TPLP **24** (3): 505–532, 2024.

doi:10.1017/S1471068424000036 First published online 21 February 2024

*IASCAR: Incremental Answer Set Counting by Anytime Refinement**

JOHANNES K. FICHTE
Department of Computer Science (IDA), Linköping University, Linköping, Sweden
(*e-mail:* johannes.fichte@liu.se)

SARAH ALICE GAGGL
TU Dresden, Dresden, Germany
(*e-mail:* sarah.gaggl@tu-dresden.de)

MARKUS HECHER
Massachusetts Institute of Technology, Cambridge, MA 02139, USA
(*e-mail:* hecher@mit.edu)

DOMINIK RUSOVAC
TU Dresden, Dresden, Germany
(*e-mail:* dominik.rusovac@tu-dresden.de)

submitted 23 January 2023; revised 14 November 2023; accepted 08 January 2024

Abstract

Answer set programming (ASP) is a popular declarative programming paradigm with various applications. Programs can easily have many answer sets that cannot be enumerated in practice, but counting still allows quantifying solution spaces. If one counts under assumptions on literals, one obtains a tool to comprehend parts of the solution space, so-called *answer set navigation*. However, navigating through parts of the solution space requires counting many times, which is expensive in theory. *Knowledge compilation* compiles instances into representations on which counting works in polynomial time. However, these techniques exist only for conjunctive normal form (CNF) formulas, and compiling ASP programs into CNF formulas can introduce an exponential overhead. This paper introduces a technique to iteratively count answer sets under assumptions on knowledge compilations of CNFs that encode supported models. Our anytime technique uses the inclusion–exclusion principle to improve bounds by over- and undercounting systematically. In a preliminary empirical analysis, we demonstrate promising results. After compiling the input (offline phase), our approach quickly (re)counts.

KEYWORDS: ASP, answer set counting, knowledge compilation

* Research was funded by the BMBF, Grant 01IS20056_NAVAS, by ELLIIT funded by the Swedish government, by the Austrian Science Fund (FWF) grants J4656, P32830, and Y1329. The authors gratefully acknowledge the GWK support for funding this project by providing computing time through the Center for Information Services and HPC (ZIH) at TU Dresden. Additional computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) at Linköping partially funded by the Swedish Research Council through grant agreement no. 2022-06725.

1 Introduction

Answer set programming (ASP) (Marek and Truszczyński 1999; Niemelä 1999; Brewka *et al.* 2011) is a widely used declarative problem modeling and solving paradigm with many applications in artificial intelligence (AI) such as knowledge representation, planning, and many more (Baral 2003; Pontelli *et al.* 2012). It is widely used to solve difficult search problems while allowing compact modeling (Gebser *et al.* 2012). In ASP, a problem is represented as a set of rules, called *logic program*, over atoms. Models of a program under the stable semantics (Gelfond and Lifschitz 1988, 1991) form its solutions, so-called *answer sets*. Beyond the search for one solution or an optimal solution, an increasingly popular question is counting answer sets, which provides extensive applications for quantitative reasoning. For example, counting is crucial for probabilistic logic programming, c.f., Fierens *et al.* (2015), Wang and Lee (2015), Lee and Wang (2015) or encoding Bayesian networks and their inference (Sang *et al.* 2005).

Interestingly, counting also facilitates more fine-grained reasoning modes between brave and cautious reasoning. To this end, one examines the ratio of an atom occurring in answer sets over all answer sets, which yields a notion of *plausibility* of an atom. When considering sets of literals, which represent assumptions, one obtains a detailed tool to *comprehend search spaces* that contain a large number of answer sets (Fichte *et al.* 2022b), for example, for configuration problems (Dimopoulos *et al.* 1997; Lifschitz 1999; Nogueira *et al.* 2001). However, already for ground normal programs, answer set counting is #·P-complete (Fichte *et al.* 2017), making it harder than decision problems. Recall that brave reasoning is just NP-complete, but by Toda's Theorem we know that $\mathrm{PH} \subseteq \mathrm{P}^{\#\cdot\mathrm{P}}$ (Toda 1991) where $\bigcup_{k\in\mathbb{N}} \Delta_k^P = \mathrm{PH}$ and $\mathrm{NP} \subseteq \Delta_2^P = \mathrm{P}^{\mathrm{NP}}$ (Stockmeyer 1976). Approximate counting is in fact easier, that is, approx-$\#\cdot\mathrm{P} \subseteq \mathrm{BPP}^{\mathrm{NP}} \subseteq \Sigma_3^P$ (Lautemann 1983; Sipser 1983; Stockmeyer 1983), and approximate answer set counters have very recently been suggested (Kabir *et al.* 2022). Still, when navigating large search spaces, we need to count answer sets many times rendering such tools conceptually ineffective. There, knowledge compilation comes in handy (Darwiche 2004).

In *knowledge compilation*, computation is split in two phases. Formulas are compiled in a potentially very expensive step into a representation in an *offline phase* and reasoning is carried out in polynomial time on such representations in an *online phase*. Such a conceptual framework would be perfectly suited when answer sets are counted many times, providing us with quick re-counting. While we can translate programs into propositional formulas (Lee and Lifschitz 2003; Lee 2005; Janhunen and Niemelä 2011) and directly apply techniques from propositional formulas (Lagniez and Marquis 2017a), it is widely known that one can easily run into an exponential blowup (Lifschitz and Razborov 2006) or introduce level mappings (Janhunen 2006) that are oftentimes large grids and hence expensive for counters. In practice, solvers that find one answer set or optimal answer sets can avoid a blowup by computing supported models, which can be encoded into propositional formulas with limited overhead, and implementing propagators on top (Gebser *et al.* 2009).

In this paper, we explore a counterpart of a propagator-style approach for counting answer sets. We encode finding supported models as a propositional formula and use a knowledge compiler to obtain, in an offline phase, a representation, which allows us to construct a counting graph that in turn can be used to compute the number of supported

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

models efficiently. The resulting counting graph can be large but evaluated in parallel. Counting supported models only provides an upper bound on the number of answer sets. Therefore, we suggest a combinatorial technique to systematically improve bounds by over- and undercounting while incorporating the external support, whose absence can be seen as the cause of overcounting in the first place. Our technique can be used to approximate the counts but also provides the exact count on the number of answer sets when taking the entire external support into account.

Contributions. Our main contributions are as follows.

