_2(s))$. In fact, we observed that the implementation of JN(.) in LP2LP2 does exactly this, meaning that it comes with a treewidth upper bound of $\mathcal{O}(k\log_2(s))$.

We see that JN(.) gives better guarantees than MJ(.), since $k\log_2(s)$ grows slower asymptotically than $k \cdot s \cdot c$ even when $c = 1$. However, it uses a binary counter for the encoding of the level ranking. While this provides very good asymptotic size guarantees, both in terms of the resulting program and its treewidth, binary encodings can sometimes have negative effects on the practical performance of SAT solvers [96] and can result in much slower performance than encodings of a larger asymptotic size.

7.4. $\mathrm{T}_{\mathcal{P}}$*-unfolding*

We have seen that the current cycle breakings either come with weak upper bounds on the treewidth or have good upper bounds but use an encoding that can lead to impairments of the practical performance of SAT solvers and therefore also of knowledge compilers.

We want to avoid both drawbacks as far as possible. For this, we take a closer look at dependency graphs of programs and introduce cbs(G) a novel parameter, the *component-boosted backdoor size* of a digraph. Intuitively, cbs(.) (defined later in Definition 27) provides a more fine grained measure of directed cyclicity. More importantly, we can exploit it in $\mathrm{T}_{\mathcal{P}}$-Unfolding, our new cycle breaking, which leads to the following guarantees:

Theorem 22. *For any factorized measure $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$, we can construct in polynomial time in the size of Π given access to an* NP*-oracle a factorized measure $\mu' = \langle \Pi', \alpha, \mathcal{R} \rangle$ with an acyclic program Π' such that*

(i) *for all $a \in \mathcal{A}(\Pi)$ it holds that $\mu(a) = \mu'(a)$,*
(ii) *the treewidth of Π' is at most $k \cdot$* cbs(DEP(Π))*, where k is the treewidth of Π, and*
(iii) *the size of Π' is at most* cbs(DEP(Π)) $\cdot |\Pi|$.

Here, the increase of the treewidth and the program is bounded by cbs(DEP(Π)). Thus, this factor only depends on the dependency graph of Π and its cyclicity, rather than the number of atoms in the program or the size of the largest SCC of cbs(DEP(Π)).

As the name says, $\mathrm{T}_{\mathcal{P}}$-Unfolding is related to another approach for AASC called $\mathrm{T}_{\mathcal{P}}$-compilation [39]. Intuitively, the idea of $\mathrm{T}_{\mathcal{P}}$-compilation is to capture the derivations of atoms by iteratively compiling SDDs that are increasingly precise approximations thereof. For this, we take at each step all rules $r_1, \dots, r_m$ in Π such that $head(r_i) = a$ for some atom $a \in \mathcal{A}(\Pi)$. Then, we first conjoin the SDDs that represent the derivations of the body atoms of each rule r_i. By disjoining the resulting SDDs for each rule r_i, we get a better approximation of the derivations of a.

We use a similar idea to iteratively capture increasingly more derivations of atoms, but in a different way. Namely, we introduce at each step a new atom $a^{(i)}$ that represents the set of derivations of a that are currently captured. Intuitively, these new atoms $a^{(i)}$ are

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

similar to the atoms a_F introduced by Mantadelis and Janssens's cycle breaking, since both capture a subset of the derivations of a. However, the captured subsets of derivations are different, which allows us to obtain better theoretical guarantees for $\mathrm{T}_\mathcal{P}$-Unfolding.

$\mathrm{T}_\mathcal{P}$-Unfolding, which takes as input a program Π and an *unfolding sequence* $s \in \mathcal{A}(\Pi)^*$, which is a list of atoms $s_1 \dots s_n$, with $s_i \in \mathcal{A}(\Pi)$, is described in Algorithm 1. Intuitively, where the immediate consequence operator $\mathrm{T}_\mathcal{P}$[97] checks if an atom a follows from previously derived atoms, $\mathrm{T}_\mathcal{P}$-Unfolding introduces copies of all rules that derive a from previously considered atoms. For this, we iterate over the unfolding sequence $s = s_1 \dots s_n$, considering s_i at the i^{th} step (line 4). As the head atom of the rule-copies we introduce a new copy $s_i^{cnt(s_i)+1}$ or take the original atom s_i depending on whether s_i occurs again in $s_{i+1} \dots s_n$ (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).

We consider the effect of $\mathrm{T}_\mathcal{P}$-Unfolding on the program from our running example.

Example 15 *(cont'd).* First, recall that choice constraints $\{a\} \leftarrow$ are a shorthand for $a \leftarrow \text{not}\, na$ and $na \leftarrow \text{not}\, a$. We compute $\mathrm{T}_\mathcal{P}$-UNFOLD(Π_{sm}, s) using the unfolding sequence

$$\begin{aligned} s &= s^{guess} s^{smokes} \\ s^{guess} &= \text{st}(1)\text{nst}(1)\dots\text{st}(3)\text{nst}(3)\text{inf}(3,1)\text{ninf}(3,1)\dots\text{inf}(2,3)\text{ninf}(2,3) \\ s^{smokes} &= \text{sm}(1)\text{sm}(2)\text{sm}(3)\text{sm}(1)\text{sm}(2) \end{aligned}$$

and obtain:

$$\begin{array}{ccc} \{\text{st}(1)\} \leftarrow & \{\text{st}(2)\} \leftarrow & \{\text{st}(3)\} \leftarrow \\ \{\text{inf}(3,1)\} \leftarrow & \{\text{inf}(1,2)\} \leftarrow & \{\text{inf}(2,3)\} \leftarrow \end{array}$$

$$\begin{array}{ll} \text{sm}(1)^1 \leftarrow \text{st}(1) & \text{sm}(1)^1 \leftarrow \text{inf}(3,1), \bot \\ \text{sm}(2)^1 \leftarrow \text{st}(2) & \text{sm}(2)^1 \leftarrow \text{inf}(1,2), \text{sm}(1)^1 \\ \text{sm}(3) \leftarrow \text{st}(3) & \text{sm}(3) \leftarrow \text{inf}(2,3), \text{sm}(2)^1 \\ \text{sm}(1) \leftarrow \text{st}(1) & \text{sm}(1) \leftarrow \text{inf}(3,1), \text{sm}(3) \\ \text{sm}(2) \leftarrow \text{st}(2) & \text{sm}(2) \leftarrow \text{inf}(1,2), \text{sm}(1) \end{array}$$

We observe that $\mathrm{T}_\mathcal{P}$-UNFOLD($., s$) is faithful for Π_{sm}.

The output of $\mathrm{T}_\mathcal{P}$-UNFOLD(Π, s) is always an acyclic program, however, the faithfulness for Π depends on s. To ensure that $\mathrm{T}_\mathcal{P}$-UNFOLD($., s$) is faithful, it is enough to iterate over all 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 be sufficient 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 involving it: e.g. $\text{st}(i), i = 1, \dots, 3$ does not positively depend on any variable and can thus be considered once before all other variables and never again afterwards.

We give a sufficient condition for faithfulness. Notably, our condition abstracts away the actual program Π and is based on a structural property of the *digraph unfolding* of DEP(Π).

Definition 23 *(Digraph Unfolding).* Let G be a digraph and let $s \in V(G)^*$ be an unfolding sequence. Let $\text{cnt}(a, s) = |\{j \mid s_j = a\}|$ be the number of occurrences of a in the unfolding sequence s, then the unfolding UF(G, s) of G with respect to s is the digraph U such that

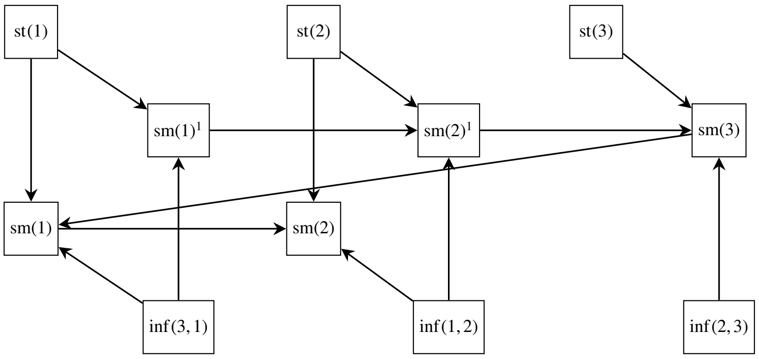


Fig. 5. Dependency Graph of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π_{sm}, s).

(i) $V(U) = \{a^i \mid 1 \leq i \leq \mathrm{cnt}(a, s)\}$, and
(ii) $(b^i, a^j) \in E(U)$ if $(b, a) \in E(G)$, $\mathrm{cnt}(a, s_1 \dots s_k) = j$ for some k and $\mathrm{cnt}(b, s_1 \dots s_k) = i > 0$.

The idea is that the digraph unfolding of DEP(Π) with respect to $s \in \mathcal{A}(\Pi)^*$ 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 24. *Let Π be an answer set program and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. Then,* UF(DEP(Π), s) = DEP($\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s)) *(when identifying a with $a^{\mathrm{cnt}(a,s)}$).*

Example 16 *(cont'd).* The dependency graph of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π_{sm}, s) is given in Fig. 5. It is acyclic and corresponds to UF(DEP(Π_{sm}), s) as expected.

Using Lemma 24, we can provide a sufficient condition for faithfulness.

Theorem 25. *Let Π be an answer set program and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. If for every simple directed path $\pi = (a_1, \dots, a_n)$ in* DEP(Π) *some directed path $\pi_c = (a_1^{c_1}, \dots, a_n^{c_n})$ in* UF(DEP(Π), s) *exists, then* $\mathrm{T}_{\mathcal{P}}$-UNFOLD(., s) *is faithful for* Π.

Proof (sketch). Let Π be some answer set program and let s be an unfolding sequence that satisfies the precondition of the theorem. Consider $\mathcal{I} \subseteq \mathcal{A}(\Pi)$. For any interpretation $\mathcal{I}_{ext}$ of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) such that $\mathcal{I}_{ext} \cap \mathcal{A}(\Pi) = \mathcal{I}$, it holds that the reduct $\mathrm{T}_{\mathcal{P}}$-UNFOLD$(\Pi, s)^{\mathcal{I}_{ext}}$ is equal to $\mathrm{T}_{\mathcal{P}}$-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 $\mathrm{T}_{\mathcal{P}}$-Unfolding 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 DEP(Π), we know there exists a corresponding SLD-tree in $\mathrm{T}_{\mathcal{P}}$-UNFOLD($\Pi^{\mathcal{I}}, s$). □

Note that we can prove a similar result for $\mathrm{T}_{\mathcal{P}}$-compilation, which reaches a fixed point iff $\mathrm{T}_{\mathcal{P}}$-UNFOLD(., s) is faithful for Π.

This theorem gives us a sufficient condition for an unfolding sequence s to lead to faithfulness of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(., s) with respect to a program Π. Notably, the condition does not depend on the actual program itself but only on its dependency graph and its unfolding with respect to s.

We are not only interested in faithfulness but we also care about the treewidth increase caused by unfolding. We can bound this increase as follows:

Lemma 26. *Let Π be an answer set program with treewidth k and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. If every variable $a \in \mathcal{A}(\Pi)$ occurs at most m times in s, then the treewidth of* $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) *is less or equal to $k \cdot m$.*

Proof (sketch). We know that during unfolding we introduce at most $m - 1$ copies $a_1, \dots, a_{m-1}$ of a variable $a \in \mathcal{A}(\Pi)$. Now, let (T, χ) be a tree decomposition for PRIM(Π) 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 $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s), and $|\chi'(t)| \leq |\chi(t)| \cdot m \leq k \cdot m$. □

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

There is another relevant observation that we need to keep in mind when it comes to knowledge compilation later on. While the guarantee generally only gives us an exponential upper bound in terms of treewidth, we actually know that this upper bound is too pessimistic. Consider the following scenario: if we introduce m copies $a, a_1, \dots, a_{m-1}$ of a variable a using $\mathrm{T}_{\mathcal{P}}$-Unfolding, then we have a monotonic relation between the truth values of the copies. That is, if a_i is true in a stable model $\mathcal{I}$ of the program and $j > i$, then also a_j must be true in $\mathcal{I}$. Thus, restricted to the variables $a, a_1, \dots, a_{m-1}$, there are actually at most m different assignments that can be extended to a stable model. It follows that for the variables in a bag $\chi'(t)$ of the tree decomposition that we constructed in the proof of Lemma 26 only $m^k = 2^{k \cdot \log_2(m)}$ different assignments can be extended to a stable model. We thus believe that knowledge compilation has better performance than Theorem 36 guarantees.

Motivated by Lemma 26 and Theorem 25, we say that an unfolding sequence s is a *path-preserving m-unfolding sequence* (for digraph G), if for every simple (directed) path $\pi = (a_1, \dots, a_n)$ in G there is a (directed) path $\pi_c = (a_1^{c_1}, \dots, a_n^{c_n})$ in $\mathrm{UF}(G, s)$ and every variable $a \in V(G)$ occurs at most m times in s. Naturally, we are interested in path-preserving m-unfolding sequences for small m.

Since also Lemma 26 does not access the actual program itself but only its dependency graph and the unfolding sequence, we can abstract away the program and search for path-preserving m unfolding sequences on digraphs directly. As we do not want to do a blind search, we instead consider backdoors [60] and generalize them to *component-boosted backdoors* into acyclicity because we can give guarantees for the maximum necessary m based on them. Backdoors have already been considered in the context of ASP [60]. Usually, the notion of a backdoor for a digraph G is a vertex set S, such that $G \setminus S$ satisfies some desirable property. For us this property is (directed or undirected) acyclicity of $G \setminus S$, since we can find a suitable unfolding sequence when given S in either case.

Both the backdoor size of a digraph and its generalization to component-boosted backdoor size intuitively measure the cyclicity of the given digraph by asking how many vertices need to be “cut out” in order for the remaining digraph to be (almost) acyclic. When a parameter value is low, an unfolding sequence s exists such that $\mathrm{T}_{\mathcal{P}}$-$\textsc{Unfold}(., s)$ is faithful and every variable occurs only a few times in s.

Definition 27 *(Backdoor, Component-boosted Backdoor).* Let G be a digraph. Then, $\mathrm{bs}(G)$, the *backdoor size* of G is

$$\min\{|S| \mid S \subseteq V(G), G \setminus S \text{ is a polyforest or acyclic}\},$$

where a polyforest is graph that has no undirected cycles.

Furthermore, $\mathrm{cbs}(G)$, the *component-boosted backdoor size* of G, is

(i) 1, if G is acyclic (which includes $V(G) = \emptyset$),
(ii) 2, if G is a polytree, i.e. a connected polyforest,
(iii) $\max\{\mathrm{cbs}(C) \mid C \in \mathrm{SCC}(G)\}$, if G is cyclic but not strongly connected,
(iv) $\min\{\mathrm{cbs}(G \setminus S) \cdot (|S| + 1) \mid S \subseteq V(G), S \neq \emptyset\}$ otherwise.

As the name suggests, *cbs* adds *component-boosting* to a specific variant of *backdoors*. A related parameter is elimination distance [98]. However, for elimination distance the removal set S in (iv) may only contain 1 element and results in cost $\mathrm{cbs}(G \setminus S) + 1$. Thus, the elimination distance to acyclic digraphs and polytrees is bounded iff $\mathrm{cbs}(.)$ is bounded, but naïvely, we can only assert that if the elimination distance is k then $\mathrm{cbs}(.)$ is less or equal to 2^k.

The main difference compared to the backdoor size is that component-boosted backdoor size additionally takes into account that $G \setminus S$ may consist of separate SCCs that can be handled recursively. We can therefore bound $\mathrm{cbs}(.)$ in terms of $\mathrm{bs}(.)$ as follows:

Lemma 28. *Let G be a digraph. Then,* $\mathrm{cbs}(G) \leq 2 \cdot (\mathrm{bs}(G) + 1)$.

Proof. If G is acyclic, then

$$\mathrm{cbs}(G) = 1 \leq 2 = 2 \cdot (0 + 1).$$

If G is a polytree, then

$$\mathrm{cbs}(G) = 2 \leq 2 = 2 \cdot (0 + 1).$$

If G is cyclic and strongly connected, let S^* be a set of vertices such that $|S^*| = \mathrm{bs}(G)$ and $G \setminus S^*$ is a polyforest or acyclic. Then

$$\begin{aligned}\mathrm{cbs}(G) &= \min\{\mathrm{cbs}(G \setminus S) \cdot (|S| + 1) \mid S \subseteq V(G), S \neq \emptyset\} \\ &\leq \mathrm{cbs}(G \setminus S^*) \cdot (|S^*| + 1) \\ &\leq 2 \cdot (\mathrm{bs}(G) + 1).\end{aligned}$$

If G is cyclic but not strongly connected, then

$$\mathrm{cbs}(G) = \max\{\mathrm{cbs}(C) \mid C \in \mathrm{SCC}(G)\}.$$

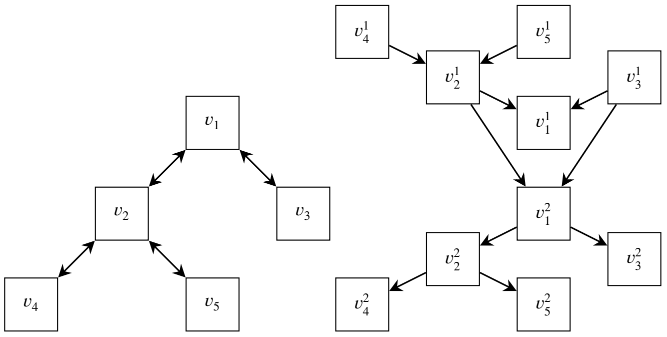


Fig. 6. A polytree G with root v_1 (left) and the unfolding $\mathrm{UF}(G, s_{post}s_{pre})$, where $s_{post} = v_4v_5v_2v_3v_1$, $s_{pre} = v_1v_3v_2v_5v_4$ (right).

Let S^* be a set of vertices such that $|S^*| = \mathrm{bs}(G)$ and $G \setminus S^*$ is a polyforest or acyclic. For each $C \in \mathrm{SCC}(G)$ it holds that $C \setminus S^*$ is a polyforest of acyclic, thus $\mathrm{bs}(C) \leq bsG$. From the fact that each C is either acyclic, a polytree, or cyclic and strongly connected, it follows by the previous three cases that $\mathrm{cbs}(C) \leq 2 \cdot (\mathrm{bs}(C) + 1) \leq 2 \cdot (\mathrm{bs}(G) + 1)$. □

Example 17 *(cont'd).* Consider the dependency graph $\mathrm{DEP}(\Pi_{sm})$ in Fig. 11a. It is strongly connected and not a polytree, therefore $\mathrm{cbs}(\mathrm{DEP}(\Pi_{sm}))$ is given by case (iv)

$$\min\{\mathrm{cbs}(\mathrm{DEP}(\Pi_{sm}) \setminus S)(|S| + 1) \mid S \subseteq V(\mathrm{DEP}(\Pi_{sm})), S \neq \emptyset\}.$$

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. It follows that $\mathrm{cbs}(\mathrm{DEP}(\Pi_{sm})) \leq \mathrm{cbs}(\mathrm{DEP}(\Pi_{sm}) \setminus S_i) \cdot (|S_i| + 1) = 2$. Since we need to remove at least one element, this is also a lower bound and hence $\mathrm{cbs}(\mathrm{DEP}(\Pi_{sm})) = 2$.

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

With the definition of the component-boosted backdoor size in mind, we state the main result of this section.

Theorem 22. *For any factorized measure $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$, we can construct in polynomial time in the size of Π given access to an NP-oracle a factorized measure $\mu' = \langle \Pi', \alpha, \mathcal{R} \rangle$ with an acyclic program Π' such that*

(i) *for all $a \in \mathcal{A}(\Pi)$ it holds that $\mu(a) = \mu'(a)$,*
(ii) *the treewidth of Π' is at most $k \cdot \mathrm{cbs}(\mathrm{DEP}(\Pi))$, where k is the treewidth of Π, and*
(iii) *the size of Π' is at most $\mathrm{cbs}(\mathrm{DEP}(\Pi)) \cdot |\Pi|$.*

Of course it is somewhat undesirable that we need access to an NP-oracle for the construction. However, this is unavoidable, since computing cbs(.) is NP-hard.

Theorem 29. *The problem of checking whether $\mathrm{cbs}(G) \leq k$ given a digraph G and $k \in \mathbb{N}$ in the input is NP-complete.*

Proof (Sketch). NP-membership is easy to see by a guess and check algorithm.

For NP-hardness, we use a reduction from SAT by constructing a digraph G such that its backdoor size is k iff the SAT instance is solvable. Then, we modify the digraph in such a manner that backdoor size and component-boosted backdoor size are equal to obtain the desired result. □

We will see later on that luckily the NP-hardness does not cause any problems in practice.

To prove Theorem 22, we show that every digraph G has some path-preserving $\mathrm{cbs}(G)$-unfolding sequence, using structural induction on the definition of cbs(.).

Lemma 30. *Let G be an acyclic digraph. Then, there exists a path-preserving 1-unfolding sequence. For case (i), we obtain:*

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 case (ii), where G is a polytree.