1. We consider knowledge compilation from an ASP perspective. We recap features such as counting under assumptions, known as conditioning, that make knowledge compilations (sd-DNNFs) quite suitable for navigating search spaces. We suggest a domain-specific technique to compress counting graphs that were constructed for supported models using Clark's completion.
2. We establish a novel combinatorial algorithm that takes an sd-DNNF of a completion formula and allows for systematically improving bounds by over- and undercounting. The technique identifies not supported atoms and compensates for overcounting on the sd-DNNF.
3. We apply our approach to instances tailored to navigate incomprehensible answer set search spaces. While the problem is challenging in general, we demonstrate feasibility and promising results on quickly (re-)counting.

Related Works. Previous work (Bogaerts and den Broeck 2015) considered knowledge compilation for logic programs. There an eager incremental approximation technique incrementally computes the result whereas our approach can be seen as an incremental lazy approach on the counting graph. Moreover, the technique by Bogarts and Broeck focuses on well-founded models and stratified negation, which does not work for normal programs in general without translating ASP programs into conjunctive normal forms (CNFs) directly. Note that common reasoning problems on answer set programs without negation can be solved in polynomial time (Truszczyński 2011). Model counting can significantly benefit from preprocessing techniques (Lagniez *et al.* 2016; Lagniez and Marquis 2014), which eliminate variables. Widely used propositional knowledge compilers are c2d (Darwiche 2004) and d4. Very recent works consider enumerating answer sets (Alviano *et al.* 2023), which can be beneficial for counting if the number of answer sets is sufficiently low. More advanced enumeration techniques have also recently been studied for propositional satisfiability (Masina *et al.* 2023; Spallitta et al. 2023).

Prior Work. This paper extends the conference publication (Fichte *et al.* 2022a). The paper contains more elaborate examples and proofs that have been omitted in the preliminary version. We now provide an empirical evaluation on relevant instances and instances that have been used for counting in previous works. We formulate detailed questions and hypotheses for our algorithm's implementation and evaluation. Now, our evaluation incorporates two instance sets containing a large number of instances, and we compare our approach to state-of-the-art model counters.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

2 Preliminaries

We assume familiarity with propositional satisfiability (Kleine Büning and Lettmann 1999), graph theory (Bondy and Murty 2008), and propositional ASP (Gebser *et al.* 2012). Recall that a *cycle* C on a (di)graph G is a (directed) walk of G where the first and the last vertex coincide. For cycle C, we let V_C be its vertices and $\mathit{cycles}(G) \coloneqq \{V_C \mid C \text{ is a cycle of } G\}$. We consider propositional *variables* and mean by formula a propositional formula. By $\top$ and $\bot$ we refer to the variables that are always evaluated to 1 or 0 (constants). A literal is an atom a or its negation $\neg a$, and $\mathrm{vars}(\varphi)$ denotes the set of variables that occur in formula φ. The set of models of a formula φ is given by $\mathcal{M}(\varphi)$. Below, we introduce the necessary background and notation used in the paper for ASP, and knowledge compilation.

Answer Set Programming. Let us recall basic notions of ASP, for further details we refer to standard texts (Gebser *et al.* 2012). In the context of ASP, we usually say atom instead of variable. A *(propositional logic) program* Π is a finite set of *rules* r of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

where $0 \leq m \leq n$ and $a_0, \ldots, a_n$ are atoms and usually omit $\top$ and $\bot$. For a rule r, we define $H(r) \coloneqq \{a_0\}$ called *head* of r. The *body* consists of $B^+(r) \coloneqq \{a_1, \ldots, a_m\}$ and $B^-(r) \coloneqq \{a_{m+1}, \ldots, a_n\}$. The set $\mathit{at}(r)$ of atoms of r consists of $H(r) \cup B^+(r) \cup B^-(r)$. Let Π be a program. Then, we let the set $\mathit{at}(\Pi) \coloneqq \bigcup_{r \in \Pi} \mathit{at}(r)$ of Π contain its atoms. Its *positive dependency digraph* $DP(\Pi) = (V, E)$ is defined by $V \coloneqq \mathit{at}(\Pi)$ and $E \coloneqq \{(a_1, a_0) \mid a_1 \in B^+(r), a_0 \in H(r), r \in \Pi\}$. The *cycles of* Π are given by $\mathit{cycles}(\Pi) \coloneqq \mathit{cycles}(DP(\Pi))$. Π is *tight*, if $DP(\Pi)$ is acyclic. An *interpretation* of Π is a set $I \subseteq \mathit{at}(\Pi)$ of atoms. I *satisfies* a rule $r \in \Pi$ if $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. I satisfies Π, if I satisfies each rule $r \in \Pi$. The *GL-reduct* Π_I is defined by $\Pi_I \coloneqq \{H(r) \leftarrow B^+(r) \mid I \cap B^-(r) = \emptyset, r \in \Pi\}$. I is an *answer set*, sometimes also called stable model, if I satisfies Π_I and I is subset-minimal. The *completion* (Clark 1978) of Π is the propositional formula

$$\mathrm{comp}(\Pi) \coloneqq \bigwedge_{a \in \mathit{at}(\Pi)} a \leftrightarrow \bigvee_{r \in \Pi, H(r) = a} BF(r)$$

where

$$BF(r) \coloneqq \bigwedge_{b \in B^+(r)} b \wedge \bigwedge_{c \in B^-(r)} \neg c.$$

where, as usual, the conjunction for an empty set is understood as $\top$ and the empty disjunction as $\bot$. An interpretation I is a *supported model* (Apt *et al.* 1988) of Π, if it is a model of the formula $\mathrm{comp}(\Pi)$. Let $\mathcal{S}(\Pi)$ be the set of all supported models of Π. It holds that $\mathcal{AS}(\Pi) \subseteq \mathcal{S}(\Pi)$ (Marek and Subrahmanian 1992), but not vice-versa. If Π is tight, then $\mathcal{AS}(\Pi) = \mathcal{S}(\Pi)$ (Fages 1994). In practice, we use the completion in CNF, thereby introducing auxiliary variables and still preserving the number of supported models.

Example 1
Let $\Pi_1 = \{a \leftarrow b; b \leftarrow; c \leftarrow c\}$. We see that $DP(\Pi_1)$ is cyclic due to rule $c \leftarrow c$. Thus, Π_1 is not tight and its respective answer sets $\mathcal{AS}(\Pi_1) = \{\{a, b\}\}$ and supported models $\mathcal{S}(\Pi_1) = \{\{a, b\}, \{a, b, c\}\}$ differ. △

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Assumptions. We define $\neg L \coloneqq \{\neg a \mid a \in L\}$ for a set L of literals and assume that $\neg\neg a$ stands for a. Let Π be a program and $\mathcal{L}(\Pi) \coloneqq at(\Pi) \cup \neg at(\Pi)$ be its literals. An *assumption* is a literal $\ell \in \mathcal{L}(\Pi)$ interpreted as rule $\mathrm{ic}(\ell) \coloneqq \{\bot \leftarrow \neg\ell\}$. For set L of assumptions of Π, we say that L is *consistent*, if there is no atom $a \in L$ for which $\neg a \in L$. Throughout this paper, by L we refer to consistent assumptions. Furthermore, we define $\mathrm{ic}(L) \coloneqq \bigcup_{\ell \in L} \mathrm{ic}(\ell)$ and let $\Pi[L] \coloneqq \Pi \cup \mathrm{ic}(L)$.

Example 2
Consider program Π_1 from Example 1, with $\mathcal{AS}(\Pi_1) = \{\{a, b\}\}$. For $L_1 \subseteq \{a, b, \neg c\}$, we obtain the same answer sets, that is, $\mathcal{AS}(\Pi_1) = \mathcal{AS}(\Pi_1[L_1])$. However, for any $L_2 \not\subseteq \{a, b, \neg c\}$ we obtain $\mathcal{AS}(\Pi_1[L_2]) = \emptyset$. △

Knowledge Compilation and Counting on Formulas in sd-DNNF. Let φ be a formula, φ is in *negation normal form (NNF)* if negations ($\neg$) occur only directly in front of variables and the only other operators are conjunction ($\wedge$) and disjunction ($\vee$) (Robinson and Voronkov 2001). NNFs can be represented in terms of *rooted directed acyclic graphs* (DAGs) where each leaf node is labeled with a literal, and each internal node is labeled with either a conjunction ($\wedge$-*node*) or a disjunction ($\vee$-*node*).

We use an NNF and its DAG interchangeably. The *size of an NNF* φ, denoted by $|\varphi|$, is given by the number of edges in its DAG. Formula φ is in *DNNF*, if it is in NNF and it satisfies the *decomposability* property, that is, for any distinct subformulas ψ_i, ψ_j in a conjunction $\psi = \psi_1 \wedge \cdots \wedge \psi_n$ with $i \neq j$, we have $\mathrm{vars}(\psi_i) \cap \mathrm{vars}(\psi_j) = \emptyset$ (Darwiche 2004). Formula φ is in *d-DNNF*, if it is in DNNF and it satisfies the *decision* property, that is, disjunctions are of the form $\psi = (x \wedge \psi_1) \vee (\neg x \wedge \psi_2)$. Note that x does not occur in ψ_1 and ψ_2 because of decomposability. ψ_1 and ψ_2 may be conjunctions. Formula φ is in *sd-DNNF*, if all disjunctions in ψ are smooth, meaning for $\psi = \psi_1 \vee \psi_2$ we have $\mathrm{vars}(\psi_1) = \mathrm{vars}(\psi_2)$.

Determinism and smoothness permit traversal operations on sd-DNNFs to count models of φ in linear time in $|\varphi|$ (Darwiche 2001). The traversal takes place on the so-called counting graph of an sd-DNNF. The *counting graph* $\mathcal{G}(\varphi)$ is the DAG of φ where each node N is additionally labeled by $val(N) \coloneqq 1$, if N consists of a literal; labeled by $val(N) \coloneqq \Sigma_i val(N_i)$, if N is an $\vee$-node with children N_i; labeled by $val(N) \coloneqq \Pi_i val(N_i)$, if N is an $\wedge$-node. By $val(\mathcal{G}(\varphi))$ we refer to $val(N)$ for the root N of $\mathcal{G}(\varphi)$. Function val can be constructed by traversing $\mathcal{G}(\varphi)$ in post-order in polynomial time.

It is well-known that $val(\mathcal{G}(\varphi))$ equals the model count of φ. For a set L of literals, counting of $\varphi^L \coloneqq \varphi \wedge \bigwedge_{\ell \in L} \ell$ can be carried out by *conditioning* of φ on L (Darwiche 1999). Therefore, the function val on the counting graph is modified by setting $val(N) = 0$, if N consists of ℓ and $\neg\ell \in L$. This corresponds to replacing each literal ℓ of the NNF φ by constant $\bot$ or $\top$, respectively. From now on, we denote by $\Phi_{\Pi[L]}$ an equivalent sd-DNNF of $\mathrm{comp}(\Pi[L])$ and its counting graph by $\mathcal{G}_{\Pi[L]}$. Note that $\Pi[L] = \Pi$ for $L = \emptyset$. The conditioning of $\mathcal{G}_\Pi$ on L is denoted by $(\mathcal{G}_\Pi)^L$.

3 Counting supported models

In our applications mentioned in the introduction, we are interested in counting multiple times under assumptions. In other words, we count the total number of answer sets and the number of answer sets under various changing assumptions. Therefore, we extend known techniques from knowledge compilation (Darwiche and Marquis 2002).

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

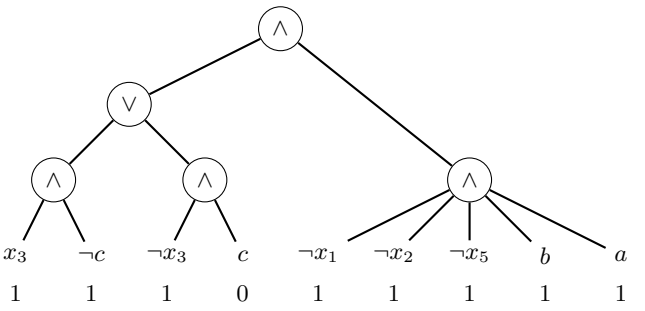

Fig. 1. Counting graph $\mathcal{G}(\varphi \wedge \neg c)$ labeled with literals and their respective value.

The general outline for a given program Π is as follows: (i) we construct the formula comp(Π) that can (ii) be compiled in a computationally expensive step into a formula $\Phi_{\text{comp}(\Pi)}$ in a normal form, so-called sd-DNNF by existing knowledge compilers. Then, (iii) on the sd-DNNF $\Phi_{\text{comp}(\Pi)}$ counting can be done in polynomial time in the size of $\Phi_{\text{comp}(\Pi)}$. We can even count under a set L of propositional assumptions by the technique known as conditioning.

However, this approach yields only the number of supported models under assumptions and we overcount compared to the number of answer sets. To this end, in Section 4, (iv) we present a technique to incrementally reduce the overcount.

In the following, we recall how knowledge compilation can be used to count formulas under assumptions by assuming that a formula is in sd-DNNF and constructing a counting graph.