Lemma 31. *For every polytree G there exists a path-preserving 2-unfolding sequence s.*

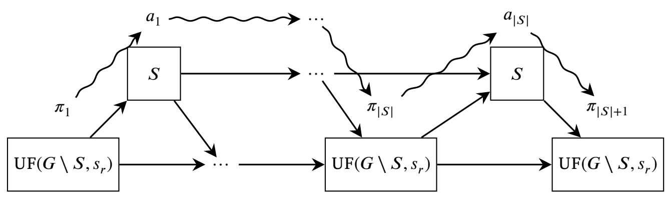


Fig. 7. Sketch of UF(G, s) and a path π through it, for the second recursive case, as in the proof of Lemma 33.

Proof. As G is a polytree, the corresponding undirected graph G^{tree} 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, the concatenation $s_{post}s_{pre}$ of s_{post} and s_{pre}, is a path-preserving 2-unfolding sequence of G, as depicted in Fig. 6. □

In case (iii), which is the first recursive one, we assume that 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 32. *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* cbs(C)-*unfolding sequence for C. Then, some path-preserving* $\mathrm{cbs}(G) = \max_{C \in \mathrm{SCC}(G)} \mathrm{cbs}(C)$*-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 $(C_1, \dots, C_n)$ of G^{con} to be given. Consider, $s = s_{C_1} \dots s_{C_n}$, the concatenation of the unfolding sequences for the SCCs in the chosen topological order. It is a path-preserving unfolding 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. Furthermore, 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 $\max_{C \in \mathrm{SCC}(G)} \mathrm{cbs}(C) = \mathrm{cbs}(G)$. □

Last but not least, we consider case (iv), 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. cbs$(G \setminus S)$, is small and handle S and $G \setminus S$ separately.

Lemma 33. *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 $S = \{a_1, \dots, a_{|S|}\}$. We define $s_S = a_1 \dots a_{|S|}$ and $s = (s_r s_S)^{|S|} s_r$, i.e. $s \in V(G)^*$ is the sequence obtained by iterating $|S|$ times over the sequence $s_r s_S$ and finally concatenating s_r. Then, s is an $m_r(|S| + 1)$-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 $\pi = \pi_1, a_{i_1}, \pi_2, a_{i_2}, \dots, a_{i_k}, \pi_k$, where π_i is a simple directed path in $G \setminus S$. Consider Fig. 7, 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 on. □

8. Implementation

In the previous sections, we discussed the general pipeline for solving AASC problems, and we considered for different aspects such as the knowledge compilation, the cycle breaking and the Clark Completion which options are available and which theoretical guarantees we can give for them. In the following, these theoretical considerations will serve as a basis for the choices we make when implementing our approach to solving AASC problems in our open-source solver aspmc.[4]

As other available solvers, aspmc follows the general pipeline given in Fig. 3. In contrast to some of them, aspmc does not skip any of its steps. In the following, we discuss the implementation aspects of each of the steps both technically and in relation to the theoretical results discussed above.

8.1. Input specification

For the inputs we do not allow general algebraic measures but restrict ourselves to the fragment of programs that can be specified similarly to ProbLog syntax. From an implementation perspective, it would also be easily possible to accept algebraic measures in their general form, however we did not deem this necessary, since Theorem 7 shows that this is not a restriction in terms of

[4] aspmc can be installed as a python package from https://pypi.org/project/aspmc/ for easy use and its source code can be downloaded from https://github.com/raki123/aspmc/ for development purposes.

expressivity. We chose this syntax, since it already well-known from ProbLog and allows ProbLog, aspmc, and PITA to be used interchangeably.

More formally, we allow programs consisting of rules that are either standard normal ASP rules, or rules of the form

$$r_1 :: a_1; \dots ; r_k :: a_k :\mp b_1, \dots, b_n, \text{not } c_1, \dots, \text{not } c_m. \tag{21}$$

Here, $r_1, \dots, r_k$ are semiring values over the specified semiring and all a_i, b_j, and c_l are atoms. By default, the probabilistic semiring $\mathcal{P}$ is assumed, other semirings can be given as an input argument, specifying either one of the standard semirings that is included in aspmc or a non-standard semiring that is specified as a python module, which is dynamically loaded based in its name.

A rule of the form (21) specifies that if the body is satisfied, then we can derive up to one of the atoms $a_1, \dots, a_k$. If we use the rule to derive a_i, then the rule contributes a factor of r_i to the weight of the model. If the body of the rule is satisfied but none of the atoms $a_1, \dots, a_k$ are derived by it, then the rule contributes a factor of $\text{negate}(r_1 \oplus \dots \oplus r_k)$ to the weight of the model. Here, negate is a function defined by the semiring module. E.g., for the probabilistic semiring it makes sense to define $\text{negate}(p)$ as $1 - p$ in order to ensure that the overall probability mass defined by a program is 1. Another generally applicable option is to specify $\text{negate}(r)$ as $e_\oplus$ to ensure that exactly one of the a_i is derived if the body of the rule is satisfied, since otherwise the weight of the corresponding model is zero.

Furthermore, atom queries are specified immediately within the program via statements of the form

$$query(a),$$

for an atom a. A program can contain multiple such statements, which then are each evaluated as the weight of all answer sets such that a holds.

We want to stress that the rule only contributes a factor r_i, if it was used to derive a_i, rather than if its body is satisfied and a_i is true. This has important implications, such as the following:

Example 18 *(Two Possible Derivations).* Consider the following program over the probability semiring $\mathcal{P}$:

$$0.5 :: a. \qquad 0.5 :: a. \qquad query(a).$$

This program specifies that there are two *independent* ways to derive a and we are interested in the weight of the answer sets such that a holds. We can derive a either only via the first rule, only via the second rule, via both rules or not at all. Each of these possibilities leads to an answer set with probability 0.25. Thus, the result of the query for a is 0.75. Clearly, the program

$$0.5 :: a. \qquad query(a),$$

would instead only lead to a result of 0.5.

This behavior may seem unintuitive at first glance, however, it makes much more sense when we interpret rules of the form (21) as a possible reason that can cause a_i. If the rule always contributed a weight r_i, when a_i and its body hold, then the program

$$a. \qquad 0.5 :: a. \qquad query(a),$$

would lead to a query result of 0.5 although clearly a has a 100% probability of being true in a randomly selected answer set. We stress that in aspmc the above program returns a 100% probability of a being true, as expected.

We allow non-ground programs, which may even have variables in the place of semiring values but do not go into the details here for simplicity.

In order to parse programs that are specified using the above syntax, we wrote our own parser in python using the parser generator lark [99], which is able to generate an LL parser on the fly given its specification as a grammar. The generation on the fly is important, as we need to incorporate parsing of semiring values dynamically based on the semiring we use.

8.2. Grounding & simplification

After parsing the possibly non-ground input specification, we need to ground the program in a first step. There exists a variety of grounding engines for programs already, for example, in ProbLog or clingo. Neither of them accept exactly the same kind of programs as we do, nevertheless, we were still able to exploit the grounding capabilities of clingo, instead of reinventing the wheel. For this, we restructure the parsed input as follows.

Standard ASP rules are kept as they are. A rule of the form

$$r = r_1 :: a_1; \dots ; r_k :: a_k :\mp b_1, \dots, b_n, \text{not } c_1, \dots, \text{not } c_m,$$

is transformed into the set of rules

$$guess(r) :\mp b_1, \dots, b_n, \text{not } c_1, \dots, \text{not } c_m.$$

$$a_1 :\mp b_1, \dots, b_n, \text{not } c_1, \dots, \text{not } c_m, \text{alg}(id, 1, \text{var}(r), a_1, r_1).$$

$$\ldots$$

$$a_k :\mp b_1, \ldots, b_n, \text{not } c_1, \ldots, \text{not } c_m, \text{alg}(id, k, \text{var}(r), a_k, r_k),$$

where $guess(r)$ is

$$\text{alg}(id, 1, \text{var}(r), a_1, r_1); \ldots ; \text{alg}(id, k, \text{var}(r), a_k, r_k); \text{alg}(id, k+1, \text{var}(r), n, n).$$

Here, an atom of the form $\text{alg}(id, i, \text{var}(r), a_i, r_i)$ for $i = 1, \ldots, k$ is true, when the rule with id id is used to derive the atom a_i with weight r_i, instantiated with the variables in $\text{var}(r)$. Thus, the first rule states for each of the instantiations of the variables $\text{var}(r)$ of r that if the body holds, then we may derive one of the atoms a_i. If $\text{alg}(id, k+1, \text{var}(r), n, n)$ is true, then none of the atoms are supposed to be derived.

The remaining rules then implement that if the body holds and the atom $\text{alg}(id, i, \text{var}(r), a_i, r_i)$ is set to true, then a_i is derived.

This rewriting is similar to the one that the ProbLog applies to rules of the form (21). If the semiring is the probabilistic semiring, we apply some further optimizations of the ProbLog solver in addition. Since they are described in detail in Appendix B of [100], we do not discuss them here and only note that they seem to be beneficial for performance but do not trivially generalize to other semirings.

Notably, clingo not only grounds programs but performs basic simplifications, such as removing atoms that are stated as facts from the bodies of rules. Apart from this, the only additional simplification that we perform after grounding is that we omit rules whose head atom occurs positively in the body. There are other simplifications that one could apply, such as the restriction of the program to its relevant ground part or the replacement of algebraic atoms a with weight $e_\oplus$ by the constraints $:\mp a.$. The former has so far only been considered for probabilistic reasoning and is implemented in ProbLog. We did not implement the restriction to the relevant ground part yet, however if this capability is desired, ProbLog can be used as a preprocessor. We leave generalized and possibly extended preprocessing for future work.

8.3. Cycle breaking

Recall that the three approaches to eliminate cycles that we considered in Section 7 have the following properties:

- MJ(.) is implemented in ProbLog, has a treewidth upper bound of $\mathcal{O}(k \cdot s \cdot c)$, where k is the original treewidth s is the size of the largest SCC of the dependency graph and c is the maximum number of simple cycles in an SCC of the dependency graph.
- JN(.), which has been considered for ProbLog [36] (an implementation is available online [101]) comes instead with a treewidth upper bound of $\mathcal{O}(k \log(s))$.
- Our $\text{T}_\mathcal{P}$-Unfolding algorithm has a treewidth upper bound of $\mathcal{O}(k \cdot \text{cbs}(\text{DEP}(\Pi)))$, where $\text{cbs}(.)$ is a structural parameter that intuitively measures the cyclicity of the dependency graph $\text{DEP}(\Pi)$ of the program Π.

Judging by the treewidth guarantee alone, it suggests itself to use JN(.), since its treewidth increase scales logarithmically with the size of the largest SCC. Thus, unless the program dependency graph $\text{DEP}(\Pi)$ is rather sparse, we expect that $\text{cbs}(\text{DEP}(\Pi))$ supersedes $\log_2(s)$. However, as we mentioned above, JN(.) uses binary counters in the encoding, which as well-known may impact negatively the performance during solving.

Furthermore, Fierens et al.'s [36] comparison of MJ(.) and JN(.) provides additional interesting aspects to take into consideration. They found that on small programs MJ(.) produces significantly smaller CNFs than JN(.), while JN(.) leads to much smaller CNFs on larger programs. This suggests that MJ(.) allows for faster inference on small programs. Furthermore, they found that only very few programs seem to be large enough to make JN(.) profitable, meaning that knowledge compilation is feasible and JN(.) yields better results. While the state of the art in knowledge compilation has seen significant improvements since 2011, we expect that both binary counters and significant size differences still will play a role in addition to treewidth.

This brings us to the $\text{T}_\mathcal{P}$-Unfolding approach. It is an improvement upon MJ(.) both in terms of the size of the encoding, which is at most quadratic with a small constant factor, and in terms of the treewidth, which is at most linear in the size of the largest SCC S even if S is fully connected. We thus expect that $\text{T}_\mathcal{P}$-Unfolding significantly outperforms both MJ(.) and JN(.), even when using the current state of the art knowledge compilers.

In order to perform $\text{T}_\mathcal{P}$-Unfolding by Algorithm 1, we need a path preserving unfolding sequence s though. We have shown that based on a small component-boosted backdoor, which is guaranteed to exist if $\text{cbs}(.)$ is small, we can compute a good such sequence s. However, computing this parameter exactly is NP-hard (see Theorem 29). While the backdoor size is also NP-hard to compute, there exist efficient solvers for it. Thus, we restrict ourselves to computing a (small) backdoor into polytrees for every SCC S of $\text{DEP}(\Pi)$, i.e. a subset $B \subset S$ such that no undirected cycle in $\text{DEP}(\Pi)$ lies in $S \setminus B$.

Note that a backdoor into polytrees is a Feedback Vertex Set (FVS). Computing a FVS is a well studied problem and there are many open source solvers available due to the 2016 edition of PACE [102]. We use the implementation [63] of Kiljan and Pilipczuk [103], who re-implemented many of the participating algorithms under a permissive license. We run the exact version of the solver using a fixed timeout of 30 seconds to try to obtain an optimal backdoor B. If the answer is not produced within the timeout, we run the heuristic version of the solver to obtain an approximation quickly.

8.4. Clark's completion

After performing $T_{\mathcal{P}}$-Unfolding, the resulting program is tight, i.e. its dependency graph is acyclic. Therefore, we can apply Clark's Completion on it to receive a CNF that we can then compile in the next step. Here, we again have three possible options:

- Clark(.), which does not admit any treewidth guarantees;
- PClark(.), i.e. Hecher's primal tree decomposition guidance, which admits a treewidth guarantee of $3(k+1)$ if k is the treewidth of the tree decomposition of PRIM(Π) in use;
- IClark(.), i.e. our novel incidence tree decomposition guidance, which admits a treewidth guarantee of $3(k+1)$ if k is the treewidth of the used tree decomposition of INC(Π) in use.

Based on the treewidth guarantees alone, it is suggestive to always use IClark(.). However, while it has the best worst case guarantees, it is not guaranteed that the obtained CNF always has the lowest treewidth out of the three. While the resulting treewidth cannot become much higher than that of a CNF obtained by Clark(.) or PClark(.), it may be as high or slightly higher and lead to a slightly larger CNF.

We therefore implemented all three versions of Clark's Completion and allow for a selection via input parameters, which we will use to compare the different strategies below.

In order to apply the tree decomposition guided versions PClark(.) and IClark(.) of Clark's Completion, we need access to tree decompositions of PRIM(Π) and INC(Π). Generating optimal tree decompositions is expensive but there exist several good heuristic solvers such as flow-cutter [104], tamaki-2017 [105], and htd [106]. We use flow-cutter since it performed very well in the PACE competition of 2017 regarding the heuristic computation of treewidth, achieving the best average approximation ratio and the best maximum approximation ratio of all solvers [107]. Furthermore, it has a permissive open source license that allows for combining it with closed source code such as c2d (version 2.20). Whenever we generate tree decompositions, we use a given timeout from the input. We wait until a tree decomposition has been found and the timeout has passed, whichever happens later. Then, we use the best tree decomposition found. This means that on large graphs we may supersede the given timeout.

After Clark's Completion, we extend the resulting CNF with annotations denoting the semiring in use, and the weights of literals over the semiring.

8.5. Knowledge compilation

For the knowledge compilation step, which we perform in an online manner, there is a wide range of tools available that compile to different tractable circuit representations, such as

- to BDDs, e.g., [108,44]
- to SDDs, e.g., [43], and
- to d-DNNFs, e.g., [46,53,109,45].

We focus on the compilers for d-DNNFs, as they tend to have the best worst-case guarantees in terms of treewidth. Furthermore, they also usually perform well on the related tasks of model counting and weighted model counting as seen in the Model Counting Competition 2020 [110].

More specifically, we implemented the knowledge compilation step for solvers that are recent or ranked high in the 2021 edition of the weighted model counting track of the Model Counting Competition 2021 [111]. We focus on this track, as we are interested in the performance of the actual compilation. The unweighted track could possibly have skewed results as preprocessing that is not applicable to weighted can lead to significant performance improvements. The solvers SHARPSAT-TD [47], d4 [53], and c2d [46] finished top three in the weighted model counting track, miniC2D [45] is a recent knowledge compiler, which is why we also included it.

Notably, whereas c2d, miniC2D, and d4 allow for knowledge compilation by default, this is not the case for SHARPSAT-TD, which was originally a solver conceived only for (weighted) model counting. Thus, we modified SHARPSAT-TD to enable its usage for knowledge compilation. This turned out to be rather easy: since SHARPSAT-TD was designed to be extensible to general semirings, we could exploit the idea of the arithmetic circuit semiring from Section 4.1. Specifically, in our implementation we used d-DNNF nodes as semiring elements and combined them in a conjunctive/disjunctive manner whenever SHARPSAT-TD multiplied/added two values. Not only does this lead to an algebraic circuit but to a d-DNNF, which is moreover smooth.[5]

Apart from SHARPSAT-TD, we also performed a minor modification in d4. Namely, by default d4 does not produce smooth d-DNNFs, which leads to an increase in the evaluation time. For this reason, we made minor modifications to the part of the source code of d4 that writes d-DNNFs, to ensure their smoothness d-DNNFs. Notably, the necessary information was already present meaning that we could keep our modifications to a minimum.[6] The source code of c2d and miniC2D were left unmodified.

[5] Source code of the modified version: https://github.com/raki123/sharpsat-td; For the original: https://github.com/Laakeri/sharpsat-td.

[6] Source code of the modified version: https://github.com/raki123/d4; For the original: https://github.com/crillab/d4.

The exact input arguments that we use for the different knowledge compilers can be found in Appendix C. Most settings are rather standard, however, we should mention that for c2d and miniC2D we supply custom dtrees and vtrees, respectively, which determine the order in which variables are decided during compilation.

We chose to generate custom dtrees and vtrees, since Korhonen and Järvisalo, [47] showed that the performance of c2d and miniC2D benefits from it. Namely, they used the fact that, for a given tree decomposition, we can generate dtrees and vtrees that give us a performance guarantee[7] of $\mathcal{O}(2^k \cdot k \cdot |C|^c)$, where k is the width of the tree decomposition, $|C|$ is the size of the CNF, and c is some constant. This guarantee can be achieved by first deciding all variables in the root of the tree decomposition and proceeding recursively for the children [47]. Both c2d and miniC2D were shown to solve significantly more instances using this strategy than when using the standard settings [47]. We provide details regarding how we generate dtrees and vtrees in Appendix D, which only insignificantly differs from Korhonen and Järvisalo's method. The tree decompositions used in this step are again generated using flow-cutter [104] using a timeout given in the input.

8.6. Evaluation

Given a (smooth) d-DNNF, the evaluation procedure is rather standard. We initialize the weights of the literals using the weight labels of our extended CNF format. Then, we parse the d-DNNF line by line and interpret it as an arithmetic circuit by combining the inputs to AND gates via multiplication and the inputs to OR gates via addition. The result we obtain at the root node of the d-DNNF then corresponds to the final result of our query. This step is well-known and well-understood, which is why we do not go into details here but refer the interested reader to [30].

The only additional comment in order regards the evaluation of non-smooth d-DNNFs as produced by miniC2D, since not all variables are guaranteed to occur in every branch of the d-DNNF, which can lead to wrong results during naive evaluation as described above. The only case in which a literal may not occur in a branch is if both values are accepted in the branch. E.g. for the program $\{a\} \leftarrow$ the empty NNF is a valid d-DNNF. If the variable a does not occur in a branch, we need to multiply the value of the branch by the sum of the weights of a and $\neg a$. Naively, this is easily possible by tracking the assigned and unassigned variables that occur below each node but costly. However, given that we know the vtree that the CNF was compiled with, we can do this more efficiently by keeping track of which vtree node a d-DNNF node belongs to and multiplying its weight by the corresponding factor when we notice that variables are missing.

9. Experiments

In order to evaluate the impact of our theoretical results in practice, we performed an extensive experimental evaluation. Furthermore, we compared the performance of our solver aspmc to state of the art solvers on probabilistic inference instances. All results and benchmarks can be found online.[8]

9.1. Questions & hypotheses

The first question we consider addresses the properties of the encodings that different cycle breakings result in.

Q1. CNF encodings How do the cycle breakings MJ(.), JN(.) and $\mathrm{T}_{\mathcal{P}}$-Unfolding(.) differ in terms of *size* and *treewidth (upper bound)* of the final CNF encoding?

We expect that $\mathrm{T}_{\mathcal{P}}$-Unfolding is almost always better than MJ(.) with respect to both size and treewidth (see Section 7.4). Furthermore, we expect that for small-sized and medium-sized programs, $\mathrm{T}_{\mathcal{P}}$-Unfolding results in lower CNF sizes than JN(.), but yields larger sizes for bigger programs due to the asymptotic guarantee. In terms of treewidth, we expect that JN(.) leads to lower treewidth unless the given program has only very minor cyclic positive dependencies, which means a very small component-boosted backdoor size.

Second, we are interested in finding the best configuration of options to use for aspmc. While we always use $\mathrm{T}_{\mathcal{P}}$-Unfolding for cycle breaking, we still have to decide which version of Clark's Completion to apply and which knowledge compiler to use.

Q2.1. Variants of Clark's completion Does tree decomposition-guidance of Clark's Completion provide a benefit compared to unguided Clark's Completion?

Q2.2. Effect of knowledge compiler Which knowledge compiler leads to the best AASC performance?

For Q2.1, we anticipate that tree decomposition-guidance can only decrease the performance, if the treewidth of the CNF obtained with guidance is the same or slightly higher than the one obtained without guidance. Even then, the decrease should be marginal. However, when the guidance has a positive impact on the treewidth of the resulting CNF, we naturally expect a performance increase.

[7] Strictly speaking, it is not clear whether this is only a guarantee for miniC2D but also for c2d at least in the way that we as well as Korhonen and Järvisalo generate dtrees and run c2d. This is because c2d has an optimization built in that skips deciding variables whose "turn" it would usually be, if these variables are not necessary to decompose the current CNF into separate components. This optimization can be turned off using the input option "force" of c2d. However, this seems to only degrade the overall performance.