Example 3
Consider the sd-DNNF $\varphi_1 = ((x_3 \wedge \neg c) \vee (\neg x_3 \wedge c)) \wedge (\neg x_1 \wedge \neg x_2 \wedge \neg x_5 \wedge a \wedge b)$. We observe in Figure 1 that its rooted DAG has 14 nodes, 7 variables, and 13 edges. In consequence, we have that $|\varphi_1| = 13$. By conditioning of φ on $L = \{\neg c\}$, each variable in L will be removed from $\mathcal{G}(\varphi_1)$ and we obtain $\varphi_1 \wedge \neg c = ((x_3 \wedge \neg\bot) \vee (\neg x_3 \wedge \bot)) \wedge (\neg x_1 \wedge \neg x_2 \wedge \neg x_5 \wedge a \wedge b)$. From Figure 1, we observe that the model count $val(\mathcal{G}(\varphi \wedge \neg c))$ of formula $\varphi \wedge \neg c$ is 1. △

Using the techniques as described above, we can compile the formula comp(Π) into an sd-DNNF $\Phi_{\text{comp}(\Pi)}$ and count the number $|\mathcal{S}(\Pi)|$ of supported models. We illustrate this in the following example.

Example 4
Consider Π_1 from Example 1. When constructing comp(Π_1) in CNF, we obtain 10 clauses with 4 new auxiliary variables x_1, x_2, x_3, and x_5. We can compile it into an sd-DNNF Φ_{Π_1} which is logically equivalent to comp(Π_1). For illustration purposes, we chose formula φ_1 from Example 3 such that Φ_{Π_1} is equivalent to φ_1. Hence, we can obtain the number $|\mathcal{S}(\Pi_1)|$ of supported models from $val(\mathcal{G}_{\Pi_1})$. △

3.1 Counting supported models under assumptions

Since assumptions of formulas and programs behave slightly differently due to the GL reduct, it is not immediately clear that we can use conditioning to obtain the number of

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

supported models of a program under given assumptions. In the following we will show that supported models of Π under assumptions L coincide with models of $\Phi_{\Pi[L]}$.

Observation 1
Let Π be a program and L assumptions. Then, $\mathcal{M}(\Phi_{\Pi[L]}) = \mathcal{S}(\Pi[L])$

For any program Π the conditioning $(\Phi_\Pi)^L$ on assumptions L allows us to identify supported models of a program $\Pi[L]$.

Lemma 3.1
Let Π be a program and L be assumptions. Then, $\mathcal{M}((\Phi_\Pi)^L) = \mathcal{S}(\Pi[L])$.

Proof
We first establish the following claim:

$$\mathrm{comp}(\Pi[L]) = \mathrm{comp}(\Pi \cup \mathrm{ic}(L)) = \mathrm{comp}(\Pi) \wedge \bigwedge_{\ell \in L} \ell \tag{1}$$

By definition, we have that $\mathrm{comp}(\Pi[L]) = \mathrm{comp}(\Pi \cup \mathrm{ic}(L))$. This further evaluates to $\mathrm{comp}(\Pi) \cup \mathrm{ic}(L)$. Since $\bot$ evaluates to false always and

$$\mathrm{comp}(\{\bot \leftarrow B(r)^+, \neg B(r)^- \mid r \in \Pi, H(r) = \bot\}) = \bot \leftrightarrow \bigvee_{r \in \Pi, H(r) = \bot} BF(r),$$

we obtain that

$$\mathcal{M}(\bot \leftrightarrow \bigvee_{r \in \Pi, H(r) = \bot} BF(r)) = \mathcal{M}(\bigwedge_{r \in \Pi, H(r) = \bot} \bot \leftrightarrow BF(r)), \tag{2}$$

$$= \mathcal{M}(\bigwedge_{r \in \Pi, H(r) = \bot} \neg BF(r)). \tag{3}$$

As a result,

$$\mathcal{M}(\mathrm{comp}(\Pi[L] \setminus \mathrm{ic}(L)) \cup \mathrm{ic}(L)) = \mathcal{M}(\mathrm{comp}(\Pi[L] \setminus \mathrm{ic}(L)) \cup \bigcup_{\ell \in L} \mathrm{comp}(\mathrm{ic}(\ell)) \tag{4}$$

$$= \mathcal{M}(\mathrm{comp}(\Pi) \wedge \bigwedge_{\ell \in L} \mathrm{comp}(\mathrm{ic}(\ell))) \tag{5}$$

$$= \mathcal{M}(\mathrm{comp}(\Pi) \wedge \bigwedge_{\ell \in L} \neg BF(\mathrm{ic}(\ell))) \tag{6}$$

$$= \mathcal{M}(\mathrm{comp}(\Pi) \wedge \bigwedge_{\ell \in L} \ell). \tag{7}$$

In consequence, equation (1) holds. It remains to show that conditioning $(\Phi_\Pi)^L$ in the sd-DNNF Φ_Π preserves all models according to Π under the set L of assumptions. By definition of conditioning, it holds that $\mathcal{M}((\Phi_\Pi)^L) = \mathcal{M}(\Phi_\Pi \wedge \bigwedge_{\ell \in L} \ell)$. By assumption, it is true that $\mathcal{M}(\Phi_\Pi \wedge \bigwedge_{\ell \in L} \ell) = \mathcal{M}(\mathrm{comp}(\Pi) \wedge \bigwedge_{\ell \in L} \ell)$. From equation (1), we obtain that $\mathcal{M}(\mathrm{comp}(\Pi) \wedge \bigwedge_{\ell \in L} \ell) = \mathcal{M}(\mathrm{comp}(\Pi[L]))$. By definition, $\mathcal{M}(\mathrm{comp}(\Pi[L])) = \mathcal{S}(\Pi[L])$. In consequence, we established that $\mathcal{M}((\Phi_\Pi)^L) = \mathcal{S}(\Pi[L])$. Hence, the Lemma sustains. □

Immediately, we obtain that we can count the number of supported models by first compiling the completion into an sd-DNNF and then applying conditioning. For tight programs, this already yields the number of answer sets.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Algorithm 1 Counting Graph Compression

In: sd-DNNF Φ_Π, $\mathcal{L}(\Pi)$
Out: Compressed counting graph $\tau(\mathcal{G}_\Pi)$

1: initialize array `t` and traverse nodes $N \in \Phi_\Pi$ bottom-up such that
2: **if** N contains a literal $\ell \in \mathcal{L}(\Pi)$ **then** label N with $val(N)$
3: **else if** N contains a literal $\ell \notin \mathcal{L}(\Pi)$ **then** mark N as `ignored`
4: **else** check the number of children of N that are not marked as `ignored`
5: **if** N has no remaining children **then** mark N as `ignored`
6: **else if** N has one remaining child C **then** $N \leftarrow C$ and mark N as `ignored`
7: **else** $v \leftarrow val(N)$ w.r.t. `t` and remaining children of N and label N with v
8: add N to `t`
9: remove all nodes marked with `ignored` from `t`
10: **return** `t`

Corollary 1
Let Π be a program and L be assumptions. Then,

$$val((\mathcal{G}_\Pi)^L) = |\mathcal{M}((\Phi_\Pi)^L)| = |\mathcal{S}(\Pi[L])|.$$

If Π is tight, also $val((\mathcal{G}_\Pi)^L) = |\mathcal{AS}(\Pi[L])|$ holds. Furthermore, counting can be done in time linear in $|\Phi_\Pi|$.

Example 5
Consider program Π_1 from Example 1, which has two supported models $\{a, b\}$ and $\{a, b, c\}$. Without setting $val(c)$ to 0 in Figure 1, we would obtain 2, which corresponds to these two models. By assumption $\neg c$, we set $val(c)$ to 0, which results in a total count of 1 as the $\wedge$-node gives only one count in the subgraph. △

3.2 Compressing counting graphs

When computing the counting graph of the completion of a program Π, in practice, we usually construct a CNF of the completion by introducing so-called nogoods (Gebser *et al.* 2012) similar to Tseitin's transformation (Tseytin 1983). It is well-known that there is a one-to-one correspondence, however, auxiliary variables are introduced, see, for example, Kuiter *et al.* (2023). For counting, the one-to-one correspondence immediately allows to establish a bijection between the models of the CNF and the supported models making it practicable on CNFs.

However, from Corollary 1, we know that the runtime counting models on $(\mathcal{G}_\Pi)^L$ depends on the size of Φ_Π. In consequence, introducing auxiliary variables affects the runtime of our approach. To this end, we introduce a compressing technique in Algorithm 1 that takes a counting graph $\mathcal{G}_\Pi$ and produces a *compressed counting graph* (CCG) $\tau(\mathcal{G}_\Pi)$, thereby removing auxiliary variables that have been introduced by the Tseitin transformation. The algorithm takes as input an sd-DNNF Φ_Π, and literals $\mathcal{L}(\Pi)$; and returns the CCG $\tau(\mathcal{G}_\Pi)$. In Line 3, we check whether the literal node consists of an auxiliary variable, and if so, it will be ignored. The case distinction in Lines 5–7 distinguishes how many not ignored children a non-literal node still has. Remember that each non-literal node is either an $\wedge$-node or an $\vee$-node. In Line 5, the node can be removed, as it has no child. In Line 6, the node needs to be absorbed, as it has only one child meaning that

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

the node ultimately becomes its child. In all other cases (Line 7), the node needs to be evaluated on the CCG `t` such that the ignored nodes are treated as neutral element of the respective sum or product. Ignored nodes are then removed from `t`. It remains to show that compressing $\mathcal{G}_\Pi$ leaves *val* unchanged, which is the topic of the following statement and subsequent proof.

Lemma 3.2
Let Π be a program, Φ_Π an sd-DNNF of comp(Π) after a transformation that preserves the number of models, but introduces auxiliary variables, and $\mathcal{G}_\Pi$ its counting graph. Then, $val(\tau(\mathcal{G}_\Pi)) = val(\mathcal{G}_\Pi)$ and $\tau(\mathcal{G}_\Pi)$ can be constructed in time $\mathcal{O}(2 \cdot |\Phi_\Pi|)$.

Proof
Let $\mathcal{G}_\Pi$ be the counting graph of an sd-DNNF that is equivalent to the CNF that has been constructed from comp(Π) using a transformation that preserves the number of models, which usually is the Tseitin transformation. We show that the value $val(N)$ of each node N of $\mathcal{G}_\Pi$, which is not removed in $\tau(\mathcal{G}_\Pi)$, does not change, since for N and its respective children $children(N)$ in Algorithm 1 we modify only literals that occur in the program Π. By $N_\tau \in \tau(\mathcal{G}_\Pi)$ we denote the modified version of N, and by $children_\tau(N)$ we denote the children of N in $\tau(\mathcal{G}_\Pi)$. We distinguish the cases:

1. Suppose N is a literal node. Let ℓ denote the corresponding literal. If $\ell \notin \mathcal{L}(\Pi)$, then N is removed in $\tau(\mathcal{G}_\Pi)$, thus by contraposition, we know that, if N is not removed in $\tau(\mathcal{G}_\Pi)$, then $\ell \in \mathcal{L}(\Pi)$. Assume $\ell \in \mathcal{L}(\Pi)$. Then $N = N_\tau$. Therefore, $val(N) = val(N_\tau) \in \{0, 1\}$.
2. Suppose N is not a literal node. Then, since N is an $\wedge$- or an $\vee$-node, we know that $|children(N)| \geq 2$. However, in general $0 \leq |children_\tau(N)| \leq |children(N)|$.
   (a) Assume $|children_\tau(N)| = 0$. Then, in Algorithm 1, N will be ignored and thus not belong to $\tau(\mathcal{G}_\Pi)$.
   (b) Assume $|children_\tau(N)| = 1$. Then, in Algorithm 1, N will be absorbed by its only child. Thus, N does not belong to $\tau(\mathcal{G}_\Pi)$.
   (c) Assume $|children_\tau(N)| \geq 2$. Then in Algorithm 1, N will be evaluated on $children_\tau(N)$, which means N_τ will be contained in $\tau(\mathcal{G}_\Pi)$. We now need to show that $val(N)$ on $children(N)$ corresponds to $val(N)$ on $children_\tau(N)$, that is, $val(N) = val(N_\tau)$. By assumption (number of models is preserved), we have a bijection between $M(\Phi_\Pi)$ and $\mathcal{S}(\Pi)$ which ignores auxiliary variables. Therefore, we can simply set the values of children $children(N)$ that have been removed or absorbed due to Cases 2a, 2b, or 2c – as a consequence of removing auxiliary variables – to the corresponding neutral element of the value of N.
      i Assume N is an $\wedge$-node. Accordingly, in Algorithm 1, N will be evaluated on $children_\tau(N)$ such that in the product corresponding to $val(N)$, the value of each removed branch (removed child), due to removing auxiliary variables, corresponds to the neutral element of multiplication, that is, 1. Therefore, we conclude that $val(N) = val(N_\tau)$.
      ii Assume N is an $\vee$-node. Again, accordingly, in Algorithm 1, N will be evaluated on $children_\tau(N)$ such that in the sum corresponding to $val(N)$, the value of each removed branch (removed child), due to removing aux-

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

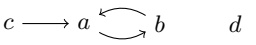

Fig. 2. The positive dependency graph of Π_2.

iliary variables, corresponds to the neutral element of addition, that is, 0. Therefore, $val(N) = val(N_\tau)$, which concludes the proof.