[8] github.com/raki123/aspmc_benchmarks/tree/aspmc_results.

Regarding Q2.2, it seems reasonable to assume that SHARPSAT-TD performs best, followed by d4, and c2d since they were the top three solvers on the weighted model counting track of the Model Counting Competition 2021. miniC2D did not participate. With Q2.2 we want to find out, whether this translates to the same performance when we perform knowledge compilation first and then evaluate rather than performing weighted model counting immediately within the solver.

Last but not least, given the overall best configuration for AASC with aspmc from the results of the previous questions, we want to compare our solver with the state of the art software in probabilistic logic programming.

Q3. Overall performance Does aspmc provide a significant performance increase compared to the state of the art solvers for probabilistic logic programming?

We expect this to be the case, since the pipeline that uses knowledge compilation of CNFs comes with the best performance guarantees. Furthermore, we anticipate our cycle breaking and Clark's Completion to exhibit better solving capacity than other approaches that compile CNF to sd-DNNF. On top of that, we use the state of the art knowledge compilers for CNF to sd-DNNF, which have seen significant performance improvements in recent years.

9.2. Setup

Benchmark instances We use probabilistic logic programming instances for all questions, divided into four sets of instances:

- **Smokers** [35] For programs with varying levels of positive cyclic dependencies, we use the well-known smokers example (see Example 1) with varying extensions of the input predicates person and friend. We generated the benchmarks ourselves using the Barbàsi-Albert graph model [112], which is known to generate so called *scale-free* graphs that resemble typical networks in real life. For each vertex in the graph, we add one person and for each edge (v, v') we add $\mathtt{friend}(v, v')$ to the input. We used varying graph sizes ($n = 3, \dots, 49$) and vertex degrees ($m = 2, \dots, \min(n-1, 10)$) leading to graphs of varying density (namely, $(n-1) \cdot m$ edges).
- **Near Tree** Additionally, we introduce what we call *near tree* instances: a new cyclic benchmark set concerning reachability on almost trees. To generate the programs, we used as ingredients a directed graph G and the base program

$$r(s).$$
$$0.1 :: trap(Y) :\mp p(X,Y).$$
$$r(Y) :\mp p(X,Y).$$
$$1/d(X) :: p(X, s_1(X)); \dots ; 1/d(X) :: p(X, s_d(X)) :\mp r(X), \text{not}\, trap(X).$$

  Here, $d(X)$ is the number of outgoing arcs of X in G, and the vertices $s_1(X), \dots, s_d(X)$ are the immediate descendants. We obtain the final program by replacing the variables X, Y with constants corresponding to the vertices of G.

  This program models that we reach (denoted by $r(.)$) the starting vertex s and, at each vertex v that we reach, decide uniformly at random which outgoing arc we include in our path (denoted by $p(., .)$). If we include the arc (v, w), then we reach the vertex w. However, we only include an arc, if we do not get trapped (denoted by $trap(.)$) at v.

  As for the digraph G, we used two parameters $n, k \in \mathbb{N}$ to generate it as follows. We first generated a random tree of size n using the python library networkx, where each arc in the tree is bidirectional. Then, we added k vertices and connected them to each of the n original vertices in the tree bidirectionally. Finally, we added one vertex as the goal vertex, with incoming arcs from each of the k additional vertices. As the start s we use the root of the tree.

  These graphs are almost trees in the sense that (i) their treewidth is approximately $\min(k, n)$[9] and (ii) its component-boosted backdoor size is $2 \cdot k$, since we obtain a (poly)tree after removing the k added vertices.

  We use one instance for each combination of $n = 10, 20, \dots, 100$ and $k = 1, 2, \dots, 5$.
- **Growing Heads & Growing Negated Bodies** [85] The synthetic benchmark sets of growing heads (gh) and growing negated bodies (gnb) compare the difficulty of reasoning in the presence of long heads such as

  $$p_1 :: a_1; \dots ; p_i :: a_i :\mp a_{i+1},$$

  and long negated bodies

  $$0.5 :: a_i :\mp \text{not}\, a_{i+1}, \dots, \text{not}\, a_{n-1}, a_n.$$

  Here the length of the heads and negated bodies varies from 1 to 25 and 1 to 200 (after 50 only in steps of 10), respectively.

  Both gh and gnb only feature acyclic programs.
- **Blood** [85] The blood benchmark set consists of 100 instances, which compute probabilities of blood types given the blood types of ancestors. Also the blood benchmarks are all acyclic; they have vastly varying sizes but treewidth at most 10.

[9] Observe that every vertex in the graph has at least degree $\min(n, k)$, which is known to imply treewidth $\geq \min(n, k)$.

Benchmark platform We ran all solvers on a cluster of 12 nodes, each equipped with two Intel Xeon E5-2650 CPUs with 2.2 GHz and access to 256 GB shared RAM under Ubuntu 16.04.1 LTS powered on kernel 4.4.0-139 with no hyperthreading operating Python 3.7.6. Per instance, we set a memory limit of 32 GB and a time limit of 1800 secs on a single core.

Solvers compared We compare a variety of solvers from probabilistic logic programming, as well as different configurations of our own solver. Namely, we include

- **LP$^{\text{MLN}}$** [10] version 1.1: LP$^{\text{MLN}}$ performs probabilistic reasoning by enumeration of all stable models. plingo [38] is an alternative solver with the same strategy that however only exhibits slightly better performance [38].
- **ProbLog** [35] version 2.1.0.42: ProbLog uses the MJ(.) cycle breaking followed by knowledge compilation. It supports different circuit classes for compilation. We include compilation of CNF to sd-DNNF using either DSHARP [109] (denoted "+ Dsharp") or c2d (denoted "+ c2d") as knowledge compilers, and of the acyclic programs to SDD using PySDD [113,114] (denoted "-k sdd).
- **PITA** [13] version 4.5.0 using the Prolog engine swipl version 8.5.1: PITA compiles cyclic programs to BDD using the CuDD package version 3.0.0 [44].
- **LP2LP2** [56] version 1.23: LP2LP2 performs the JN(.) cycle breaking. The input options "-g" and "-l" ensure a bijective correspondence between stable models. Here, we do not perform probabilistic inference, as the weights get lost during cycle breaking. Instead we use SHARPSAT-TD afterwards to compile the resulting CNF into an sd-DNNF to count its models.
- **aspmc** (our solver) version 1.0.7. We always use the input option "-dt 10" to specify that tree decompositions should be computed with a timeout of ten seconds.

  To vary the version of Clark's Completion that we use, we specify "-g none", "-g ors", or "-g both" to use unguided, primal tree decomposition guided or incidence tree decomposition guided Clark's Completion.

  To specify the knowledge compiler we use "-k c2d", "-k miniC2D", "-k d4", or "-k sharpsat-td" to use c2d version 2.20, miniC2D version 1.0.0, d4 from https://github.com/raki123/d4, or our modified version of SHARPSAT-TD from https://github.com/raki123/sharpsat-td.

Comparisons

Q1. CNF encodings We produce a CNF using the different cycle breaking algorithms of aspmc, ProbLog, and LP2LP2 and extract the size of the encoding (i.e. number of clauses) as well as decent treewidth upper bounds from the first tree decomposition found by flow-cutter.

Notably, since the CNF encodings all use Clark's Completion, the differences in the size of the encoding and treewidth are due to the cycle breaking implemented by the solver. Therefore, we restrict our comparison here to the smokers and the near tree benchmark sets, which contains the instances where cycle breaking is necessary. Furthermore, we only consider instances whose number of clauses was less than 25000 and whose treewidth upper-bound was less than 200, which includes all solved instances apart from very few outliers.

Recall that the asymptotic size guarantees tell us that large enough instances JN(.) is better than $T_{\mathcal{P}}$-Unfolding, which in turn is better than MJ(.). We assume that the instances, where this kicks in, are so large that knowledge compilation is bound to fail, regardless of the encoding. Consequently, we also take into account the runtime of the instances such that we can establish the properties of the encoding on the instances relevant to us, i.e. those that are in principle solvable using the given cycle breaking.

Q2.1. Variants of Clark's completion Here, we want to ensure that effects on performance for different versions of Clark's Completion results from the exploitation of a lower treewidth in the decision heuristic of the knowledge compiler. Since d4's decision heuristic is not based on tree decompositions and SHARPSAT-TD uses a secondary heuristic in combination with the tree decomposition, there is no guarantee that variables are only decided based on the order entailed by the tree decomposition. On the other hand, for c2d and miniC2D, we always specify an input dtree/vtree such that the variables are decided in the order corresponding to the tree decomposition. Therefore, we ran aspmc with varying versions of Clark's Completion and c2d respectively miniC2D as knowledge compiler for this question.

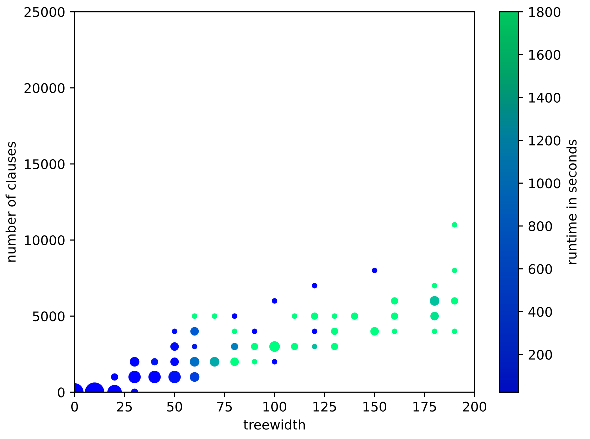
Q2.2. Effect of knowledge compiler To test the performance of the different knowledge compilers, we ran aspmc with the best performing version of Clark's Completion from Q2.1.

Q3. Overall performance For the general comparison, we include all solvers and configurations listed above, while for aspmc we use only the best performing Clark Completion.

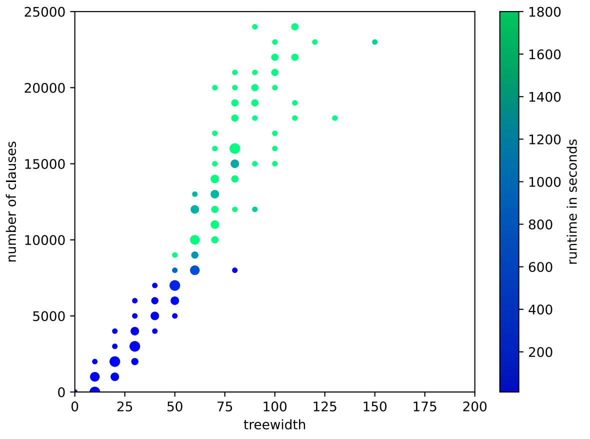
9.3. Results & discussion

Q1. CNF encodings The results for this experiment are shown in Fig. 8, which contains one XY-plot for $T_{\mathcal{P}}$-Unfolding, JN(.), and MJ(.) in Figs. 8a, 8b, and 8c, where the runtime is color-coded. We note that in almost all cases a light green data point not only indicates a high running time, close to 1800 seconds, but even that the instances represented by that data point all timed out.

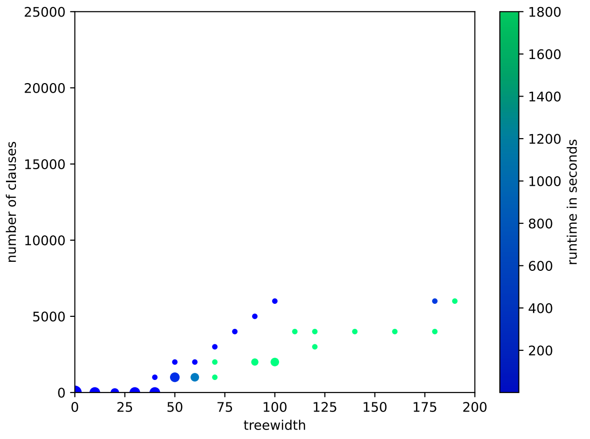
As expected, we see that for MJ(.) very few CNF encodings are in the range of 0 to 25000 clauses and 0 to 200 treewidth upper-bound compared to $T_{\mathcal{P}}$-Unfolding and JN(.), and even fewer are solvable within the time limit shown by the colors of the



8a. Numbers of clauses, treewidth upper bounds, and average runtime, when using $T_{\mathcal{P}}$-Unfolding (i.e. aspmc) for cycle breaking.



8b. Numbers of clauses, treewidth upper bounds, and average runtime, when using JN(.) (i.e. LP2LP2) for cycle breaking.



8c. Numbers of clauses, treewidth upper bounds, and average runtime, when using MJ(.) (i.e. ProbLog) for cycle breaking.

Fig. 8. The number of clauses, treewidth upper-bounds, and average runtimes of CNF encodings produced by different cycle breaking algorithms on the smokers and tree benchmark set. Restricted to instances, where the CNF encoding has less than 25000 clauses and a treewidth upper-bound less than 200. For each instance we round the number of clauses down to the next multiple of 1000 and round the treewidth upper-bound down to the next multiple of 10. We group the instances with the same (rounded) values and add one data point to the plot. Here, the size corresponds to the number of instances in the group and the color corresponds to the average runtime of the instances in the group. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

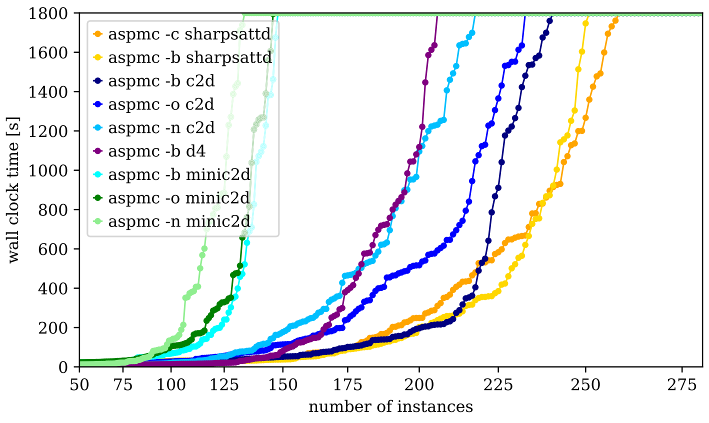


Fig. 9. Cactus plot of the results for selected aspmc configurations over all benchmark instances, ordered (top is better) by the number of solved instances.

data points. Both $T_\mathcal{P}$-Unfolding and JN(.) have significantly more CNF encodings within the plotted range including also many more solved instances. Interestingly however, the distribution varies significantly between $T_\mathcal{P}$-Unfolding and JN(.):

- The resulting CNFs from JN(.) often have many more clauses than those from $T_\mathcal{P}$-Unfolding. This matches the experimental results of Fierens et al. (2011), who made similar observations for MJ(.) and JN(.) on small instances. However, while for MJ(.) this effect only occurs for small instances, we observe this effect to be more consistent for $T_\mathcal{P}$-Unfolding. Here, the better asymptotic guarantees of JN(.) do not take effect on the instances that can be solved after $T_\mathcal{P}$-Unfolding or JN(.).
- The treewidth upper-bound is often lower for JN(.) than for $T_\mathcal{P}$-Unfolding. We thus suspect that while the constant factors in the size and treewidth guarantee that comes with $T_\mathcal{P}$-Unfolding are both small, only the constant factor for the treewidth guarantee that comes with JN(.) is small.
- For low treewidth upper-bounds and low numbers of clauses the instance density is higher for $T_\mathcal{P}$-Unfolding than for JN(.). This is again explained by small constant factors in the size and treewidth guarantees.
- $T_\mathcal{P}$-Unfolding leads to a solution for significantly more instances, namely 119 vs. 66 for JN(.). We especially highlight here that $T_\mathcal{P}$-Unfolding produces some CNF encodings with treewidth upper-bound above 100, which however are solved quickly (indicated by the blue color of the data points). This is not entirely surprising, since for $T_\mathcal{P}$-Unfolding we expected[10] the actual performance to supersede the guarantees entailed by treewidth. And, as we already mentioned, the binary counters that JN(.) uses may also have a negative impact on performance.

The results match the expectations that we had for the different cycle breakings. When MJ(.) results in a solution, then the CNF encoding is usually small. However, even in this case we may reach relatively high treewidth upper bounds and as a consequence, solve very few instances. JN(.) results in (relatively) large CNF encodings on which knowledge compilation is still feasible, presumably because it has considerably lower treewidth upper bounds than MJ(.) as guaranteed by Theorem 21. $T_\mathcal{P}$-Unfolding is somewhere in between: successfully evaluated instances can reach remarkable CNF sizes and still be solved. At the same time, they can have higher treewidth upper bounds than the ones coming from JN(.); combined with medium CNF sizes, this seems to be less of an issue than for the other encodings. We attribute this to the arguably lower hardness of the encoding obtained by $T_\mathcal{P}$-Unfolding compared to JN(.). While this results in a worse asymptotic scaling, it apparently leads to a smaller CNF size on instances that can still be solved by knowledge compilation in principle. We suspect that this in combination with the lower "semantic complexity" (i.e. lack of binary counters) is the cause for the performance improvement of $T_\mathcal{P}$-Unfolding compared to the other cycle breakings.

Q2.1. Variants of Clark's completion Recall that we compare the differences in performance caused by Clark(.), PClark(.), and IClark(.), the different versions of Clark's Completion, when using c2d and miniC2D with a tree decomposition-guided variable order for knowledge compilation. The results are shown in Fig. 9, where Clark(.), PClark(.), and IClark(.) are denoted by "-n", "-o", and "-b", respectively. We see that both for c2d and miniC2D Clark(.) is outperformed by PClark(.) and IClark(.). Especially, IClark(.) exhibits a significant performance increase compared to Clark(.).

We noticed that while tree decomposition guidance is beneficial for the overall performance, this is not so clear cut for the runtime of individual instances that Fig. 10 shows. We see that IClark(.) mostly leads to better performance, especially for c2d but to a lesser extent also for miniC2D. However, in both cases, there are also some instances that are solved slower using IClark(.) than using Clark(.). We assume this is because IClark(.) yields more general worst case guarantees but may lead to an increase in CNF size and treewidth by a constant factor compared to Clark(.) when the high treewidth is inherent rather than due to translation issues that are addressed by tree decomposition guidance (i.e. rules with long bodies or many rules with the same head). In this case, the

[10] Recall the observation after Lemma 26.

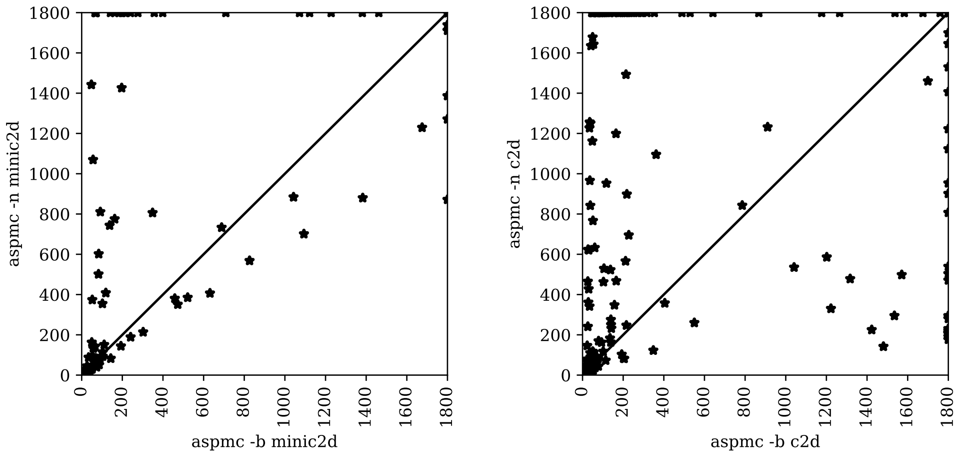


Fig. 10. Scatter plots comparing the runtimes of individual files computed by configuration "-b" (incidence guiding), compared to "-n" (no guiding) for miniC2D (on top) and c2d (on bottom).

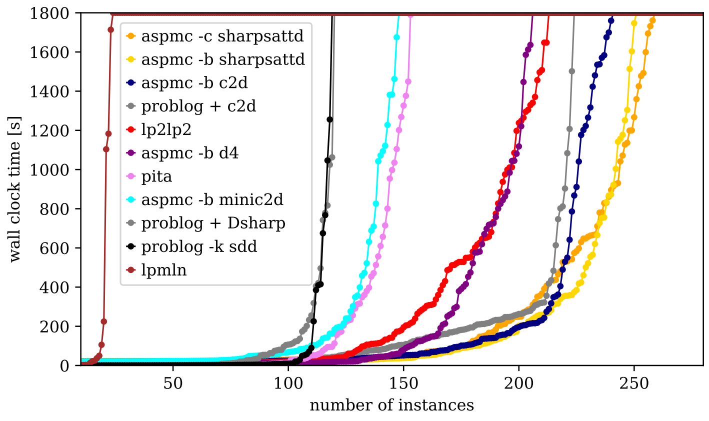


Fig. 11. Cactus plot of the results for selected solver configurations over all benchmark instances, ordered (top is better) by the number of solved instances.

performance naturally suffers a bit. This matches the intuition that we get from the scatter plots in Fig. 10. While sometimes no guidance leads to better performance, the gain is somewhat limited.