Inspecting Algorithm 1, we see that we require two traversals of the original counting graph, one from Lines 3–8 and another one in Line 9 where we remove the nodes that do not belong to the CCG. Runtime follows from the fact that we need to traverse Φ_Π twice. □

Corollary 2
Let Π be a tight program, then $val(\tau(\mathcal{G}_\Pi)) = |\mathcal{AS}(\Pi)|$.

4 Incremental counting by inclusion–exclusion

In the previous section, we illustrated how counting on tight programs works and introduced a technique to speed up practical counting. To count answer sets of a non-tight program, we need to distinguish supported models from answer sets on $\tau(\mathcal{G}_\Pi)$, which can become quite tedious. Therefore, we use the positive dependency graph $DP(\Pi)$ of Π. A set $X \subseteq at(\Pi)$ of atoms is an answer set, whenever it can be derived from Π in a finite number of steps. In particular, the mismatch between answer sets and supported models is caused by atoms $C \in cycles(\Pi)$ involved in cycles in $DP(\Pi)$ that are not supported by atoms from outside the cycle. We call those supporting atoms of C the *external support* of C.

Definition 1
Let Π be a program and $r \in \Pi$. An atom $a \in B^+(r)$ is an *external support* of $C \in cycles(\Pi)$, whenever $H(r) \subseteq C$ and $B^+(r) \cap C = \emptyset$. By $ES(C)$ we denote the set of all external supports of C.

Next, we illustrate the effect of external supports on the answer sets derivation.

Example 6
Let $\Pi_2 = \{a \leftarrow b; b \leftarrow a; a \leftarrow c; c \leftarrow \neg d; d \leftarrow \neg c\}$. The positive dependency graph of Π_2 is given in Figure 2. We obtain a cycle $C = \{a, b\}$ due to rules $a \leftarrow b$ and $b \leftarrow a$ with external support $ES(C) = \{c\}$ due to rule $a \leftarrow c$. However, due to rules $c \leftarrow \neg d$ and $d \leftarrow \neg c$, we see that whenever d is true, c is false, so that d deactivates the support of C, which means that $\{a, b, d\}$ cannot be derived from Π_2 in a finite number of steps. Accordingly, we have $\mathcal{S}(\Pi_2) = \{\{a, b, c\}, \{a, b, d\}, \{d\}\}$, but $\mathcal{AS}(\Pi_2) = \{\{a, b, c\}, \{d\}\}$. △

Note that external supports are sets of atoms. However, we can simulate such a set by introducing an auxiliary atom; hence one atom, as in this definition, is sufficient (Gebser *et al.* 2012).

Example 7
Let $a \leftarrow b$, $b \leftarrow a$, and $b \leftarrow c, \neg d$ be rules. Then the external support of atoms $\{a, b\}$, which are involved in cycles, is $\{c\}$. If instead of $b \leftarrow c, \neg d$ we use two alternative rules $b_r \leftarrow c, \neg d$ and $b \leftarrow b_r$, we have $ES(\{a, b\}) = \{b_r\}$. △

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

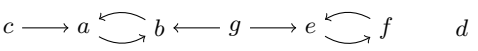

Fig. 3. The positive dependency graph of Π_3 from Example 8.

To approach the answer set count of a non-tight program under assumptions, we employ the well-known *inclusion–exclusion principle*, which is a counting technique to determine the number of elements in a finite union of finite sets $X_1, \ldots, X_n$. Therefore, first the cardinalities of the singletons are summed up. Then, to compensate for potential overcounting, the cardinalities of all intersections of two sets are subtracted. Next, the number of elements that appear in at least three sets are added back, that is, the cardinality of the intersection of all three sets – to compensate for potential undercounting – and so on. As an example, for three sets X_1, X_2, X_3 the procedure can be expressed as $|X_1 \cup X_2 \cup X_3| = |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3|$. This principle can be used to count answer sets via supported model counting.

Next we define a notion that is useful to identify or prune supported models that are not stable.

Definition 2
We define the *unsupported constraint* for a set $C = \{c_0, \ldots, c_n\} \in \mathit{cycles}(\Pi)$ of atoms involved in cycles and its respective external supports $ES(C) = \{s_0, \ldots, s_m\}$ by the rule $\lambda(C) \coloneqq \bot \leftarrow c_0, \ldots, c_n, \neg s_0, \ldots, \neg s_m$.

The unsupported constraints as defined here, (i) are inspired by *loop formulas* (Lin and Zhao 2004; Ferraris *et al.* 2006); and (ii) contain the whole set C, which is slightly weaker than constraints (nogoods) defined in related work (Gebser *et al.* 2012), but sufficient for characterizing answer sets.

Lemma 4.1
Let Π be a program with cycles $\mathit{cycles}(\Pi) = \{C_1, \ldots, C_n\}$, then

$$\mathcal{AS}(\Pi) = \mathcal{S}(\Pi \cup \{\lambda(C_1), \ldots, \lambda(C_n)\}).$$

Proof
Recall that $\mathcal{AS}(\Pi) \subseteq \mathcal{S}(\Pi)$. However, supported models – in particular those that are not answer sets – might contain a cycle $C = \{c_0, \ldots, c_m\} \in \mathit{cycles}(\Pi)$ without external support from $ES(C) = \{s_0, \ldots, s_k\}$, which are precisely those supported models we exclude by adding a rule

$$\bot \leftarrow c_0, \ldots, c_m, \neg s_0, \ldots, \neg s_k$$

in the form of unsupported constraints $\lambda(C)$ to Π for each $C \in \mathit{cycles}(\Pi)$. This ensures that atoms involved in cycles are not present without external support in any supported model, which provides us with supported models that are answer sets. □

Example 8
Let $\Pi_3 = \Pi_2 \cup \{b \leftarrow g; f \leftarrow g; e \leftarrow f; f \leftarrow e\}$, which has two cycles $C_0 = \{a, b\}$ and $C_1 = \{e, f\}$. Their corresponding external supports are $ES(C_0) = \{c, g\}$ and $ES(C_1) = \{g\}$. Accordingly, we have unsupported constraints $\lambda(C_0) = \bot \leftarrow a, b, \neg c, \neg g$ and $\lambda(C_1) = \bot \leftarrow e, f, \neg g$. Figure 3 illustrates the positive dependency graph of program Π_3. △

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Before we discuss our approach on incremental answer set counting, we need some further notation. From now on, by $\Lambda_d(\Pi) \coloneqq \{\{\lambda(C_1), \ldots, \lambda(C_d)\} \mid \{C_1, \ldots, C_d\} \subseteq \mathit{cycles}(\Pi)\}$ we denote the set of all combinations of unsupported constraints of cycles that occur in any subset of $\mathit{cycles}(\Pi)$ with cardinality $0 \leq d \leq n$, where $n \coloneqq |\mathit{cycles}(\Pi)|$. Further, we define body literals of a set of unsupported constraints Γ by $B(\Gamma) \coloneqq \bigcup\{B(\lambda(C)) \mid \lambda(C) \in \Gamma\}$.