Nevertheless, we chose to add an additional configuration to aspmc in order to test whether we can counter this phenomenon. The idea is that we only use guidance if it is expected to provide a benefit. To estimate the latter, we compute an approximate prediction for the treewidths of the CNFs obtained by all three versions of Clark's Completion and use the version with the lowest value. We included the runtime of this configuration of aspmc in Fig. 9 together with the knowledge compiler SHARPSAT-TD as "aspmc -c sharpsattd". While it has higher runtime than "aspmc -b sharpsattd" on many instances, it indeed solves a few more instances. Therefore, we chose to include it for SHARPSAT-TD.

Q2.2. Effect of knowledge compiler Apart from Clark's Completion, also the knowledge compiler used to obtain a tractable circuit representation has an effect on the performance. The results of the comparison of such compilers are also given in Fig. 9. Here, we see that SHARPSAT-TD leads to the best performance as expected; c2d and d4 have worse performance but both outperform miniC2D significantly. Interestingly, c2d solves more instances than d4 although d4 performed better in the weighted track of the 2021 model counting competition. We suspect the comparatively poor performance of miniC2D results as it produces SDDs with a fixed vtree, while all other solvers produce sd-DNNFs. This means that variables always need to occur in the same order in every branch of the produced circuit, which lowers the benefit of unit propagation in the solver.

Given the findings of *Q2.1* and *Q2.2*, we conclude that configuration "aspmc -c sharpsattd", i.e. dynamically choosing which version of Clark's Completion to apply based on approximations of the treewidth of the resulting CNF in combination with SHARPSAT-TD as knowledge compiler, is the most promising configuration for aspmc.

Q3. Overall performance Having established the best configuration for aspmc, we investigated the overall performance of aspmc compared to other solvers for probabilistic inference. Here, we exclude the configurations of aspmc that use no guidance ('-n') or primal guidance ('-o'), since they performed worse during the internal evaluation. The results are summarized in Fig. 11.

We observe that aspmc's best configuration ("aspmc -c sharpsattd") has by far the best overall performance of all solvers with 259 solved instances. But also the other configurations, except the one that uses miniC2D performed very well, with 251 ("aspmc -b sharpsattd"), 241 ("aspmc -b c2d"), and 206 ("aspmc -b d4") solved instances. ProbLog solved 224 instances when using c2d as the knowledge compiler ("ProbLog + c2d"), however interestingly it only solved 120 and 119 instances when DSHARP ("ProbLog + Dsharp") and PySDD ("ProbLog-k sdd") are used, respectively.

The next best performing approach was LP2LP2,[11] which solved 213 instances - only 7 more than "aspmc -b d4". LP2LP2 and "aspmc -b d4" are followed by PITA, which solved 154 instances, thus beating "aspmc -b minic2d", which only solved 148 instances. Last but not least, $\mathrm{LP}^{\mathrm{MLN}}$ solved 24 instances.

It is unsurprising that $\mathrm{LP}^{\mathrm{MLN}}$ solved the least instances, since it explicitly enumerates all the models. For probabilistic inference on ProbLog programs this quickly becomes infeasible, since the number of answer sets it needs to consider is exactly 2^F, where F is the number of probabilistic facts of the program.

Similarly, it was expected that aspmc solved the most instances, due to our advancements of cycle breaking and Clark's Completion combined with the high efficiency of modern knowledge compilers.

The results of ProbLog in comparison to the other solvers are surprising. In Q1, we found that JN(.) outperforms MJ(.) on many instances but has worse performance than $\mathrm{T}_{\mathcal{P}}$-Unfolding on the relevant instances. Thus, one may expect that the different cycle breakings determine the performance on cyclic instances and that on acyclic instances the performance mostly depends on the knowledge compiler. Here, we however see that this is not the case. While aspmc solves more instances than LP2LP2 and ProbLog, LP2LP2 solves less instances overall, even though it uses the likely best knowledge compiler (i.e. SHARPSAT-TD) and solves more cyclic instances.

Additionally, using c2d instead of DSHARP for compilation in ProbLog seems to have a huge positive impact on the performance; while a positive effect was not completely unexpected, the extent was surprising.

The relation of the performance of ProbLog's compilation to SDDs ("ProbLog -k sdd") and PITA, however, may come as a surprise. The latter uses bottom up compilation to BDDs, whereas the former performs bottom up compilation to SDDs, which have strictly better theoretical guarantees than BDDs. However, BDDs may already be good enough; in this case a more complex representation using SDDs may come with unnecessary overhead. Additionally, existing BDD libraries may use additional performance optimizations not implemented in the SDD library that ProbLog uses. Furthermore, neither implementation uses the theoretical guarantees explicitly by computing a vtree/variable order e.g. from a tree decomposition. Instead, PITA's BDD library CUDD [44] uses dynamic variable reordering to speed up compilation. While PySDD also has a similar feature, its usage has been disabled in ProbLog for probabilistic inference.[12] The fact that ProbLog does not solve any of the blood instances, which have a simple structure that could be exploited, using option "-k sdd" but solves some (resp. all of them) when compiling to sd-DNNF with DSHARP (resp. c2d), as can be observed in Table 2. Apart from that, PITA does not use the same idea for handling cyclic dependencies as ProbLog, which may also contribute to the improved performance on cyclic instances.

This covers the overall performance comparison of the different solvers. We consider in more detail the performance on the different benchmark sets, shown in Table 2. Mostly, the results per benchmark set are reflected by the general performance already. However, we want to highlight some interesting findings separately.

Especially an explanation of the high overall performance of "ProbLog + c2d" is of interest here, given the relatively poor performance on cyclic instances. We see immediately that the high performance comes from the blood and gnb sets. Not only does "ProbLog + c2d" solve the most instances here, it is also noticeably faster than other configurations. While it also solves the most instances on the gh set, we see that there is a tie with aspmc here.

One may think that merely the use of c2d as a compiler causes the performance. However, on the cyclic benchmark sets "ProbLog + Dsharp" actually solves slightly more instances than "ProbLog + c2d". Additionally, we see that also "aspmc-b c2d" did not solve as many instances on the blood and gnb data sets as "ProbLog + c2d". Hence, there is likely an additional different reason for the performance.

We investigated ProbLog more closely and found that while its Clark Completion is standard (i.e. corresponding to Clark(.)), it makes use of a manifold of preprocessing techniques on the program level, in order to reduce the size of the program representation before Clark's Completion is applied [88,35]. We therefore compared the CNF sizes produced by ProbLog, LP2LP2, and aspmc to see if there were significant differences on the gnb and blood sets. (See Fig. 12.)

The results are given in Section 9.3. We indeed see a significant difference in the sizes of the CNFs produced by the different tools. While aspmc and LP2LP2 behave similarly, ProbLog's encodings are much smaller due to its program level preprocessing capabilities. The smaller size and the usage of c2d as a knowledge compiler indeed reasonably explain why ProbLog is so much faster on these instances. Especially the blood instances have low treewidth and thus the performance bottle neck is not the complicated structure (as it is e.g. for the smokers instances) but the size of the encoding.

Apart from that, let us closely look at using aspmc with d4. We note that even though the performance is not extremely high in general, it performs best on the benchmark sets gnb and tree. We attribute this to the fact that d4 uses a different heuristic for the selection of variables during compilation than those of the other compilers, which are all primarily tree decomposition guided. However, while this seems to help for gnb and tree, on the blood benchmark set a tree decomposition guided heuristic seems much

[11] The results for LP2LP2 must be taken with a grain of salt, since model counting over an sd-DNNF is slightly easier than evaluating multiple probabilistic queries over an sd-DNNF.

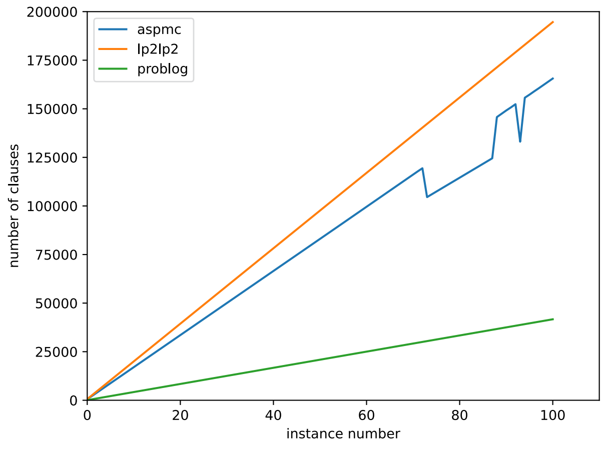
[12] At least this is what we found when inspecting the source code of ProbLog.

Table 2
Results of the overall comparison of the different solvers and configurations. The columns show the solver, the sum of solved instances on the benchmark set, the maximum treewidth (upper bound) of a solved instance, followed by the number of solved instances in different ranges of treewidth upper bounds. The last two columns show the number of instances solved by this solver only and the total time spent on the benchmark set. Boldface highlights best values.

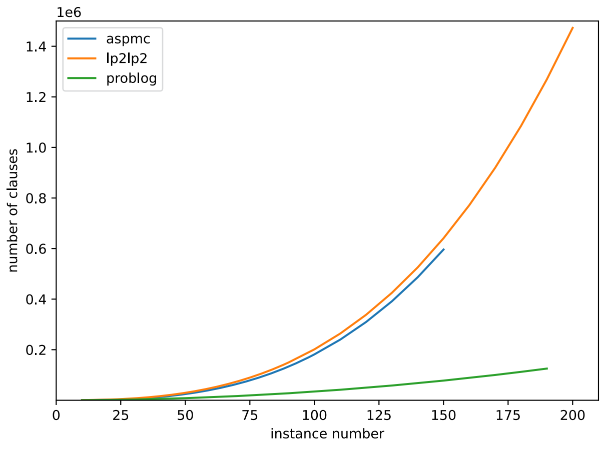
		tw range group					
solver	∑	max(tw)	0-10	10-20	>20	unique	time[h]
growing negated body							
problog + c2d	**64**	**191.0**	**9**	**10**	**45**	**5**	**2.15**
aspmc -b d4	59	141.0	**9**	**10**	40	0	4.33
aspmc -c sharpsattd	59	141.0	**9**	**10**	40	0	4.88
aspmc -b sharpsattd	58	131.0	**9**	**10**	39	0	4.92
lp2lp2	56	121.0	**9**	**10**	37	0	5.66
problog + Dsharp	53	81.0	**9**	**10**	34	0	7.09
problog -k sdd	52	71.0	**9**	**10**	33	0	6.68
aspmc -b c2d	51	61.0	**9**	**10**	32	0	8.54
pita	50	61.0	**9**	**10**	31	0	10.42
aspmc -b minic2d	24	25.0	**9**	**10**	5	0	21.18
lpmln	6	7.0	6	0	0	0	29.51
blood							
problog + c2d	**100**	**10.0**	**100**	**0**	**0**	**13**	**4.27**
lp2lp2	75	**10.0**	75	**0**	**0**	0	22.11
aspmc -c sharpsattd	73	**10.0**	73	**0**	**0**	0	23.48
aspmc -b c2d	68	**10.0**	68	**0**	**0**	0	22.34
aspmc -b sharpsattd	67	**10.0**	67	**0**	**0**	0	21.18
aspmc -b d4	29	**10.0**	29	**0**	**0**	0	39.56
pita	14	**10.0**	14	**0**	**0**	0	43.85
aspmc -b minic2d	11	**10.0**	11	**0**	**0**	0	46.87
problog + Dsharp	9	**10.0**	9	**0**	**0**	0	46.06
problog -k sdd	0	0	0	**0**	**0**	0	50.5
lpmln	0	0	0	**0**	**0**	0	50.5
near tree							
aspmc -b d4	**22**	**53.0**	**1**	**6**	**15**	**3**	**15.68**
aspmc -c sharpsattd	19	50.0	**1**	**6**	12	0	16.37
aspmc -b c2d	18	50.0	**1**	5	12	0	16.49
aspmc -b sharpsattd	16	50.0	**1**	4	11	0	17.17
aspmc -b minic2d	16	50.0	**1**	5	10	0	17.48
problog + Dsharp	13	50.0	**1**	4	8	0	18.67
problog + c2d	13	50.0	**1**	4	8	0	18.96
pita	10	36.0	**1**	5	4	0	20.62
lp2lp2	9	29.0	**1**	5	3	0	21.63
problog -k sdd	3	13.0	**1**	2	0	0	23.52
lpmln	0	0	0	0	0	0	25.0
smoker							
aspmc -b sharpsattd	**92**	**7.0**	**92**	**0**	**0**	**3**	131.76
aspmc -c sharpsattd	90	**7.0**	90	**0**	**0**	2	**131.3**
aspmc -b c2d	88	**7.0**	88	**0**	**0**	1	132.84
aspmc -b minic2d	82	**7.0**	82	**0**	**0**	0	136.44
aspmc -b d4	81	**7.0**	81	**0**	**0**	0	135.97
pita	64	6.0	64	**0**	**0**	0	144.34
lp2lp2	57	6.0	57	**0**	**0**	0	147.89
problog -k sdd	52	6.0	52	**0**	**0**	0	149.38
problog + Dsharp	29	4.0	29	**0**	**0**	0	160.45
problog + c2d	28	4.0	28	**0**	**0**	0	160.59
lpmln	12	3.0	12	**0**	**0**	0	168.75
growing head							
problog + c2d	**19**	**21.0**	8	**10**	**1**	**1**	**3.54**
aspmc -c sharpsattd	18	**21.0**	**9**	8	**1**	0	4.18
aspmc -b sharpsattd	18	19.0	**9**	9	0	0	4.47
pita	16	18.0	8	8	0	0	4.56
lp2lp2	16	17.0	**9**	7	0	0	4.72
problog + Dsharp	16	18.0	8	8	0	0	4.75
aspmc -b c2d	16	17.0	**9**	7	0	0	4.97
aspmc -b minic2d	15	16.0	**9**	6	0	0	5.15
aspmc -b d4	15	16.0	**9**	6	0	0	5.16
problog -k sdd	12	14.0	8	4	0	0	6.53
lpmln	6	8.0	6	0	0	0	9.98

Table 2 (*continued*)

solver	Σ	tw range group max(tw)	0-10	10-20	>20	unique	time[h]
		Σ					
aspmc -c sharpsattd	**259**	141.0	**182**	**24**	53	2	180.22
aspmc -b sharpsattd	251	131.0	178	23	50	3	**179.51**
aspmc -b c2d	241	61.0	175	22	44	1	185.18
problog + c2d	224	**191.0**	146	**24**	54	**19**	189.51
lp2lp2	213	121.0	151	22	40	0	202.01
aspmc -b d4	206	141.0	129	22	**55**	3	200.71
pita	154	61.0	96	23	35	0	223.8
aspmc -b minic2d	148	50.0	112	21	15	0	227.13
problog + Dsharp	120	81.0	56	22	42	0	237.03
problog -k sdd	119	71.0	70	16	33	0	236.6
lpmln	24	8.0	24	0	0	0	283.75



(a)



(b)

Fig. 12. Numbers of clauses of the CNF produced by aspmc, LP2LP2 and ProbLog on (a) the blood benchmarks and (b) the gnb benchmarks.

better. Here, d4 only solves 29 instances, whereas aspmc with SHARPSAT-TD solves more than 70. This makes sense, since the maximum treewidth upper-bound for blood was only 10, which is rather low for an acyclic instance.

Next, we take a closer look at the cyclic instances, i.e. the smokers and tree benchmark sets. Here, PITA turns out to be the best non-aspmc based approach, solving 74 instances overall. While "aspmc -c sharpsattd" solves significantly more instances, namely 119, this is still better than LP2LP2, which solved 66 instances even though compilation is performed with SHARPSAT-TD. Also ProbLog only solved 55, 31, and 32 instances, in its configurations "-k sdd", "+ Dsharp", and "+ c2d", respectively.

Summing up, we observe that on all benchmark sets except for blood a configuration of aspmc performed best or second best. Second, we saw that ProbLog performs very well in the acyclic case. However, this is not due to a better strategy for knowledge compilation or Clark's Completion but due to its advanced preprocessing capabilities. Considering the preprocessing techniques of ProbLog for aspmc is thus interesting and desirable, which however is non-trivial and goes beyond the scope of this work.

Nevertheless, this supports the claim that combining our significant improvements for cycle breaking, Clark's Completion, and the knowledge compilation step in one solver together with ProbLog's preprocessing techniques will result in even better performance on both cyclic and acyclic instances. Thus, this work brings us significantly closer to a scalable approach for probabilistic inference.

10. Conclusion

We first provide an overall summary and then we turn to issues for future work.

10.1. Summary & findings

We considered the knowledge compilation step that is usually used in order to solve Algebraic Answer Set Counting (AASC) problems, revealing interesting new insights. Its incorporation into the AASC solver aspmc led to significant performance improvements, especially on instances with positive cyclic dependencies.

First, we considered a variety of different approaches to the knowledge compilation step. For the target representation MODS exploited in enumeration-based approaches, we consider each answer set exactly once, resulting in high efficiency for "few" answer sets but insurmountable intractability otherwise. Interestingly, most of the currently available worst case guarantees for the other, more succinct tractable circuit representations are based on the same kind of structural parameters. For compilation to sd-DNNFs and SDDs, we can derive from Theorems 16, 36 and 38 upper bounds in terms of the treewidth of a Boolean circuit representing the program. This applies regardless of whether compilation proceeds in a bottom up (using the apply operator for SDDs) or top down manner (using a knowledge compiler for CNFs). Similarly to SDDs, BDDs have performance guarantees in terms of the pathwidth of a Boolean circuit representing the program, for both bottom up and top down compilation.

There exist, however, vast differences in the quality of the guarantees. These different guarantees favor top down compilation to sd-DNNF unless the given program is much larger than the number of inputs to a Boolean circuit representing the program. Apart from that, the theoretical guarantees for SDDs seem to be better than those for BDDs, especially since every BDD can be represented as an SDD. Arguably, SDDs also come at an overhead incurred by their potentially smaller sizes. Nevertheless, one would expect SDDs to perform better than BDDs for bottom up compilation.

Second, given our preference for top down compilation to sd-DNNFs, we investigated how programs can be favorably translated to CNFs for compilation afterwards. For the customary steps in such a translation, i.e. cycle breaking and Clark Completion, we provided advancements.

For Clark's Completion, Hecher's [55] idea of guiding it with a tree decomposition in $\mathrm{PClark}(.)$ limits the increase of the treewidth to a factor of three compared to the *primal* treewidth of the program. We instead provided the same result for the *incidence* treewidth of the program using $\mathrm{IClark}(.)$, which may result in even better guarantees.

For cycle breaking, we analyzed the approaches MJ(.) [58] and JN(.) [56], providing novel results showing upper bounds on the treewidth increase incurred by both translations. Here, we found guarantees for MJ(.) that are rather weak: Theorem 19 only gives a guarantee that grows linearly in the number of simple positive cycles in the program and the number of atoms that are in positive cyclic dependencies. While this upper bound may not be tight, the lower bounds of Theorem 20 show that there is a family $(\Pi_n)_{n\in\mathbb{N}}$ of programs with constant treewidth and exactly one simple positive cycle such that the treewidth of $(\mathrm{MJ}(\Pi_n))_{n\in\mathbb{N}}$ grows at least linearly in the size of the program. The guarantees of Theorem 21 for JN(.) on the other hand are rather strong, leading only to an increase by at most a logarithmic factor in the size of the largest SCC of the positive dependency graph $\mathrm{DEP}(\Pi)$ of the program Π. Nevertheless, previous experimental results revealed that large constant factors in the size bound for JN(.) limit the effectiveness of JN(.) for probabilistic reasoning on instances that are not large enough for the asymptotic guarantees to kick in [36]. We therefore introduced a new cycle breaking called $\mathrm{T}_{\mathcal{P}}$-Unfolding. While the latter may lead to a linear increase of the treewidth in the size of the largest SCC of $\mathrm{DEP}(\Pi)$ this may be avoided in many cases, unless $\mathrm{DEP}(\Pi)$ is highly dense. To achieve this, we introduced the component-boosted backdoor size $\mathrm{cbs}(.)$, which intuitively measures the distance to directed or undirected acyclicity, similar to parameters such as elimination distance [98]. We used $\mathrm{cbs}(.)$ in Theorem 22 to bound both the size increase of the program and treewidth, notably with low constant factors.

The implementation of the theoretical advancements in the AASC solver aspmc proved to be successful. The in-depth experimental evaluation confirmed our hypotheses on the effectiveness of the different approaches to cycle breakings, showing that while $\mathrm{T}_{\mathcal{P}}$-Unfolding can lead to higher treewidth upper bounds than JN(.), the latter together with the lower CNF size and presumably lower "semantic complexity" are still sufficient to allow for improved efficiency in the compilation step. Overall we see that in the cyclic setting, MJ(.), JN(.), and $\mathrm{T}_{\mathcal{P}}$-Unfolding solved 44, 66, and 119 instances, respectively, when cycle breaking is followed by CNF conversion and compilation to sd-DNNF.

Also investigating Clark's Completion paid off: both c2d and miniC2D solve the largest number of instances, if we use the incidence guided version of Clark's Completion.

Last but not least, we see most of our assumptions on the relationship between the various knowledge compilation techniques confirmed. That is, the compilation of CNFs to sd-DNNFs provides very good performance, as long as the MJ(.) cycle breaking is avoided. Unsurprisingly, the enumeration-based approach of $\mathrm{LP}^{\mathrm{MLN}}$ struggles on the probabilistic inference benchmark sets that we considered: they always have exponentially many answer sets in the number of probabilistic facts. Previous work however showed that for programs that have a complicated structure but not many answer sets, enumeration can be beneficial [61]. While different explanations seem possible, the exact cause of the low performance of bottom up compilation to SDD compared to bottom up compilation to BDDs remains somewhat mysterious.

Summing up, our work not only improves the performance of probabilistic inference, and as such AASC evaluation beyond that, but also sheds light on why some approaches work better than others based on a solid theoretical analysis.

10.2. Outlook

While our theoretical considerations of the different knowledge compilation approaches give some intuition on when and why they should be successful, the non-aligning empirical results make a further investigation desirable. In particular, the fact that PITA's bottom up compilation to BDDs outperforms ProbLog's bottom up compilation to SDDs. Possible differences in the way positive cyclic dependencies are handled may explain some of the differences but not the poor performance of this variant of ProbLog on the blood benchmark set. That ProbLog's implementation uses a randomly generated vtree and no dynamic variable ordering, whereas the BDD implementation of PITA employs dynamic variable reordering may be a possible factor of the performance difference. It seems plausible that dynamic variable reordering for SDDs is more expensive than for BDDs and thus disabled. However, it would be interesting to explore possibilities to generate a vtree using the structure of a Boolean circuit representing the program. This seems possible based on a tree decomposition of the Boolean circuit, given results such as Theorem 38. However, for bottom up compilation we are unaware of work in this direction.

Apart from this, preprocessing on the program level before cycle breaking is an interesting yet unexplored topic in the context of general AASC. The only simplifications aspmc currently uses are those that occur while clingo grounds the instance. In contrast, ProbLog and PITA employ a wide range of simplifications: identifying relevant ground parts of the program [35], program level size optimizations [88], and using zero/one probabilities [13]. As we have seen, they can have a significant impact on evaluation.

While aspmc still solved the most instances overall, we can expect another significant performance boost by combining such preprocessing techniques and the advancements of aspmc. However, these simplifications are geared towards ProbLog programs and not all of them trivially generalize to AASC setting, where a more general class of programs (e.g., non-stratified ones) is allowed. It would be interesting to adopt them to the AASC setting, and possible find new techniques, especially since preprocessing in the context of model counting can have significant benefits [115].

An open question is also whether the double exponential upper bound of answer set counting in terms of treewidth k, proven in [22], is tight when using treewidth as the only parameter. For checking whether some answer set exists, we know a $2^{\mathcal{O}(k\log(k))}\cdot|\Pi|$ upper bound that cannot be substantially improved under the Strong Exponential Time Hypothesis [55], but for answer set counting the gap remains.

Besides the aforementioned points that focus on AASC specifically, there are two open problems relevant in a bigger picture. First, many preprocessing techniques on CNFs that contribute to the success of propositional model counters do not trivially generalize to the weighted/algebraic setting. It would be interesting to see whether and how we can achieve an efficient such generalization. Second, it would be interesting to see whether compilation of acyclic programs can deliver faster knowledge compilation given low incidence treewidth. Our technique using the incidence-guided version of Clark's Completion guarantee an upper bound of $\mathcal{O}(2^{3k}\cdot|\Pi|)$ on the time needed for model counting; however, recent work showed that model counting on a CNF C parameterized by incidence treewidth is feasible in $\mathcal{O}(2^k\cdot poly(|C|))$ worst case time [116].

Beyond further efficiency improvements for AASC, another big open question is how aggregates [117] should be treated. While it is in principle possible to translate programs with aggregates into ones without them that are accepted by aspmc [118,119,56], aggregates are not available natively in aspmc. However, this approach seems to severely impede the efficiency of inference [81] likely due to increases in the treewidth and the size of the encoding. To circumvent this problem, we see multiple approaches. On the one hand, it may be possible to adapt the current translations to become treewidth-aware. Beyond that a specialized knowledge compiler that supports aggregates as first hand language constructs seems highly promising although likely a deeply intricate endeavor [120].

CRediT authorship contribution statement

Thomas Eiter: Writing – review & editing, Conceptualization, Methodology, Supervision. **Markus Hecher:** Conceptualization, Formal analysis, Methodology, Visualization, Writing – review & editing. **Rafael Kiesel:** Conceptualization, Formal analysis, Methodology, Software, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All data and source is available open source.

Acknowledgements

This research was funded in whole or in part by the Austrian Science Fund (FWF) [10.55776/P32830, 10.55776/W1255, 10.55776/J4656]. For open access purposes, the author has applied a CC BY public copyright license to any author accepted manuscript version arising from this submission. Further funding was given by the Society for Research Funding in Lower Austria (GFF, Gesellschaft für Forschungsförderung NÖ) grant ExzF-0004, as well as by the Vienna Science and Technology Fund (WWTF) grant ICT19-065. Part of the work was carried out while Hecher was visiting the Simons Institute for the Theory of Computing.

Appendix A. Full proofs and auxiliary lemmas

Theorem 16. *Given a normal answer set program* Π *and a tree decomposition* $\mathcal{T}=(T,\chi)$ *of* $\mathrm{INC}(\Pi)$ *with width* k *and root* t_r*, the CNF* $\mathrm{IClark}(\Pi,\mathcal{T},t_r)$ *can be constructed in time linear in* $|\Pi|+|T|$ *and satisfies that*

(i) *every model of* $\mathrm{Clark}_{\mathrm{Prop}}(\Pi)$ *can be uniquely extended to a model of* $\mathrm{IClark}(\Pi,\mathcal{T},t_r)$*,*
(ii) *the size of* $\mathrm{IClark}(\Pi,\mathcal{T})$ *is in* $\mathcal{O}(k|\Pi|)$*, and*
(iii) *the treewidth of* $\mathrm{PRIM}(\mathrm{IClark}(\Pi,\mathcal{T},t_r))$ *is at most* $3(k+1)$*.*

Proof. We can assume w.l.o.g. that every node $t\in T$ has at most 2 children and that T has at most $|\Pi|$ nodes. Otherwise, we transform the tree decomposition into an equivalent one, where this is the case. This is possible in linear time in $|T|$ [121].

In order to obtain a CNF we express every equivalence $aux\leftrightarrow\bigvee_{x\in X}$ in Definition 15 by the clauses $\neg aux\vee\bigvee_{x\in X}x$ and $aux\vee\neg x$ for $x\in X$. Similarly, for $aux\leftrightarrow\bigwedge_{x\in X}$ we use the clauses $aux\vee\bigvee_{x\in X}\neg x$ and $\neg aux\vee x$ for $x\in X$.

For 1. first recall that the Tseitin transformation has the desired property. Then, observe that the given translation partially applies the Tseitin transformation on the formula

$$a \leftrightarrow \bigvee_{r\in\Pi, head(r)=a} \bigwedge_{l\in body(r)} l$$

for every $a \in \mathcal{A}(\Pi)$. Since Clark's Completion is the conjunction of the above formula for every $a \in \mathcal{A}(\Pi)$, this proves the desired claim.

2. follows immediately from Definition 15. We introduce one auxiliary atom $forced_r$ for every rule in Π and one auxiliary atom $upto_t^x$ for every combination x, t such that $x \in \chi(t)$. Last but not least, observe that we have at most six equivalences for each pair x, t such that $x \in \chi(t)$. Furthermore, each of these equivalences involves at most $k+2$ variables, since every node has at most two children. Thus, the clauses corresponding to the equivalences have at most $k+2$ atoms and there are at most $6 * (k+2)^2$ many of them for each node $t \in T$. Since $|T| \leq |\Pi|$, we are done.

In order to prove 3. we construct a tree decomposition $\mathcal{T}' = (T', \chi')$ of $\mathrm{PRIM}(C)$ from the given tree decomposition of $\mathrm{INC}(\Pi)$. Namely, we assume that $\chi(t) = \{x_1^t, \dots, x_{|\chi(t)|}^t\}$ and use $T' = (V', E')$, where

$$\begin{aligned} V' &= \{(t,i) \mid t \in T, i = 1, \dots, |\chi(t)|\} \\ E' &= \{((t,i+1),(t,i)) \mid t \in V, i = 1, \dots, |\chi(t)| - 1\} \\ &\quad \cup \{((t,1),(t',|\chi(t')|)) \mid (t,t') \in E\} \end{aligned}$$

and

$$\begin{aligned} \chi'(t,i) = &\chi(t) \cap \mathcal{A}(\Pi) \cup \{forced_r \mid r \in \chi(t)\} \\ &\cup \{upto_{t'}^{x_j^t} \mid t' \in children(t), x_j^t \in \chi(t'), j \geq i\} \cup \{upto_t^{x_j^t} \mid j \leq i\}. \end{aligned}$$

Then, it holds that

$$\begin{aligned} |\chi'(t,i)| &= |\chi(t) \cap \mathcal{A}(\Pi)| + |\{forced_r \mid r \in \chi(t)\}| \\ &\quad + |\{upto_{t'}^{x_j^t} \mid t' \in children(t), x_j^t \in \chi(t'), j \geq i\}| + |\{upto_t^{x_j^t} \mid j \leq i\}| \\ &\leq |\chi(t)| + 2(k+1-i+1) + i \\ &\leq 3(k+1)+1. \end{aligned}$$

Therefore, the width of $\mathcal{T}'$ is less or equal to $3(k+1)$. Furthermore, $\mathcal{T}'$ is a tree decomposition of $\mathrm{PRIM}(C)$, since for each equivalence in Definition 15 there is a bag that contains all the variables that occur in it. $\square$

Lemma 18 *(Faithfulness Implies Query Invariance). Let $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$ be a measure and let $C(.)$ be a faithful cycle breaking for Π. Then, for $\mu' = \langle C(\Pi), \alpha, \mathcal{R} \rangle$ it holds that $\mu(a) = \mu'(a)$ for every $a \in \mathcal{A}(\Pi)$.*

Proof. In the proof of Theorem 25 we see that for each interpretation $\mathcal{I} \subseteq \mathcal{A}(\Pi)$ there can exist at most one answer set $\mathcal{I}_{ext}$ of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) such that $\mathcal{I} = \mathcal{A}(\Pi) \cap \mathcal{I}_{ext}$. Therefore, if $\mathrm{T}_{\mathcal{P}}$-unfolding is faithful for Π and s we see that either $\mathcal{I}$ is an answer set of Π and there exists *exactly* one answer set $\mathcal{I}_{ext}$ of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) such that $\mathcal{I} = \mathcal{A}(\Pi) \cap \mathcal{I}_{ext}$ or $\mathcal{I}$ is not an answer set of Π and there does not exist an answer set $\mathcal{I}_{ext}$ of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) such that $\mathcal{I} = \mathcal{A}(\Pi) \cap \mathcal{I}_{ext}$. Further, this implies that if $\mathcal{I}_{ext}$ is an answer set of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s), then $\mathcal{I} = \mathcal{A}(\Pi) \cap \mathcal{I}_{ext}$ is an answer set of Π.

Now let $a \in \mathcal{A}(\Pi)$. We know that $\mu'(a)$ is the sum of $[\![\alpha]\!]_{\mathcal{A}}(\mathcal{I}_{ext})$ over all answer sets $\mathcal{I}_{ext}$ of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s). Since α contains only variables in $\mathcal{A}(\Pi)$ it holds that $[\![\alpha]\!]_{\mathcal{A}}(\mathcal{I}_{ext}) = [\![\alpha]\!]_{\mathcal{A}}(\mathcal{I}_{ext} \cap \mathcal{A}(\Pi))$. However, $\mathcal{I} = \mathcal{I}_{ext} \cap \mathcal{A}(\Pi)$ is guaranteed to be an answer set of Π. Together with our previous observations it follows $\mu'(a) = \mu(a)$. $\square$

Theorem 20. *There is a family of programs $(\Pi_n)_{n\in\mathbb{N}}$ such that*

1. $\mathrm{DEP}(\Pi_n)$ *has exactly one simple cycle,*
2. *the treewidth of Π_n is bounded by a constant (independent of n),*
3. *the number of atoms and rules of Π_n is linear in n, and*
4. *the treewidth of $\mathrm{MJ}(\Pi_n)$ grows linearly with n.*

Proof. For this, we define

$$\begin{aligned} \Pi_n = &\{\{v(i)\} \leftarrow \mid i = 1, \dots, n\} \\ &\cup \{\{e(i,j)\} \leftarrow \mid i, j = 1, \dots, n, i+1 \equiv j \mod n\} \end{aligned}$$

$$\cup \{in(i) \leftarrow v(i) \mid i = 1, \dots, n\}$$
$$\cup \{in(i) \leftarrow e(i,j), in(j) \mid i,j = 1, \dots, n, i+1 \equiv j \mod n\}.$$

Intuitively, Π_n takes the directed graph over n vertices with arcs

$$(1,2), (2,3), \dots, (n-1,n), (n,1),$$

thus inducing exactly one cycle $1, 2, \dots, n, 1$. Then, it guesses a random subset of it. All vertices are kept such that $v(i)$ holds or the edge of the predecessor is present $e(i,j)$ and the vertex j was kept.

For all n it holds that $\mathrm{DEP}(\Pi_n)$ has exactly one simple cycle, namely

$$in(1), in(2), \dots, in(n), in(1).$$

In order to prove that the treewidth of $\mathrm{PRIM}(\Pi_n)$ is bounded by a constant, we define a tree decomposition (T_n, χ_n) as follows:

$$T_n = (V_n, E_n),$$
$$V_n = \{t_i \mid i = 1, \dots, n\},$$
$$E_n = \{(t_i, t_{i+1}) \mid i = 1, \dots, n-1\},$$
$$\chi_n(t_i) = \begin{cases} \{v(i), nv(i), in(i), in(i+1), e(i,i+1), ne(i,i+1), in(1)\} & \text{if } i = 1, \dots, n-1, \\ \{v(n), nv(n), in(n), in(1), e(n,1), ne(n,1)\} & \text{if } i = n. \end{cases}$$

First note that (T_n, χ_n) is a tree decomposition of $\mathrm{PRIM}(\Pi)$, since every rule is contained in a bag completely and every subgraph of T_n induced by taking only those vertices t_i whose bag contains an atom a is connected. Furthermore, the width of (T_n, E_n) is 6 and therefore bounded by a constant independent of n.

It remains to show that the treewidth of $\mathrm{PRIM}(\mathrm{MJ}(\Pi_n))$ grows linearly with n. For this, we first construct the program $\mathrm{MJ}(\Pi_n)$:

$$\{\{\{v(i)\} \leftarrow \mid i = 1, \dots, n\}$$
$$\cup \{\{e(i,j)\} \mid i,j = 1, \dots, n, i+1 \equiv j \mod n\}$$
$$\cup \{in(i)_F \leftarrow v(i) \mid i = 1, \dots, n, F \in \mathrm{ch}_i\}$$
$$\cup \{in(i)_F \leftarrow e(i,j), in(j)_{F \cup \{in(i)\}} \mid i,j = 1, \dots, n, i+1 \equiv j \bmod n, F \in \mathrm{ch}_i\},$$

where ch_i denotes the set of chains of atoms that can be used to derive $\mathrm{in}(i)$, given by

$$\{\emptyset, \{in(i-1)\}, \{in(i-1), in(i-2)\}, \dots,$$
$$\{in(i-1), \dots, in(1), in(n), \dots, in(i+1)\}\}.$$

Then, the primal graph of $\mathrm{MJ}(\Pi_n)$ contains at least the following edges:

1. $\{in(i)_F, in(j)_{F \cup \{in(i)\}}\}$ for $i,j = 1, \dots, n$ and $F \in \mathrm{ch}_i$ such that $i+1 \equiv j \mod n$ and $in(j) \notin F$, and
2. $\{in(i)_F, e(i,j)\}$ for $i,j = 1, \dots, n$ and $F \in \mathrm{ch}_i$ such that $i+1 \equiv j \mod n$ and $in(j) \notin F$.

In order to show lower bounds on the width of every tree decomposition of the primal graph, we need the following two facts:

Min degree lower bound [94] The treewidth of a graph G is at least its smallest node degree, i.e.

$$\min_{v \in V(G)} |\{v' \mid \{v, v'\} \in E(G)\}|.$$

Minor monotonicity [73] Given a graph G, the treewidth of any graph minor G' of G, i.e. a graph obtained from G by deleting vertices or edges or by contracting edges, is at most as high as the treewidth of G.

This means that it is sufficient to show that there is a graph minor of the primal graph of Π_n such that the minimum degree of a vertex in it is in $\Omega(n)$.

We proceed as follows. First, we contract all the edges of the first form. Since, we have for each $i = 1, \dots, n$ an edge sequence

$$\{in(i)_\emptyset, in(i+1)_{\{in(i)\}}\},$$
$$\{in(i+1)_{\{in(i)\}}, in(i+2)_{\{in(i), in(i+1)\}}\}, \dots, \{in(i-2)_F, in(i-1)_{F \cup \{in(i-2)\}}\}$$

that we contract, the vertex $res(i)$ resulting from contracting it has neighbors

$$\{e(i', j') \mid i', j' = 1, \dots, n, i' \neq i, i'+1 \equiv j' \mod n\}$$

due to the second kind of edges.

Similarly, $e(i, i+1)$ for $i = 1, \ldots, n-1$ has neighbors $\{res(i') \mid i' \neq i+1\}$. Thus, if we remove every other vertex that is not of the form $res(i)$ or $e(i,j)$, then every vertex in the remaining graph minor has at least $n-1$ neighbors. From the min-degree lower bound it follows that the primal graph of $\mathrm{MJ}(\Pi_n)$ has treewidth at least $n-1$. □

Theorem 21. *Let Π be a normal answer set program of treewidth k such that the largest strongly connected component of* $\mathrm{DEP}(\Pi)$ *has size s. Then, the treewidth of* $\mathrm{JN}(\Pi)$ *is in $\mathcal{O}(k^2 + k\log_2(s))$.*

Proof. Let (T, χ) be an optimal tree decomposition of $\mathrm{PRIM}(\Pi)$. We can modify it to be a tree decomposition of $\mathrm{PRIM}(\mathrm{JN}(\Pi))$, resulting in (T', χ'), where

$$\begin{aligned} T' &= T, \\ \chi'(t) &= \chi(t) \cup \{\mathbf{next}(a), \mathbf{just}(a) \mid a \in \chi(t)\} \cup \bigcup_{a \in \chi(t)} \mathrm{bin}(a) \\ &\quad \cup \{\mathbf{lt}(b,a), \mathbf{succ}(b,a) \mid a, b \in \chi(t), \mathbf{lt}(b,a) \in \mathcal{A}(\mathrm{JN}(\Pi))\}. \end{aligned}$$

Here, $\mathrm{bin}(a)$ refers to the set of variables that are used to encode the binary counter for $a \in \mathcal{A}(\Pi)$. For each $a \in \mathcal{A}(\Pi)$ there are at most $\lceil \log_2(s) \rceil$ such variables.

Observe that the width upper bound holds, since

$$\begin{aligned} |\chi'(t)| &= |\chi(t)| + |\{\mathbf{next}(a), \mathbf{just}(a) \mid a \in \chi(t)\}| + \sum_{a \in \chi(t)} |\mathrm{bin}(a)| \\ &\quad + |\{\mathbf{lt}(b,a), \mathbf{succ}(b,a) \mid a, b \in \chi(t), \mathbf{lt}(b,a) \in \mathcal{A}(\mathrm{JN}(\Pi))\}| \\ &\leq k + k + k\lceil \log_2(s) \rceil + k^2 \end{aligned}$$

The last expression is in $\mathcal{O}(k^2 + k\log_2(s))$, as claimed.

It remains to show that (T', χ') is a tree decomposition of $\mathrm{PRIM}(\mathrm{JN}(\Pi))$. This follows form the fact that (T, χ) is a tree decomposition of $\mathrm{PRIM}(\Pi)$. Therefore, for every rule $r \in \Pi$ there exists a bag $t \in T$ such that $head(r) \cup body(r) \subseteq \chi(t)$. In $\mathrm{JN}(\Pi)$ we only add rules that are based on an original rule $r \in \Pi$, meaning that they only use atoms from r or auxiliary atoms that are related to atoms in r. i.e. when an additional rule uses any of $\mathbf{next}(a), \mathbf{just}(a), \mathbf{lt}(b,a)$ or $\mathbf{succ}(b,a)$ then both a and b must occur in r. Since we defined $\chi'(t)$ in such a way that it includes all auxiliary atoms that are related to the atoms in $\chi(t)$ this means all additional rules derived from a rule r are completely contained in $\chi'(t)$, when r is completely contained in $\chi(t)$. □

Lemma 24. *Let Π be an answer set program and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. Then,* $\mathrm{UF}(\mathrm{DEP}(\Pi), s) = \mathrm{DEP}(\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s))$ *(when identifying a with $a^{\mathrm{cnt}(a,s)}$).*

Proof. We proof this lemma by induction on the length of s. If $|s| = 0$, then the program $\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s)$ contains only constraints and thus $\mathrm{DEP}(\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s))$ is the empty digraph. On the other hand, $\mathrm{UF}(\mathrm{DEP}(\Pi), s)$ is also equal to the empty digraph.