Example 9 (*Continued*)
Consider program Π_3 from Example 8. We have $\Lambda_0(\Pi_3) = \emptyset$, $\Lambda_1(\Pi_3) = \{\{\lambda(C_0)\}, \{\lambda(C_1)\}\}$ and $\Lambda_2(\Pi_3) = \{\{\lambda(C_0), \lambda(C_1)\}\}$. △

Now, we define the *incremental count* of $|\mathcal{AS}(\Pi[L])|$ by a_d^L, using the combinatorial principle of inclusion–exclusion as follows:

$$a_d^L \coloneqq \sum_{i=0}^{d} (-1)^i \sum_{\Gamma \in \Lambda_i(\Pi)} |\mathcal{S}(\Pi[L \cup B(\Gamma)])| \tag{8}$$

$$= |\mathcal{S}(\Pi[L])| - \sum_{\Gamma \in \Lambda_1(\Pi)} |\mathcal{S}(\Pi[L \cup B(\Gamma)])| \tag{9}$$

$$+ \sum_{\Gamma \in \Lambda_2(\Pi)} |\mathcal{S}(\Pi[L \cup B(\Gamma)])| - \cdots + (-1)^d \sum_{\Gamma \in \Lambda_d(\Pi)} |\mathcal{S}(\Pi[L \cup B(\Gamma)])| \tag{10}$$

By subtracting $|\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])|$ for each $\Gamma \in \Lambda_1(\Pi)$ we subtract the number of supported models that are *not answer sets* under assumptions L with respect to each cycle $C \in \mathit{cycles}(\Pi)$. However, we need to take into account the interaction of cycles and their respective external supports under assumptions L. Thus we enter the first alternation step, where we proceed by adding back $|\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])|$ for each $\Gamma \in \Lambda_2(\Pi)$, which means that we add back the number of supported models that were mistakenly subtracted from $|\mathcal{S}(\Pi[L])|$ in the previous step, and so on, until we went through all Λ_i where $0 \leq i \leq d$. Note that therefore in total we have d alternations. In general, we show that $a_n^L = |\mathcal{AS}(\Pi[L])|$ as follows.

Theorem 1
Let Π be a program, $\mathit{cycles}(\Pi) = \{C_1, \ldots, C_n\}$, and further $U \coloneqq \{\lambda(C_1), \ldots, \lambda(C_n)\}$ be the set of all unsupported constraints of Π. Then, for assumptions L,

$$|\mathcal{S}(\Pi[L] \cup U)| = \sum_{i=0}^{n} (-1)^i \sum_{\Gamma \in \Lambda_i(\Pi)} |\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])|$$

Proof
We proceed by induction on $|\mathit{cycles}(\Pi)|$.

Induction Base Case: We assume that $|\mathit{cycles}(\Pi)| = 0$. Then, since Π admits no positive cycle in $DP(\Pi)$, we have $\mathcal{AS}(\Pi[L]) = \mathcal{S}(\Pi[L])$, and therefore $|\mathcal{AS}(\Pi[L])| = |\mathcal{S}(\Pi[L])|$.

Induction Hypothesis (IH): We assume that the proposition holds for every program Π with a number of cycles $|\mathit{cycles}(\Pi)| < m$.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Induction Step: We need to show that the result holds for a program Π with $|cycles(\Pi)| = m + 1$. Let $C' \in cycles(\Pi)$ be a cycle. We define $U_m \coloneqq \{\lambda(C_1), \ldots, \lambda(C_m)\}$ for any $\{C_1, \ldots, C_m\} \subseteq cycles(\Pi)$ such that $|U_m| = m$ with $C_i \neq C'$ for $C_i \in \{C_1, \ldots, C_m\}$. Then, by IH, we have that

$$x \coloneqq |\mathcal{S}(\Pi[L \cup B(U_m)])| = \sum_{i=0}^{m}(-1)^i \sum_{\Gamma \in \Lambda_i(\Pi), \lambda(C') \notin \Gamma} |\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])|$$

To x, the formula $\sum_{i=0}^{m+1}(-1)^i \sum_{\Gamma \in \Lambda_i(\Pi)} |\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])|$ adds $|\mathcal{S}(\Pi \cup \lambda(C'))|$. However, this formula then subtracts supported models satisfying both constraints $\{\lambda(C'), \lambda(C'')\}$ with one of the cycles $\lambda(C'') \in U_m$ twice, which require to be added back. Thus, we proceed by adding back supported models satisfying unsupported constraints of C' with two other cycles, which again have to be subtracted in the next step. In turn, the application of the inclusion–exclusion principle ensures that

$$\sum_{i=0}^{m+1}(-1)^i \sum_{\Gamma \in \Lambda_i(\Pi)} |\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])|$$

$$= x + \sum_{i=0}^{m+1}(-1)^i \sum_{\Gamma \in \Lambda_i(\Pi), \lambda(C') \in \Gamma} |\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])|.$$

□

Finally, one can count answer sets correctly.

Corollary 3
Let Π be a program, L assumptions, and $n = |cycles(\Pi)|$. Then, $a_n^L = |\mathcal{AS}(\Pi[L])|$.

In fact, we can characterize a_n^L with respect to alternation depths. If there is no change from one alternation to another, the point is reached where the number of answer sets is obtained, as the following lemma states.

Lemma 4.2
Let Π be a program and L be assumptions. If $a_i^L = a_{i+1}^L$ for some integer $i \geq 0$, then $a_i^L = |\mathcal{AS}(\Pi[L])|$.