Next, we prove the induction step. Assume the lemma holds for all s of length up to n. Now let $s = s' s_{n+1}$ an unfolding sequence of length $n+1$. Then, $\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s)$ is equal to the union of $\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s')$ and the rules that were added in the $n+1$-th iteration (modulo renaming of $s_{n+1}^{cnt(s_{n+1})}$ to s_{n+1}). Thus we can apply the induction hypothesis and obtain that

$$\mathrm{DEP}(\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s')) = \mathrm{UF}(\mathrm{DEP}(\Pi), s').$$

We now consider the additional rules that were added in the n+1-th iteration. Then

$$\begin{aligned} V(\mathrm{DEP}(\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s))) &= V(\mathrm{DEP}(\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s'))) \cup \{s_{n+1}^{cnt(s_{n+1})}\} \\ &= V(\mathrm{UF}(\mathrm{DEP}(\Pi), s')) \cup \{s_{n+1}^{cnt(s_{n+1})}\} \\ &= V(\mathrm{UF}(\mathrm{DEP}(\Pi), s)). \end{aligned}$$

Furthermore, the edges that are added to the dependency graph after the $n+1$-th iteration are of the form (b, s_{n+1}) for $(b, s_{n+1}) \in E(\mathrm{DEP}(\Pi))$, since we only add rules that derive s_{n+1} and because we only add rule copies (see line 10). On the other hand, if (1) $(b, s_{n+1}) \in E(\mathrm{DEP}(\Pi))$ and (2) b occurred in s', then we add an edge (b, s_{n+1}) as (1) implies that there is a rule that uses b to derive s_{n+1} and (2) implies that a copy of the rule is added that has b in the body (as $last(b) = b$). This exactly correspond to the edges that we define to be in $E(\mathrm{UF}(\mathrm{DEP}(\Pi), s))$ for $k = n+1$. □

Theorem 25. *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* $\mathrm{DEP}(\Pi)$ *some directed path $\pi_c = (a_1^{c_1}, \ldots, a_n^{c_n})$ in* $\mathrm{UF}(\mathrm{DEP}(\Pi), s)$ *exists, then* $\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(., s)$ *is faithful for* Π.

Proof. Let Π be an answer set program, s an unfolding sequence that satisfies the precondition of the theorem and $\mathcal{I} \subseteq \mathcal{A}(\Pi)$. We know that for all s the reduct $\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s)^{\mathcal{I}_{ext}}$ for $\mathcal{I}_{ext} \cap \mathcal{A}(\Pi) = \mathcal{I}$ is $\mathrm{T}_{\mathcal{P}}\text{-}\textsc{Unfold}(\Pi, s)^{\mathcal{I}}$ as the rules added in line 12 use the original negative body $B^-(r)$, which only uses atoms from $\mathcal{A}(\Pi)$. Hence we can consider $\mathrm{T}_{\mathcal{P}}$-UNFOLD$(\Pi, s)^{\mathcal{I}}$, which has a unique minimal model. We see that if $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) has an answer set $\mathcal{I}_{ext}$ 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 $\mathrm{T}_{\mathcal{P}}$-Unfolding 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. W.l.o.g. we can assume an SLD tree such that every path from the root to a leaf if simple. However, we know that s preserves all simple paths and since the paths in every SLD tree correspond to paths in DEP(Π), we know there exists a corresponding SLD-tree in $\mathrm{T}_{\mathcal{P}}$-UNFOLD$(\Pi^{\mathcal{I}}, s)$. □

Theorem 22. *For any factorized measure $\mu = \langle \Pi, \alpha, \mathcal{R}\rangle$, we can construct in polynomial time in the size of Π given access to an* NP*-oracle a factorized measure $\mu' = \langle \Pi', \alpha, \mathcal{R}\rangle$ with an acyclic program Π' such that*

(i) *for all $a \in \mathcal{A}(\Pi)$ it holds that $\mu(a) = \mu'(a)$,*
(ii) *the treewidth of Π' is at most $k \cdot \mathrm{cbs}(\mathrm{DEP}(\Pi))$, where k is the treewidth of Π, and*
(iii) *the size of Π' is at most $\mathrm{cbs}(\mathrm{DEP}(\Pi)) \cdot |\Pi|$.*

For the proof we use the following auxiliary lemma.

Lemma 26. *Let Π be an answer set program with treewidth k and $s \in \mathcal{A}(\Pi)^*$ be an unfolding sequence. If every variable $a \in \mathcal{A}(\Pi)$ occurs at most m times in s, then the treewidth of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) is less or equal to $k \cdot m$.*

Proof. We know that for each variable $a \in \mathcal{A}(\Pi)$ we introduce at most $m-1$ copies $a_1, \ldots, a_{m-1}$ during unfolding. Now, let (T, χ) be an optimal tree decomposition for Π. We take (T, χ'), where

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

Recall line 11 of Algorithm 1, which is the only line where rules are added to the output program Π'. We see that for each such rule head $\leftarrow B^+_{new}, B^-(r)$ added at the iteration where we consider s_i, it holds that *head* is either a copy of s_i or s_i itself. The same holds for the atoms in B^+_{new}. For $B^-(r)$ we even only have the original atoms.

The first condition for (T, χ') to be a tree decomposition is easily seen to be true. We know that (T, χ) is a tree decomposition for PRIM(Π), therefore, every atom $a \in \mathcal{A}(\Pi)$ is in $\chi(t_a)$. Accordingly, since $\mathcal{A}(\mathrm{T}_{\mathcal{P}}$-UNFOLD$(\Pi, s))$ only contains the original atoms a or copies a^i we know that $a, a^i \in \chi(t_a)$.

Next we must check that for every edge $\{x, y\}$ in the primal graph of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) there is a node t such that $x, y \in \chi'(t)$. Again, we know that for every edge $\{x, y\} \in E(\Pi)$ there is a node $t_{x,y}$ such that $x, y \in \chi(t_{x,y})$. Furthermore, we know that there is an edge between x and y if they both occur in the same rule. Since the rules in $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s) are all copies of rules with copies of atoms, we know that if $\{x, y\} \in E(\mathrm{PRIM}(\mathrm{T}_{\mathcal{P}}$-UNFOLD$(\Pi, s)))$, where the original atom of x and y are a and b respectively, then $\{a, b\} \in E(\mathrm{PRIM}(\Pi))$. It follows that $x, y \in \chi'(t_{a,b})$.

Last but not least, since the subgraphs induced by removing all bags that do not contain a variable a are trees in the original tree decomposition, this also holds for the new tree decomposition. Thus, (T, χ') is a tree decomposition of $\mathrm{T}_{\mathcal{P}}$-UNFOLD(Π, s).

Further, it is easy to see that $|\chi'(t)| \leq |\chi(t)| \cdot m \leq k \cdot m$. □

Theorem 29. *The problem of checking whether $\mathrm{cbs}(G) \leq k$ given a digraph G and $k \in \mathbb{N}$ in the input is* NP*-complete.*

Proof. NP-membership is easy to see by a guess and check algorithm.

For NP-hardness, we use a reduction from SAT, i.e. checking whether a CNF C has a satisfying assignment. We assume w.l.o.g. that C does not have any empty clauses. We construct a digraph G with two vertices v_x and $v_{\neg x}$ for each variable $x \in Vars(C)$ and its negation $\neg x$ and one vertex v_C for each clause C. Then we add arcs to include a cycle between v_x and $v_{\neg x}$ for every variable x, in addition to one cycle $v_{l_1} \ldots v_{l_n} v_C$ for each clause $C = l_1 \vee \ldots \vee l_n \in C$.

Claim I: Let $N = |Vars(C)|$. There exists a subset $S \subseteq V(G)$ such that $G \setminus S$ is acyclic of size N iff the original CNF was satisfiable.

This can be seen as follows: Every subset $S \subseteq V(G)$ such that $G \setminus S$ is acyclic must have a size of at least N, since for each variable we need to at least include x or $\not x$ in S. Since we additionally have a cycle $v_{l_1} \ldots v_{l_n} v_C$ for each clause $C = l_1 \vee \ldots v_n \in C$ every satisfying assignment $\mathcal{I}$ to C corresponds to a set

$$S_{\mathcal{I}} = \{v_l \mid \mathcal{I} \models l, l = x, \neg x, x \in Vars(C)\}$$

that satisfies $G \setminus S_{\mathcal{I}}$ is acyclic. And additionally for every S of size N (implying that S only contains vertices of the form v_x, $v_{\neg x}$) such that $G \setminus S$ is acyclic, it holds that

$$\mathcal{I}_S = \{x \in Vars(C) \mid v_x \in S\}$$

is a satisfying assignment to C. This proves the claim.

However, $\mathrm{cbs}(G)$ does not ask for the minimum size of a backdoor S such that $G \setminus S$ is acyclic. We can however force every minimum solution for $\mathrm{cbs}(G)$ to be of that form by performing an additional modification on G. Namely, we add vertices

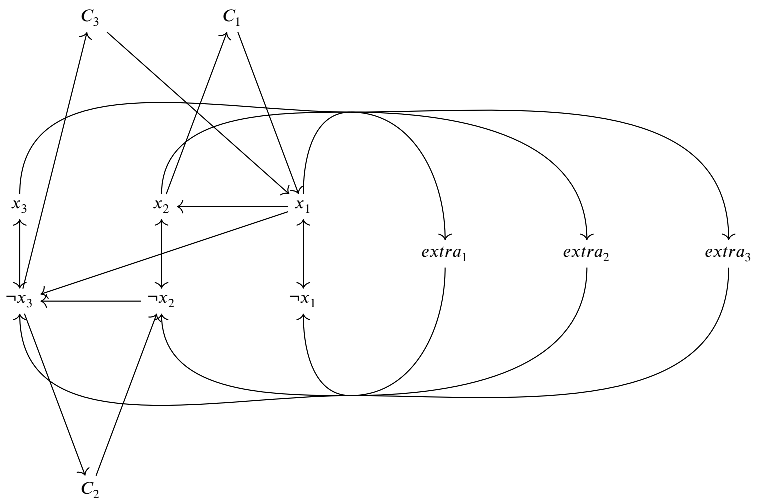


Fig. A.13. The digraph G_C constructed in the proof of Theorem 29 for the CNF $C = \{C_1, C_2, C_3\}$ with $C_1 = x_1 \vee x_2$, $C_2 = \neg x_2 \vee \neg x_3$, and $C_3 = x_1 \vee \neg x_3$. Note that the edges between x_i (resp. $\neg x_i$) and $extra_j$ are by intent drawn in an overlapping manner to improve readability.

$extra_1, \dots, extra_N$ and arcs $(v_x, extra_i)$, $(extra_i, v_{\neg x})$ for every $i \in \{1, \dots, N\}$ and $x \in Vars(C)$. An example is shown in Fig. A.13. Observe that **Claim I** still holds for the modified graph, since every added cycle uses both v_x and $v_{\neg x}$, thus, removing either of them suffices.

Claim II: Let G_C denote constructed graph. $\text{cbs}(G_C) \leq N + 1$ iff C is satisfiable.

By proving this claim the reduction is completed. We already know that if C is satisfiable, then some set S exists such that $G_C \setminus S$ is acyclic and thus by case (i) and (iv) of the definition of cbs(.) we have $\text{cbs}(G_C) = N + 1$, as desired. Assume now that $\text{cbs}(G_C) \leq N + 1$. As C contains no empty clauses, G_C is cyclic, strongly connected, and not a polytree (due to the extra vertices and arcs). Thus, we are in case (iv) of the definition of cbs(.) meaning we compute it by choosing $S \subseteq V(G_C)$ and multiplying $\text{cbs}(G_C \setminus S)$ by $(|S| + 1)$; hence $|S| \leq N$. Consider now the following three observations:

1. for $\{v_x, v_{\neg x}\}$ and $\{v_y, v_{\neg y}\}$ to lie in different SCCs of $G_C \setminus S$ without using vertices of the from $\{v_x, v_{\neg x}, v_y, v_{\neg y}\}$ for $x, y \in Vars(C)$, S needs to be at least of size N since it needs to include all vertices $extra_i$ for $i = 1, \dots, N$;
2. for $G_C \setminus S$ to be a polytree and $|S| \leq N$, the set S needs to include at least N; and
3. for $G_C \setminus S$ to be acyclic and $|S| \leq N$, the set S needs to include v_x or $v_{\neg x}$ for each $x \in Vars(\mathcal{X})$.

This means that we either need to remove a set S such that $G_C \setminus S$ is cyclic but not strongly connected or a set S of size N such that $G_C \setminus S$ is acyclic. The latter would imply that C is satisfiable and the former is impossible, since due to (1.) we would need S of size at least N to obtain N different SCCs and using a set of size k to obtain less than $k + 1$ different SCCs is not helpful, since analogous observations hold for the obtained SCCs. □

Appendix B. Knowledge compilation for AASC

Here, we discuss in detail the different possible strategies that are currently employed to compile programs into tractable circuit representations. For this, we first recall the different circuit classes that are commonly used in the area of probabilistic logic programming and their associated theoretical guarantees.

B.1. Theoretical guarantees

A very simple circuit class is the MODS representation.

Definition 34 *(MODS).* A logical theory $\mathcal{T}$ over propositional variables $\mathcal{V}$ is a MODS theory, if it is of the form

$$\bigvee_{\mathcal{I} \subseteq \mathcal{V}, \mathcal{I} \models \mathcal{T}} \bigwedge_{v \in \mathcal{I}} v \wedge \bigwedge_{v \in \mathcal{V} \setminus \mathcal{I}} \neg v,$$

i.e. if it is a disjunction of its models, where each model is represented by the conjunction of its true literals.

Clearly, given a theory $\mathcal{T}$ in MODS representation tasks like probabilistic inference are possible in polynomial time in the size of the input $\mathcal{T}$. However, the example of MODS also has obvious downsides: converting a program Π into an equivalent propositional theory $\mathcal{T}$ in MODS representation can result in a theory whose size is exponential in the one of Π, since we need one disjunct in $\mathcal{T}$

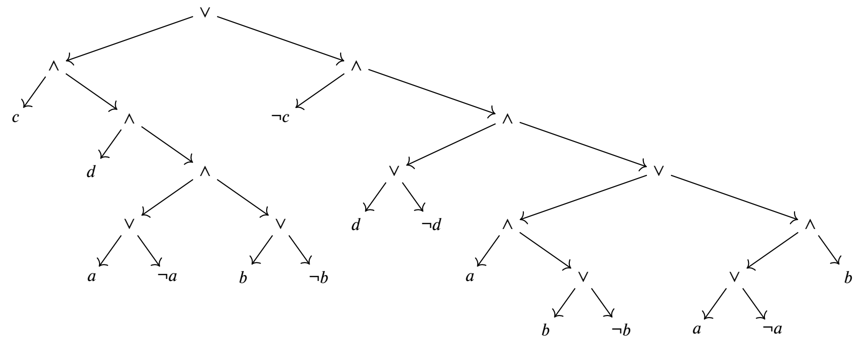


Fig. B.14. An sd-DNNFs for $C = a \vee b \vee c \wedge \neg c \vee d$.

for every answer set of Π. Thus, the size guarantee for the MODS representation of a program is rather weak, if it has many answer sets.

Corollary 35 *(Size of MODS). Let* Π *be an answer set program. The smallest MODS representation of* Π *has size* $\Theta(|\mathcal{AS}(\Pi)| \cdot |\mathcal{A}(\Pi)|)$.

Example 19. Consider the CNF

$$C = a \vee b \vee c \wedge \neg c \vee d.$$

It can be represented in MODS as follows:

$$C \equiv \begin{array}{cl} & a \wedge b \wedge c \wedge d \\ \vee & a \wedge b \wedge \neg c \wedge d \\ \vee & a \wedge b \wedge \neg c \wedge \neg d \\ \vee & a \wedge \neg b \wedge c \wedge d \\ \vee & a \wedge \neg b \wedge \neg c \wedge d \\ \vee & a \wedge \neg b \wedge \neg c \wedge \neg d \\ \vee & \neg a \wedge b \wedge c \wedge d \\ \vee & \neg a \wedge b \wedge \neg c \wedge d \\ \vee & \neg a \wedge b \wedge \neg c \wedge \neg d \\ \vee & \neg a \wedge \neg b \wedge c \wedge d \end{array}$$

There are also other more intricate tractable circuit representations, which may be exponentially smaller and more importantly allow for size guarantees with respect to structural parameters such as treewidth.

One of the most prominent tractable circuit representations is sd-DNNF. It is a special case of negation normal form (NNF) [46]. The latter is a rooted directed acyclic graph in which each leaf node is labeled with either a literal, "true", or "false", and each internal node is labeled with a conjunction $\wedge$ or a disjunction $\vee$. For any node n in an NNF graph, $\mathrm{Vars}(n)$ denotes all variables in the subgraph rooted at n. In abuse of notation, we refer by n also to the formula represented by the graph n. sd-DNNF in NNF that satisfies the following additional properties (D), (d), and (s):

Decomposability (D): $\mathrm{Vars}(n_i) \cap \mathrm{Vars}(n_j) = \emptyset$ for any two children n_i and n_j of an and-node.
Determinism (d): $n_i \wedge n_j$ is logically inconsistent for any distinct children n_i and n_j of an or-node.
Smoothness (s): $\mathrm{Vars}(n_i) = \mathrm{Vars}(n_j)$ for any two children n_i and n_j of an or-node.

Example 20 *(cont.)*. The NNF in Fig. B.14 is an sd-DNNF and equivalent to C. We observe that for each and-node, labeled $\wedge$ in the figure, the sets of variables that occur for the different children are disjoint. The NNFs of the form $a \vee \neg a$ are necessary for the circuit to be smooth. Notably, in this sd-DNNF the NNF $a \vee \neg a$ occurs twice. This is not necessary as NNFs can be DAGs. Thus, we could also point two arrows to the same NNF $a \vee \neg a$.

A big selling point of sd-DNNFs is the fixed-parameter-tractability (FPT) result that gives a guarantee on the size and time needed to construct it.

Theorem 36 *(Size of sd-DNNF [48]). Let C be a CNF and (T, χ) a tree decomposition of* $\mathrm{PRIM}(C)$*, the primal graph of C, with width k. Then, there is an equivalent sd-DNNF of size $\mathcal{O}(2^k |C|)$, which can be constructed in time $\mathcal{O}(2^k \mathrm{poly}(|C|))$ from C and (T, χ).*

This theorem even holds for a more restricted class of sd-DNNFs, namely Sentential Decision Diagrams (SDD).

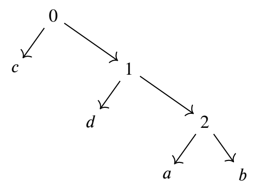


Fig. B.15. A (right-linear) vtree for the SDD in Fig. B.14.

Definition 37 *(Vtree, SDD [43]).* Let $\mathcal{V}$ be a finite set of propositional variables. A *vtree* for $\mathcal{V}$ is a rooted binary tree $T = (V, E, t_{root})$, whose leaves are labeled by variables from $\mathcal{V}$ in a one-to-one manner. Given $t \in V$, we denote by $\mathrm{Vars}(t)$ the set of labels of leafs that occur in T below t.

An *SDD* is a d-DNNF C such that every and-node of C has exactly two children, and there exists a vtree T such that for each and-node $n = n_1 \wedge n_2$ there exists a vertex t in T with children t_1 and t_2 such that $\mathrm{Vars}(n_i) \subseteq \mathrm{Vars}(t_i), i = 1, 2$.

Example 21 *(cont.).* The sd-DNNF is in fact also an SDD for the CNF C. This is witnessed by the vtree displayed in Fig. B.15.

To the best of our knowledge, it is unknown whether for each sd-DNNF C an equivalent SDD C' of size polynomial in $|C|$ exists, meaning that it may be possible that sd-DNNFs can be much smaller than SDDs for some propositional theories. However, their structuredness gives us other additional possibilities. Given two SDDs C_1, C_2 that are structured by the same vtree T, there is an operator APPLY, which computes an SDD C_3 that is equivalent to $C_1 \wedge C_2$ (or $C_1 \vee C_2$) and structured by T in polynomial time in the size of C_1 and C_2 [43]. This allows us to compile an SDD for a Boolean circuit without introducing auxiliary variables for the and-nodes and or-nodes (as it usually happens, when we apply the Tseitin Transformation to the circuit to obtain a CNF). This strategy allows for a different size guarantee.

Theorem 38 *(Size of SDD [49]). Let C be a Boolean circuit and (T, χ) be a tree decomposition of C with width k. Then, there is an SDD that is equivalent to C, whose size is in $\mathcal{O}\left(2^{2^{(k+2)2^{k+1}+1}+1} \cdot |\mathrm{Vars}(C)|\right)$.*

Importantly, the size of the circuit here is linear in the number of input variables of the Boolean circuit C, rather than the size of the CNF of Theorem 36. On the other hand, the dependency on the treewidth k is triple exponential, rather than single exponential.

It is not clear whether this is necessary. In fact, Amarilli et al. [52] were able to obtain a size upper bound of $\mathcal{O}\left(2^{4(k+1)} \cdot |\mathrm{Vars}(C)|\right)$ that is constructive, with time bound $\mathcal{O}\left(2^{5k} \cdot |\mathrm{Vars}(C)|\right)$, by dropping the requirements that vtrees are binary and that and-nodes should only have two children.

Another similar class that also features an APPLY operator is that of Binary Decision Diagrams (BDD). They can be defined as a special case of SDDs.[13]

Definition 39 *(BDD [122]).* A *BDD* is an SDD that is structured by a right-linear vtree T, meaning that for every non-leaf node of T it holds that its left child is a variable.

Example 22 *(cont.).* Since the vtree in Fig. B.15 is right-linear, the SDD in Fig. B.14 is also a BDD.

Intuitively, the restriction on the vtree to be right-linear takes away our possibility to decompose problems of the form $C = C_1 \wedge C_2$, where C_1 and C_2 are CNFs that do not share variables, into the two subproblems C_1 and C_2. For SDDs and sd-DNNFs we can solve them independently and obtain a solution for the whole CNF C. Accordingly, BDDs come with a weaker FPT result than SDDs and sd-DNNFs.

Theorem 40 *(Size of BDDs I [50]). Let C be a CNF and (T, χ) be a path decomposition of $\mathrm{PRIM}(C)$, the primal graph of C, with width k. Then, there is an equivalent BDD of size $\mathcal{O}(2^k|C|)$, which can be constructed in time $\mathcal{O}(2^k \mathrm{poly}(|C|))$ from C and (T, χ).*

Note that here the guarantee is only given in terms of pathwidth instead of treewidth, which is a weaker parameter.

As with SDDs, we have a separate result for the compilation of Boolean circuits of bounded pathwidth.

Theorem 41 *(Size of BDDs II [51,52]). Let C be a Boolean circuit and (T, χ) be a path decomposition of C with width k. Then, there is an equivalent BDD of size $\mathcal{O}(2^{(k+2)\cdot 2^{k+2}} \cdot |\mathrm{Vars}(C)|)$, which can be constructed in time $\mathcal{O}(2^{(k+2)\cdot 2^{k+2}} \cdot \mathrm{poly}(|C|))$ from C and (T, χ).*

[13] Strictly speaking these BDDs are known as Ordered BDDs (OBDDs).

Here in contrast to the same result for SDDs the dependency on the parameter k is double exponential rather than triple exponential. Arguably, pathwidth is a weaker parameter than treewidth, however only by at most a logarithmic factor in the number of vertices of the given graph [123]. In addition, since every BDD can be interpreted as an SDD, we also have the same guarantee for SDDs. Again, to the best of our knowledge it is not known whether the double exponential bound is tight [52].

This short summary of different circuit classes and known results that guarantee efficient compilation can serve us as a basis for judging different knowledge compilation based approaches to AASC. Of course, we still need to keep in mind that these theoretical guarantees may not be optimal, especially not on every instance, and that the Big-O notation hides constant factors. Furthermore, the practical efficiency also depends on how well engineered the used knowledge compilers are. Apart from this, these theoretical results are only used by some knowledge compilers practically, meaning that we may even end up with worse performance. Thus, these results should not be taken as the sole basis for choosing a knowledge compilation strategy but can at least serve as an indicator from the theoretical perspective.

B.2. Different approaches to the knowledge compilation step

In this section, we compare the different approaches to the knowledge compilation step in the overall pipeline of AASC based on the theoretical guarantees from above. Note that this is not a complete discussion in the sense that every possible approach is discussed. Instead, we only discuss approaches for which an implementation exists. Nevertheless, we still include approaches even when their current implementation does not make use of the FPT-guarantees explicitly.

Program to MODS This approach is used by LP$^{\text{MLN}}$ [10] and plingo [38]. By Corollary 35, we know that this means we need to consider each model once. Of course, finding models of a program is NP-hard but in practice this does not seem to lead to problems. When enumerating the answer sets as it is the case in LP$^{\text{MLN}}$ and plingo, the size guarantee is used explicitly. In fact, it is not even necessary to keep all the answer sets but it is sufficient to extract a value from each answer set and then discard it. Thus, this strategy is promising, when there are “few” answer sets[14] but becomes infeasible quickly, since the linear dependency on the number of answer sets cannot be avoided.

Dynamic programming This approach is used by dynasp [21], which is currently specialized on counting the number of answer sets rather than probabilistic reasoning or even general AASC. The following result is the basis for the implementation.

Theorem 42 *([21]). Let Π be a ground program and let (T, χ) be a tree decomposition of the primal dependency graph of Π with width k. Then, it is possible, assuming constant time semiring additions and multiplications, to perform AASC over Π in time $\mathcal{O}(2^{2^{k+1}}|T|)$.*

Since dynasp is based on dynamic programming over the tree decomposition, the guarantee is used explicitly.

CNF to sd-DNNF This approach has been widely considered and is for example available in ProbLog [58,36]. Here, we can use the guarantees of Theorem 36, which are both in terms of size as well as time, and have the lowest (i.e. single exponential) dependency on the treewidth of the primal graph of the input formula. In fact, in the context of CNFs the result is not only of theoretical interest, quite on the contrary, recent results showed that using them as a primary means of guiding the variable selection during model counting is also highly beneficial in practice [47].

However, this assumes that we are given a CNF of low treewidth. A priori it is unclear whether a program of low treewidth also leads to a CNF suitable for compilation of low treewidth. We will discuss this in depth in Sections 6 and 7.

Ground acyclic program to SDD This is the current standard approach in the implementation of ProbLog [35]. Here, we do not first translate the program into a CNF via Clark’s Completion but interpret the program as a Boolean circuit and obtain an SDD that represents the truth of the extensional atoms in terms of only the input variables. In the case of probabilistic inference, these are the probabilistic facts. Here, we obtain a triple exponential upper bound in terms of treewidth on the size of the smallest SDD via Theorem 38 or a double exponential upper bound in terms of the pathwidth via Theorem 41. These upper bounds are not explicitly used in the implementation of ProbLog, which employs the APPLY operator on SDDs to build the SDDs in a bottom up manner, along the Boolean circuit. In order to keep the SDDs small, dynamic reordering of the vtree can be applied heuristically.

Recall that Theorem 36 also holds for compiling CNFs to SDDs [43]. Thus, we could also make use of its guarantees, given that we can obtain a CNF of small treewidth from a Boolean circuit of small treewidth. On the one hand, this would lead to a single exponential instead of a triple exponential dependency on the treewidth for the worst case guarantee. On the other hand, it would also come with a higher remaining factor that uses the size of the CNF instead of the number of input variables.

Ground program to SDD This approach is called $\mathrm{T}_{\mathcal{P}}$-compilation [39] and is also available in the implementation of ProbLog. Since the program is not necessarily acyclic we cannot see it as a Boolean circuit. Thus, we cannot use Theorem 38 directly to obtain guarantees. However, the theorem in fact also holds when there *exists* a circuit that defines the same Boolean function as the program [49]. Thus, we have the same theoretical guarantees here as for the acyclic case. Practically, these guarantees are not used

[14] Millions of answer sets still seem to be easily feasible.

either since the APPLY operator is employed for compilation in a similar manner as before. Here, we need to make multiple passes over the program structure, which intuitively can be seen as a cyclic version of a Boolean circuit, until convergence of the SDDs.

Program to BDD This approach is used by the PITA system [13]. A theoretical guarantee can again be derived by Theorem 41. While the program is again not guaranteed to be acyclic, the theorem holds for any Boolean circuit that defines the Boolean function specified by the program [50]. Thus, the worst case guarantee here is equal or worse to the one in the previous case, since we only compile to BDD and not to SDD, which encompasses BDD as a special case. As with ProbLog, the guarantees are not used explicitly, since BDDs are built in a bottom-up manner by empolying the APPLY operator for BDDs. The BDDs can be dynamically minimized during compilation by heuristically adapting the variable ordering on the fly.

Ground program to sd-DNNF Last but not least it is important to mention the strategy of Aziz et al. [89]. They modified the knowledge compiler DSHARP to work on inputs that correspond to normal answer set programs. In their approach, they did not take care of acyclicity before but during compilation by adding so called *loop formulas* [90], which are clauses that prohibit cyclic derivations. This approach seems promising, since it does not need to blow up the size of the encoding with all potentially necessary acyclicity constraints but adds them only when needed. There is a downside to it however. Loop formulas as introduced by Lin and Zhao [90] may span a whole SCC of the dependency graph of the program.[15] Thus, the primal graph of the CNF for Clark's Completion and all loop formulas of a program Π contains a clique for each SCC of $\mathrm{DEP}(\Pi)$. Since an SCC can potentially span large parts of the program, this may increase the treewidth drastically compared to the treewidth of the original program. Apart from that, not only the treewidth may suffer: we may end up with exponentially many additional clauses in the size of the largest SCC [91]. This version of DSHARP was removed from the ProbLog implementation, presumably for the above reasons. While there are strategies to reduce the number of necessary loop formulas [92,93], their adaptation would not circumvent the exponential worst case lower bound on the number of added clauses.

Appendix C. Knowledge compilation settings

We use the different solvers for knowledge compilation as follows:

d4 For d4, we simply used standard settings adding only the input arguments "-dDNNF", to ensure that we obtain a d-DNNF as output, "-out = file_name.nnf", to ensure that the d-DNNF is saved in the desired output file, and "-smooth", to activate our minor modification that ensures the resulting d-DNNF is smooth.

SHARPSAT-TD For SHARPSAT-TD, we set the argument "-decot", i.e. the time used by SHARPSAT-TD to compute a tree decomposition based on the corresponding input argument to aspmc. The other arguments are set as follows:

- "-decow", specifies the importance of tree decomposition guidance. We use 100.
- "-dDNNF", is a new option that we added, and specifies that we want to use SHARPSAT-TD as a knowledge compiler.
- "-tmpdir", simply specifies the temporary directory that SHARPSAT-TD should use. We use "/tmp/".
- "-cs", denotes the maximum size of the cache in megabyte. We fix it to "3500".
- "-dDNNF_out", specifies the filename that the d-DNNF should be written to. We set it to the name of a new temporary file.

Without the argument "-dDNNF_out" our modified version of SHARPSAT-TD prints the d-DNNF it compiles to stdout. This allows us to perform knowledge compilation and evaluation simultaneously. While we implemented this feature in aspmc it should only lead to a speed up by a factor of up to two and is not included in the empirical evaluation.

c2d For c2d, we set the following input arguments:

- "-smooth_all", specifies that the final output d-DNNF should be smooth.
- "-reduce", specifies that c2d should perform possible reduction steps before outputting the final d-DNNF.
- "-dt_in", specifies the input dtree that should be used to guide the variable selection of c2d. While this is an optional argument, we chose to provide a dtree that we generated ourselves.
- "-cache_size", specifies the maximum size of the cache, which we set to "3500".

The other input arguments (apart from the CNF to compile) are not used.

miniC2D For miniC2D, we set the following input arguments:

- "-v", specifies the input vtree that defines the order in which variables are decided by miniC2D. Similarly to the dtree argument of c2d, this argument is optional but we chose to provide a vtree that we generated ourselves.

[15] They usually contain even more variables, however already this is devastating for knowledge compilation.

- “-s”, specifies the maximum size of the cache. We set it to “3500”.

The other input arguments (apart from CNF to compile) are not used.

Appendix D. Dtree and vtree generation

Algorithm 2 TD_TO_DTREE(C, td).

```
Input A CNF $C$ and a tree decomposition $td$ of PRIM($C$).
Output A dtree for $C$ that corresponds to $td$.
 1: td_node_to_clauses = {} # at which td node the clause should be handles
 2: last_td_node = {} # at which td node each variable occurs last
 3: idx = 0
 4: bag_idx = list(td)
 5: for bag in td.bag_iter() do
 6:     clauses[bag] = []
 7:     bag.idx = idx
 8:     for a in bag.vertices do
 9:         last_td_node = idx
10:     idx + = 1
11: # set where each clause should be handled
12: for i,c in enumerate(cnf.clauses) do
13:     idx = min([ last_td_node[abs(b)] for b in c ])
14:     td_node_to_clauses[bag_idx[idx]].append(i)
15: dtree_idx = [ None for _ in range(td.bags) ]
16: for bag in td.bag_iter() do
17:     cur_dtree = None
18:     for child in bag.children do
19:         child_dtree = dtree_idx[child.idx]
20:         if cur_dtree = = None then
21:             cur_dtree = child_dtree
22:         else
23:             if child_dtree != None then
24:                 cur_dtree = Dtree(right = cur_dtree, left = child_tree)
25:     for i in td_node_to_clauses[bag] do
26:         if cur_dtree = = None then
27:             cur_dtree = Dtree(val = i)
28:         else
29:             cur_dtree = Dtree(right = cur_dtree, left = Dtree(val = i))
30:     # remember the final dtree for this bag
31:     dtree_idx[bag.idx] = cur_dtree
32: return dtree_idx[td.get_root().idx]
```

We describe shortly how we generate dtrees and vtrees from a tree decomposition in such a way that the variables are decided in order of minimum distance from the root of the tree decomposition. Our strategy only slightly differs from that of Korhonen and Järvisalo, [47]. Dtrees describe not the order in which variables are processed but the order in which clauses are processed, which implicitly specifies multiple orders in which the variables can be processed.

The algorithm to generate a dtree from a tree decomposition td for a CNF C is given in Algorithm 2. The general idea is that each clause of C has to occur exactly once in the dtree. Then, for a dtree to correspond to a tree decomposition td, we want to first handle all the clauses whose variables occur in the root of td, remove them from the CNF and recurse for each of the children of the root on the remaining clauses, adding one subtree for each child. Practically, it is expensive to check for each clause whether it is contained in the current bag of the tree decomposition. Instead, Algorithm 2 first checks (in lines 1-12) for each variable in C, which bag of the tree decomposition is the last in post-order traversal to contain it. This tells us that the index of the last bag in post-order to contain all the variables of a clause c is the minimum index over all variables (line 15). Then, we want to handle clause c at this bag (line 16). Finally, we build the dtree from bottom up by traversing the tree decomposition again in post-order. Here, we first combine the dtrees of the children of the current bag (lines 22-27) before we add the clauses that need to be added at the current bag (lines 29-33). After processing the current bag, we store the corresponding dtree in a map from bags to dtrees. Then, the final dtree is the one that the root of the tree decomposition maps to.

The algorithm to generate a vtree from a tree decomposition works similarly; the only difference is that we directly work on variables instead of clauses.

References

[1] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, L. De Raedt, Neural probabilistic logic programming in deepproblog, Artif. Intell. 298 (2021) 103504, https://doi.org/10.1016/j.artint.2021.103504.

[2] A.N. Fadja, E. Lamma, F. Riguzzi, Deep probabilistic logic programming, in: R. Zese, C.T. Have (Eds.), Proceedings of the Workshop on Probabilistic Logic Programming 2017 Co-Located with 27th International Conference on Inductive Logic Programming (ILP 2017), Orléans, France, September 7, 2017, in: CEUR Workshop Proceedings, CEUR-WS.org, vol. 1916, 2017, pp. 3–14, http://ceur-ws.org/Vol-1916/paper1.pdf.
[3] P. Sen, B.W.S.R. de Carvalho, R. Riegel, A.G. Gray, Neuro-symbolic inductive logic programming with logical neural networks, in: Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022, AAAI Press, 2022, pp. 8212–8219, https://ojs.aaai.org/index.php/AAAI/article/view/20795.
[4] M. Klauck, On the connection of probabilistic model checking, planning, and learning for system verification, Ph.D. thesis, Saarland University, Saarbrücken, Germany, 2022, https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/33484.
[5] A. Groß, B. Kracher, J.M. Kraus, S.D. Kühlwein, A.S. Pfister, S. Wiese, K. Luckert, O. Pötz, T. Joos, D. Van Daele, et al., Representing dynamic biological networks with multi-scale probabilistic models, Commun. Biol. 2 (2019) 21.
[6] D.D. Maeyer, B. Weytjens, J. Renkens, L. De Raedt, K. Marchal, Phenetic: network-based interpretation of molecular profiling data, Nucleic Acids Res. 43 (2015) W244–W250, https://doi.org/10.1093/nar/gkv347.
[7] A. Dries, A. Kimmig, J. Davis, V. Belle, L. De Raedt, Solving probability problems in natural language, in: C. Sierra (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, ijcai.org, 2017, pp. 3981–3987.
[8] C. Baral, M. Gelfond, Logic programming and knowledge representation, J. Log. Program. 19/20 (1994) 73–148, https://doi.org/10.1016/0743-1066(94)90025-6.
[9] V. Lifschitz, What is answer set programming?, in: D. Fox, C.P. Gomes (Eds.), Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008, AAAI Press, 2008, pp. 1594–1597, http://www.aaai.org/Library/AAAI/2008/aaai08-270.php, 2008.
[10] J. Lee, Z. Yang, LPMLN, weak constraints, and P-log, in: S. Singh, S. Markovitch (Eds.), Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA, AAAI Press, 2017, pp. 1170–1177, http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14547.
[11] C. Baral, M. Gelfond, J.N. Rushton, Probabilistic reasoning with answer sets, Theory Pract. Log. Program. 9 (2009) 57–144, https://doi.org/10.1017/S1471068408003645.
[12] L. De Raedt, A. Kimmig, H. Toivonen, Problog: a probabilistic prolog and its application in link discovery, in: M.M. Veloso (Ed.), IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007, 2007, pp. 2462–2467, http://ijcai.org/Proceedings/07/Papers/396.pdf.
[13] F. Riguzzi, T. Swift, The PITA system for logical-probabilistic inference, in: S.H. Muggleton, H. Watanabe (Eds.), Latest Advances in Inductive Logic Programming, ILP 2011, Late Breaking Papers, Windsor Great Park, UK, July 31 - August 3, 2011, Imperial College Press / World Scientific, 2011, pp. 79–86.
[14] J. Vennekens, M. Denecker, M. Bruynooghe, CP-logic: a language of causal probabilistic events and its relation to logic programming, Theory Pract. Log. Program. 9 (2009) 245–308, https://doi.org/10.1017/S1471068409003767.
[15] P. Totis, A. Kimmig, L. De Raedt, Smproblog: stable model semantics in problog and its applications in argumentation, CoRR, arXiv:2110.01990 [abs], 2021, https://arxiv.org/abs/2110.01990, arXiv:2110.01990.
[16] M. Nickles, A. Mileo, Web stream reasoning using probabilistic answer set programming, in: R. Kontchakov, M. Mugnier (Eds.), Web Reasoning and Rule Systems - 8th International Conference, Proceedings, RR 2014, Athens, Greece, September 15-17, 2014, in: Lecture Notes in Computer Science, vol. 8741, Springer, 2014, pp. 197–205.
[17] V.H.N. Rocha, F.G. Cozman, A credal least undefined stable semantics for probabilistic logic programs and probabilistic argumentation, in: G. Kern-Isberner, G. Lakemeyer, T. Meyer (Eds.), Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning, KR 2022, Haifa, Israel, July 31 - August 5, 2022, 2022, pp. 309–319, https://proceedings.kr.org/2022/31/.
[18] E.M. de Morais, M. Finger, Probabilistic answer set programming, in: Brazilian Conference on Intelligent Systems, BRACIS 2013, Fortaleza, CE, Brazil, 19-24 October, 2013, IEEE Computer Society, 2013, pp. 150–156.
[19] Z. Yang, A. Ishay, J. Lee, Neurasp: embracing neural networks into answer set programming, in: C. Bessiere (Ed.), Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, ijcai.org, 2020, pp. 1755–1762.
[20] A. Skryagin, W. Stammer, D. Ochs, D.S. Dhami, K. Kersting, Neural-probabilistic answer set programming, in: G. Kern-Isberner, G. Lakemeyer, T. Meyer (Eds.), Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning, KR 2022, Haifa, Israel, July 31 - August 5, 2022, 2022, pp. 463–473, https://proceedings.kr.org/2022/48/.
[21] J.K. Fichte, M. Hecher, M. Morak, S. Woltran, Dynasp2. 5: dynamic programming on tree decompositions in action, in: International Symposium on Parameterized and Exact Computation (IPEC), in: LIPIcs, vol. 89, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 17.
[22] M. Jakl, R. Pichler, S. Woltran, Answer-set programming with bounded treewidth, in: C. Boutilier (Ed.), IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, 2009, pp. 816–822, http://ijcai.org/Proceedings/09/Papers/140.pdf.
[23] G. Brewka, J.P. Delgrande, J. Romero, T. Schaub, asprin: customizing answer set preferences without a headache, in: B. Bonet, S. Koenig (Eds.), Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA, AAAI Press, 2015, pp. 1467–1474, http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9535.
[24] F. Buccafurri, N. Leone, P. Rullo, Strong and weak constraints in disjunctive datalog, in: J. Dix, U. Furbach, A. Nerode (Eds.), Logic Programming and Nonmonotonic Reasoning, 4th International Conference, Proceedings, LPNMR'97, Dagstuhl Castle, Germany, July 28-31, 1997, in: Lecture Notes in Computer Science, vol. 1265, Springer, 1997, pp. 2–17.
[25] G. Van den Broeck, I. Thon, M. van Otterlo, L. De Raedt, DTProbLog: a decision-theoretic probabilistic prolog, in: M. Fox, D. Poole (Eds.), Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11–15, 2010, AAAI Press, 2010, pp. 1217–1222, http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1695.
[26] V. Derkinderen, L. De Raedt, Algebraic circuits for decision theoretic inference and learning, in: G.D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, J. Lang (Eds.), ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), in: Frontiers in Artificial Intelligence and Applications, vol. 325, IOS Press, 2020, pp. 2569–2576.
[27] E. Bellodi, M. Alberti, F. Riguzzi, R. Zese, MAP inference for probabilistic logic programming, Theory Pract. Log. Program. 20 (2020) 641–655, https://doi.org/10.1017/S1471068420000174.
[28] A.L.D. Latour, B. Babaki, A. Dries, A. Kimmig, G. Van den Broeck, S. Nijssen, Combining stochastic constraint optimization and probabilistic programming - from knowledge compilation to constraint solving, in: J.C. Beck (Ed.), Principles and Practice of Constraint Programming - 23rd International Conference, Proceedings, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, in: Lecture Notes in Computer Science, vol. 10416, Springer, 2017, pp. 495–511.
[29] A. Kimmig, G. Van den Broeck, L. De Raedt, An algebraic prolog for reasoning about possible worlds, in: W. Burgard, D. Roth (Eds.), Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011, AAAI Press, 2011, pp. 209–214, http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3685, 2011.
[30] A. Kimmig, G. Van den Broeck, L. De Raedt, Algebraic model counting, J. Appl. Log. 22 (2017) 46–62, https://doi.org/10.1016/j.jal.2016.11.031.
[31] T. Eiter, R. Kiesel, Weighted LARS for quantitative stream reasoning, in: G.D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, J. Lang (Eds.), ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), in: Frontiers in Artificial Intelligence and Applications, vol. 325, IOS Press, 2020, pp. 729–736.

[32] R. Kiesel, P. Totis, A. Kimmig, Efficient knowledge compilation beyond weighted model counting, Theory Pract. Log. Program. 22 (2022) 505–522, https://doi.org/10.1017/S147106842200014X.
[33] J. Eisner, Parameter estimation for probabilistic finite-state transducers, in: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, Philadelphia, PA, USA, ACL, July 6–12, 2002, 2002, pp. 1–8, https://aclanthology.org/P02-1001/.
[34] T. Eiter, R. Kiesel, On the complexity of sum-of-products problems over semirings, in: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021, AAAI Press, 2021, pp. 6304–6311, https://ojs.aaai.org/index.php/AAAI/article/view/16783.
[35] D. Fierens, G. Van den Broeck, J. Renkens, D.S. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt, Inference and learning in probabilistic logic programs using weighted Boolean formulas, Theory Pract. Log. Program. 15 (2015) 358–401, https://doi.org/10.1017/S1471068414000076.
[36] D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, L. De Raedt, Inference in probabilistic logic programs using weighted CNF's, in: Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, 2011, pp. 211–220.
[37] J. Lee, S. Talsania, Y. Wang, Computing LPMLN using ASP and MLN solvers, Theory Pract. Log. Program. 17 (2017) 942–960, https://doi.org/10.1017/S1471068417000400.
[38] S. Hahn, T. Janhunen, R. Kaminski, J. Romero, N. Rühling, T. Schaub, plingo: a system for probabilistic reasoning in clingo based on LPMLN, CoRR, arXiv:2206.11515 [abs], 2022, https://doi.org/10.48550/arXiv.2206.11515, arXiv:2206.11515.
[39] J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, L. De Raedt, Tp-compilation for inference in probabilistic logic programs, Int. J. Approx. Reason. 78 (2016) 15–32.
[40] J. Vlasselaer, J. Renkens, G. Van den Broeck, L. De Raedt, Compiling probabilistic logic programs into sentential decision diagrams, in: Proceedings Workshop on Probabilistic Logic Programming (PLP), 2014, pp. 1–10.
[41] A. Darwiche, P. Marquis, A knowledge compilation map, J. Artif. Intell. Res. 17 (2002) 229–264, https://doi.org/10.1613/jair.989.
[42] S. Mengel, Counting, knowledge compilation and applications, https://tel.archives-ouvertes.fr/tel-03611336, 2021.
[43] A. Darwiche, SDD: a new canonical representation of propositional knowledge bases, in: T. Walsh (Ed.), IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011, IJCAI/AAAI, 2011, pp. 819–826.
[44] F. Somenzi, Cudd: Cu Decision Diagram, Package Release 2.5. 0, University of Colorado at Boulder, 2012.
[45] U. Oztok, A. Darwiche, A top-down compiler for sentential decision diagrams, in: Q. Yang, M.J. Wooldridge (Eds.), Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015, AAAI Press, 2015, pp. 3141–3148, http://ijcai.org/Abstract/15/443.
[46] A. Darwiche, New advances in compiling CNF into decomposable negation normal form, in: R.L. de Mántaras, L. Saitta (Eds.), Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, Including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004, IOS Press, 2004, pp. 328–332.
[47] T. Korhonen, M. Järvisalo, Integrating tree decompositions into decision heuristics of propositional model counters (short paper), in: L.D. Michel (Ed.), 27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021, in: LIPIcs, vol. 210, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 8.
[48] U. Oztok, A. Darwiche, On compiling CNF into decision-dnnf, in: B. O'Sullivan (Ed.), Principles and Practice of Constraint Programming - 20th International Conference, Proceedings, CP 2014, Lyon, France, September 8-12, 2014, in: Lecture Notes in Computer Science, vol. 8656, Springer, 2014, pp. 42–57.
[49] S. Bova, S. Szeider, Circuit treewidth, sentential decision, and query compilation, in: E. Sallinger, J.V. den Bussche, F. Geerts (Eds.), Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017, ACM, 2017, pp. 233–246.
[50] A. Ferrara, G. Pan, M.Y. Vardi, Treewidth in verification: local vs. global, in: G. Sutcliffe, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, Proceedings, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, in: Lecture Notes in Computer Science, vol. 3835, Springer, 2005, pp. 489–503.
[51] A.K. Jha, D. Suciu, On the tractability of query compilation and bounded treewidth, in: A. Deutsch (Ed.), 15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012, ACM, 2012, pp. 249–261.
[52] A. Amarilli, F. Capelli, M. Monet, P. Senellart, Connecting knowledge compilation classes and width parameters, Theory Comput. Syst. 64 (2020) 861–914, https://doi.org/10.1007/s00224-019-09930-2.
[53] J. Lagniez, P. Marquis, An improved decision-dnnf compiler, in: C. Sierra (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, ijcai.org, 2017, pp. 667–673.
[54] F. Fages, Consistency of Clark's completion and existence of stable models, J. Methods Log. Comput. Sci. 1 (1994) 51–60.
[55] M. Hecher, Treewidth-aware reductions of normal ASP to SAT - is normal ASP harder than SAT after all?, Artif. Intell. 304 (2022) 103651, https://doi.org/10.1016/j.artint.2021.103651.
[56] T. Janhunen, I. Niemelä, Compact translations of non-disjunctive answer set programs to propositional clauses, in: M. Balduccini, T.C. Son (Eds.), Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday, in: Lecture Notes in Computer Science, vol. 6565, Springer, 2011, pp. 111–130.
[57] T. Janhunen, Representing normal programs with clauses, in: R.L. de Mántaras, L. Saitta (Eds.), Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, Including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004, IOS Press, 2004, pp. 358–362.
[58] T. Mantadelis, G. Janssens, Dedicated tabling for a probabilistic setting, in: M.V. Hermenegildo, T. Schaub (Eds.), Technical Communications of the 26th International Conference on Logic Programming, ICLP, 2010, July 16-19, 2010, Edinburgh, Scotland, UK, in: LIPIcs, vol. 7, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 124–133.
[59] F. Lin, J. Zhao, On tight logic programs and yet another translation from normal logic programs to propositional logic, in: G. Gottlob, T. Walsh (Eds.), IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003, Morgan Kaufmann, 2003, pp. 853–858, http://ijcai.org/Proceedings/03/Papers/123.pdf.
[60] J.K. Fichte, S. Szeider, Backdoors to tractable answer set programming, Artif. Intell. 220 (2015) 64–103, https://doi.org/10.1016/j.artint.2014.12.001.
[61] T. Eiter, M. Hecher, R. Kiesel, Treewidth-aware cycle breaking for algebraic answer set counting, in: M. Bienvenu, G. Lakemeyer, E. Erdem (Eds.), Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online Event, November 3-12, 2021, 2021, pp. 269–279.
[62] T. Eiter, M. Hecher, R. Kiesel, aspmc: an algebraic answer set counter, in: J. Arias, F.A. D'Asaro, A. Dyoub, G. Gupta, M. Hecher, E. LeBlanc, R. Peñaloza, E. Salazar, A. Saptawijaya, F. Weitkämper, J. Zangari (Eds.), Proceedings of the International Conference on Logic Programming 2021 Workshops Co-Located with the 37th International Conference on Logic Programming (ICLP 2021), Porto, Portugal (Virtual) September 20th-21st, 2021, in: CEUR Workshop Proceedings, CEUR-WS.org, vol. 2970, 2021, https://ceur-ws.org/Vol-2970/plppaper1.pdf.
[63] K. Kiljan, M. Pilipczuk, Feedback vertex set experiments, https://github.com/karek/FeedbackVertexSet-Experiments, 2018.
[64] C. Domshlak, J. Hoffmann, Probabilistic planning via heuristic forward search and weighted model counting, J. Artif. Intell. Res. 30 (2007), https://doi.org/10.1613/jair.2289.
[65] C.P. Gomes, A. Sabharwal, B. Selman, Chapter 20: model counting, in: Handbook of Satisfiability, in: Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009, pp. 633–654.
[66] T. Sang, P. Beame, H.A. Kautz, Performing Bayesian inference by weighted model counting, in: M.M. Veloso, S. Kambhampati (Eds.), Proceedings, the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, Pittsburgh, Pennsylvania, USA, July 9–13, 2005, AAAI Press / The MIT Press, 2005, pp. 475–482, http://www.aaai.org/Library/AAAI/2005/aaai05-075.php.

[67] P. Doubilet, G.-C. Rota, R. Stanley, On the foundations of combinatorial theory. VI. The idea of generating function, in: Berkeley Symposium on Mathematical Statistics and Probability, vol. 2, 1972, pp. 267–318.
[68] J. Lee, Y. Wang, Weight learning in a probabilistic extension of answer set programs, in: 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018), AAAI Press, 2018, pp. 22–31, https://aaai.org/ocs/index.php/KR/KR18/paper/view/18057.
[69] L.D. Smet, P.Z.D. Martires, R. Manhaeve, G. Marra, A. Kimmig, L.D. Raedt, Neural probabilistic logic programming in discrete-continuous domains, in: 39th Conference on Uncertainty in Artificial Intelligence (UAI 2023), in: Proceedings of Machine Learning Research, PMLR, vol. 216, 2023, pp. 529–538.
[70] M. Chavira, A. Darwiche, On probabilistic inference by weighted model counting, Artif. Intell. 172 (2008) 772–799.
[71] W. Faber, G. Pfeifer, N. Leone, Semantics and complexity of recursive aggregates in answer set programming, Artif. Intell. 175 (2011) 278–298, https://doi.org/10.1016/j.artint.2010.04.002.
[72] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: R.A. Kowalski, K.A. Bowen (Eds.), Logic Programming, Proceedings of the Fifth International Conference and Symposium (2 Volumes), Seattle, Washington, USA, August 15-19, 1988, MIT Press, 1988, pp. 1070–1080.
[73] H.L. Bodlaender, A partial k-arboretum of graphs with bounded treewidth, Theor. Comput. Sci. 209 (1998) 1–45.
[74] M. Droste, P. Gastin, Weighted automata and weighted logics, Theor. Comput. Sci. 380 (2007) 69–86, https://doi.org/10.1016/j.tcs.2007.02.055.
[75] V. Belle, L. De Raedt, Semiring programming: a semantic framework for generalized sum product problems, Int. J. Approx. Reason. 126 (2020) 181–201, https://doi.org/10.1016/j.ijar.2020.08.001.
[76] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, H. Fargier, Semiring-based csps and valued csps: frameworks, properties, and comparison, Constraints 4 (1999) 199–240, https://doi.org/10.1023/A:1026441215081.
[77] T.J. Green, G. Karvounarakis, V. Tannen, Provenance semirings, in: L. Libkin (Ed.), Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Beijing, China, ACM, June 11–13, 2007, 2007, pp. 31–40.
[78] F. Bacchus, J. Berg, M. Järvisalo, R. Martins (Eds.), MaxSAT Evaluation 2020: Solver and Benchmark Descriptions, Department of Computer Science Report Series B, vol. B-2020–2, Department of Computer Science, University of Helsinki, Finland, 2020.
[79] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Clingo = ASP + control: preliminary report, CoRR, arXiv:1405.3694 [abs], 2014, http://arxiv.org/abs/1405.3694, arXiv:1405.3694.
[80] U. Oztok, A. Choi, A. Darwiche, Solving $\mathrm{pp}^{\mathrm{pp}}$-complete problems using knowledge compilation, in: C. Baral, J.P. Delgrande, F. Wolter (Eds.), Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016, AAAI Press, 2016, pp. 94–103, http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12910.
[81] D. Azzolini, F. Riguzzi, Inference in probabilistic answer set programming under the credal semantics, in: R. Basili, D. Lembo, C. Limongelli, A. Orlandini (Eds.), AIxIA 2023 - Advances in Artificial Intelligence - XXIInd International Conference of the Italian Association for Artificial Intelligence, Proceedings, AIxIA 2023, Rome, Italy, November 6-9, 2023, in: Lecture Notes in Computer Science, vol. 14318, Springer, 2023, pp. 367–380.
[82] T. Sato, A statistical learning method for logic programs with distribution semantics, in: L. Sterling (Ed.), Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13–16, 1995, MIT Press, 1995, pp. 715–729.
[83] F. Riguzzi, T. Swift, A survey of probabilistic logic programming, in: M. Kifer, Y.A. Liu (Eds.), Declarative Logic Programming: Theory, Systems, and Applications, ACM / Morgan & Claypool, 2018, pp. 185–228.
[84] R. Kiesel, T. Eiter, Knowledge compilation and more with sharpsat-td, in: P. Marquis, T.C. Son, G. Kern-Isberner (Eds.), Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023, 2023, pp. 406–416.
[85] D.S. Shterionov, J. Renkens, J. Vlasselaer, A. Kimmig, W. Meert, G. Janssens, The most probable explanation for probabilistic logic programs with annotated disjunctions, in: J. Davis, J. Ramon (Eds.), Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 9046, Springer, 2014, pp. 139–153.
[86] E. Tsamoura, V. Gutiérrez-Basulto, A. Kimmig, Beyond the grounding bottleneck: datalog techniques for inference in probabilistic logic programs, in: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, AAAI Press, 2020, pp. 10284–10291, https://ojs.aaai.org/index.php/AAAI/article/view/6591, 2020.
[87] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman, Magic sets and other strange ways to implement logic programs, in: Proceedings of the acm Symposium on Principles of Database Systems, Cambridge, Massachusetts, March 1986, pp. 24–26.
[88] D. Shterionov, Design and Development of Probabilistic Inference Pipelines, Ph.D. thesis, KU Leuven, 2015.
[89] R.A. Aziz, G. Chu, C.J. Muise, P.J. Stuckey, Stable model counting and its application in probabilistic logic programming, in: B. Bonet, S. Koenig (Eds.), Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA, AAAI Press, 2015, pp. 3468–3474, http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9709.
[90] F. Lin, Y. Zhao, ASSAT: computing answer sets of a logic program by SAT solvers, Artif. Intell. 157 (2004) 115–137, https://doi.org/10.1016/j.artint.2004.04.004.
[91] V. Lifschitz, A.A. Razborov, Why are there so many loop formulas?, ACM Trans. Comput. Log. 7 (2006) 261–268, https://doi.org/10.1145/1131313.1131316.
[92] M. Gebser, J. Lee, Y. Lierler, On elementary loops of logic programs, Theory Pract. Log. Program. 11 (2011) 953–988, https://doi.org/10.1017/S1471068411000019.
[93] E. Giunchiglia, Y. Lierler, M. Maratea, Answer set programming based on propositional satisfiability, J. Autom. Reason. 36 (2006) 345–377, https://doi.org/10.1007/s10817-006-9033-2.
[94] A.M.C.A. Koster, H.L. Bodlaender, S.P.M. van Hoesel, Treewidth: computational experiments, Electron. Notes Discrete Math. 8 (2001) 54–57, https://doi.org/10.1016/S1571-0653(05)80078-2.
[95] I. Niemelä, Stable models and difference logic, Ann. Math. Artif. Intell. 53 (2008) 313–329, https://doi.org/10.1007/s10472-009-9118-9.
[96] M. Björk, Successful SAT encoding techniques, J. Satisf. Boolean Model. Comput. 7 (2011) 189–201, https://doi.org/10.3233/sat190085.
[97] M.H. van Emden, R.A. Kowalski, The semantics of predicate logic as a programming language, J. ACM 23 (1976) 733–742, https://doi.org/10.1145/321978.321991.
[98] J. Guo, F. Hüffner, R. Niedermeier, A structural view on parameterizing problems: distance from triviality, in: R.G. Downey, M.R. Fellows, F.K.H.A. Dehne (Eds.), Parameterized and Exact Computation, First International Workshop, Proceedings, IWPEC 2004, Bergen, Norway, September 14-17, 2004, in: Lecture Notes in Computer Science, vol. 3162, Springer, 2004, pp. 162–173.
[99] Lark, Lark - a parsing toolkit for python, https://github.com/lark-parser/lark, 2017.
[100] L. De Raedt, A. Kimmig, Probabilistic (logic) programming concepts, Mach. Learn. 100 (2015) 5–47, https://doi.org/10.1007/s10994-015-5494-z.
[101] lp2lp, Asp tools, https://research.ics.aalto.fi/software/asp/lp2sat/, 2006.
[102] H. Dell, T. Husfeldt, B.M.P. Jansen, P. Kaski, C. Komusiewicz, F.A. Rosamond, The first parameterized algorithms and computational experiments challenge, in: J. Guo, D. Hermelin (Eds.), 11th International Symposium on Parameterized and Exact Computation, IPEC 2016, Aarhus, Denmark, August 24–26, 2016, in: LIPIcs, vol. 63, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 30.
[103] K. Kiljan, M. Pilipczuk, Experimental evaluation of parameterized algorithms for feedback vertex set, in: G. D'Angelo (Ed.), 17th International Symposium on Experimental Algorithms, SEA 2018, L'Aquila, Italy, June 27–29, 2018, in: LIPIcs, vol. 103, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 12.
[104] M. Hamann, B. Strasser, Graph bisection with Pareto optimization, ACM J. Exp. Algorithmics 23 (2018), https://doi.org/10.1145/3173045.
[105] H. Tamaki, Positive-instance driven dynamic programming for treewidth, J. Comb. Optim. 37 (2019) 1283–1311, https://doi.org/10.1007/s10878-018-0353-z.

[106] M. Abseher, N. Musliu, S. Woltran, htd – a free, open-source framework for (customized) tree decompositions and beyond, in: D. Salvagnin, M. Lombardi (Eds.), Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, Proceedings, CPAIOR 2017, Padua, Italy, June 5-8, 2017, in: Lecture Notes in Computer Science, vol. 10335, Springer, 2017, pp. 376–386.

[107] B. Strasser, Computing tree decompositions with flowcutter: PACE 2017 submission, CoRR, arXiv:1709.08949 [abs], 2017, http://arxiv.org/abs/1709.08949, arXiv:1709.08949.

[108] J. Lind-Nielsen, Buddy: a binary decision diagram package, http://sourceforge.net/projects/buddy/, 1999.

[109] C.J. Muise, S.A. McIlraith, J.C. Beck, E.I. Hsu Dsharp, Fast d-dnnf compilation with sharpsat, in: L. Kosseim, D. Inkpen (Eds.), Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Proceedings, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012, in: Lecture Notes in Computer Science, vol. 7310, Springer, 2012, pp. 356–361.

[110] J.K. Fichte, M. Hecher, F. Hamiti, The model counting competition 2020, ACM J. Exp. Algorithmics 26 (2021) 13, https://doi.org/10.1145/3459080.

[111] J.K. Fichte, M. Hecher, Model counting competition 2021, https://mccompetition.org, 2021.

[112] R. Albert, A. Barabási, Statistical mechanics of complex networks, CoRR, arXiv:cond-mat/0106096, 2001, http://arxiv.org/abs/cond-mat/0106096.

[113] W. Meert, Pysdd: python package for sentential decision diagrams (sdd), https://github.com/wannesm/PySDD, 2018.

[114] A. Darwiche, P. Marquis, D. Suciu, S. Szeider, PySDD, in: Recent Trends in Knowledge Compilation, Dagstuhl Seminar 17381, Dagstuhl Rep. 7 (9) (2017) 81–82, https://doi.org/10.4230/DagRep.7.9.62.

[115] J. Lagniez, P. Marquis, Preprocessing for propositional model counting, in: C.E. Brodley, P. Stone (Eds.), Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada, AAAI Press, 2014, pp. 2688–2694, http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8264.

[116] F. Slivovsky, S. Szeider, A faster algorithm for propositional model counting parameterized by incidence treewidth, in: L. Pulina, M. Seidl (Eds.), Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Proceedings, Alghero, Italy, July 3-10, 2020, in: Lecture Notes in Computer Science, vol. 12178, Springer, 2020, pp. 267–276.

[117] M. Alviano, W. Faber, Aggregates in answer set programming, Künstl. Intell. 32 (2018) 119–124, https://doi.org/10.1007/s13218-018-0545-9.

[118] T. Janhunen, Cross-translating answer set programs using the ASPTOOLS collection, Künstl. Intell. 32 (2018) 183–184, https://doi.org/10.1007/s13218-018-0529-9.

[119] J. Bomanson, M. Gebser, T. Janhunen, Improving the normalization of weight rules in answer set programs, in: E. Fermé, J. Leite (Eds.), Logics in Artificial Intelligence - 14th European Conference, Proceedings, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014, in: Lecture Notes in Computer Science, vol. 8761, Springer, 2014, pp. 166–180.

[120] V. Derkinderen, P.Z.D. Martires, S. Kolb, P. Morettin, Top-down knowledge compilation for counting modulo theories, CoRR, arXiv:2306.04541 [abs], 2023, 10.48550/arXiv.2306.04541, arXiv:2306.04541.

[121] T. Kloks Treewidth, Computations and Approximations, Lecture Notes in Computer Science, vol. 842, Springer, 1994.

[122] S.B. Akers, Binary decision diagrams, IEEE Trans. Comput. 27 (1978) 509–516.

[123] E. Korach, N. Solel, Tree-width, path-width, and cutwidth, Discrete Appl. Math. 43 (1993) 97–101.