Proof
Suppose $a_i^L = a_{i+1}^L$, then $\sum_{\Gamma \in \Lambda_{i+1}(\Pi)} |\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])| = 0$. We can observe that therefore no further combination of unsupported constraints with set L of assumptions where we combine unsupported constraints of cycles that occur in subsets of $cycles(\Pi)$ with cardinality $j > i+1$ points to any supported model. In other words, we have for all $j > i$ that $\sum_{\Gamma \in \Lambda_j(\Pi)} |\mathcal{S}(\Pi[L]) \setminus \mathcal{S}(\Pi[L \cup B(\Gamma)])| = 0$, which concludes the proof. □

Using our approach on computing a_n^L, we end up with 2^n (supported model) counting operations where $n \coloneqq |cycles(\Pi)|$ on the respective compressed counting graph $\tau(\mathcal{G}_\Pi)$, which, since counting is linear in $k \coloneqq |\tau(\mathcal{G}(\Pi))|$, gives us that incremental answer set counting under assumptions is by $2^n \cdot k$ exponential in time. However, we can restrict the alternation depth to d such that $0 \leq d < n$ in order to stop after $\Lambda_d(\Pi)$. Then we need to count n times for each cycle and its respective unsupported constraints and

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Algorithm 2 Incremental Counting by Anytime Refinement

In: Program Π; assumptions L; compressed counting graph $\tau(\mathcal{G}_\Pi)$; alternation depth d
Out: Incremental count a_d^L

1: `count` $\leftarrow val(\tau(\mathcal{G}_\Pi)^L)$ and $c \leftarrow 0$
2: **if** d is odd **then** $d \leftarrow d+1$
3: for every $1 \leq i \leq d$
4: **if** $c =$ `count` **then** break **else** $c \leftarrow$ `count`
5: for every $1 \leq j \leq i$
6: $c' \leftarrow val(\tau(\mathcal{G}_\Pi)^{L \cup L'})$ where L' is the set of literals appearing in $\Gamma_j \in \Lambda_i(\Pi)$
7: **if** i is odd **then** `count` $\leftarrow$ `count` $- c'$ **else** `count` $\leftarrow$ `count` $+ c'$
8: **return** `count`

another $\binom{n}{i}$ times for $1 < i \leq d$, that is, for each number of subsets of cycles and their respective unsupported constraints with cardinality i. These considerations yield the following result.

Theorem 2
Let Π be a program, L be assumptions, and $0 \leq d \leq n$ with $n \coloneqq |cycles(\Pi)|$. We can compute a_d^L in time $\mathcal{O}(m \cdot |\tau(\mathcal{G}(\Pi))|)$ where $m = \sum_{i \leq d} \binom{n}{i}$.

Note that if we choose an even d, we will stop on adding back, potentially overcounting, and otherwise we will stop on subtracting, potentially undercounting. Algorithm 2 ensures that we end on an add-operation to avoid undercounting in Line 2. Furthermore, it uses Lemma 4.2 as a termination criterion in Line 4.

Example 10
Consider program Π_3 from Example 8, which has 6 supported models, namely, $\{\{d\}, \{d,e,f\}, \{a,b,d\}, \{a,b,c\}, \{a,b,c,e,f\}, \{a,b,d,e,f\}\}$ of which $\{d\}$ and $\{a,b,c\}$ are answer sets. Suppose we want to determine $a_1^{\{d\}}$, then:

$$\begin{aligned} a_1^{\{d\}} &= |\mathcal{S}(\Pi[\{d\}])| - |\mathcal{S}(\Pi[\{d\} \cup B(\lambda(C_0))])| - |\mathcal{S}(\Pi[\{d\} \cup B(\lambda(C_1))])| \\ &= |\mathcal{S}(\Pi[\{d\}])| - |\mathcal{S}(\Pi[\{d,a,b,\neg c,\neg g\}])| - |\mathcal{S}(\Pi[\{d,e,f,\neg g\}])| \\ &= 4 - 2 - 2 = 0. \end{aligned}$$

We see that restricting the alternation depth to 1, leads to undercounting. However, not restricting the depth leads to the exact count as:

$$\begin{aligned} a_2^{\{d\}} &= a_1^{\{d\}} + |\mathcal{S}(\Pi[\{d\} \cup B(\{\lambda(C_0), \lambda(C_1)\})])| = a_1^{\{d\}} + |\mathcal{S}(\Pi[\{d,a,b,e,f,\neg c,\neg g\}])| \\ &= 0 + 1 = 1 = |\mathcal{AS}(\Pi_3[\{d\}])|. \end{aligned}$$

△

Preprocessing Cycles. When computing the incremental count a_i^L, we can implement a simple preprocessing step. Recall that an unsatisfiable propositional formula remains unsatisfiable when adding additional clauses (Kleine Büning and Lettmann 1999). Hence, if the conjunction of an unsupported constraint and assumption leads to an unsatisfiable formula, we can immediately obtain the resulting supported model count.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

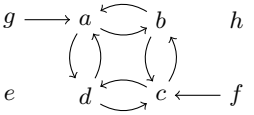

Fig. 4. The positive dependency graph of program Π_4 from Example 11.

Example 11
Consider program Π_4 given as follows:

$$\begin{aligned}\Pi_4 = \{&a \leftarrow b, & b \leftarrow a, & & b \leftarrow c, & & c \leftarrow b, \\ &a \leftarrow d, & d \leftarrow a, & & c \leftarrow d, & & d \leftarrow c, \\ &a \leftarrow g, & b \leftarrow \neg h, & & c \leftarrow f, & & d \leftarrow \neg e, \\ &e \leftarrow \neg g, & g \leftarrow \neg e, & & f \leftarrow \neg h, & & h \leftarrow \neg f\}.\end{aligned}$$

The supported models of Π_4 are $\mathcal{S}(\Pi_4) = \{\ \{e,h\},\ \{a,b,c,d,g,h\},\ \{a,b,c,d,f,g\},\ \{a,b,c,d,e,h\},\ \{a,b,c,d,e,f\}\}$. The answer sets of Π_4 are $\mathcal{AS}(\Pi_4) = \mathcal{S}(\Pi_4) \setminus \{\{a,b,c,d,e,h\}\}$. The program Π_4 admits eight cycles, which are illustrated in Figure 4 by the positive dependency graph of Π_4. Hence, the unsupported constraints of Π_4 are

$$\begin{aligned}&\lambda(C_0) = \bot \leftarrow a, b, \neg c, \neg d, \neg g, & &\lambda(C_1) = \bot \leftarrow b, c, \neg a, \neg d, \neg f, \\ &\lambda(C_2) = \bot \leftarrow c, d, \neg a, \neg b, \neg f, & &\lambda(C_3) = \bot \leftarrow a, b, c, \neg d, \neg f, \neg g, \\ &\lambda(C_4) = \bot \leftarrow a, b, d, \neg c, \neg g, & &\lambda(C_5) = \bot \leftarrow a, c, d, \neg b, \neg f, \neg g, \\ &\lambda(C_6) = \bot \leftarrow b, c, d, \neg a, \neg f, & &\lambda(C_7) = \bot \leftarrow a, b, c, d, \neg f, \neg g.\end{aligned}$$

According to Corollary 3, we have that $|\mathcal{AS}(\Pi_4^L)| = a_8^L$. Regarding the preprocessing for cycles. Assume that we have $L = \{\neg a, b\}$. Then, we can restrict $\Lambda_d(\Pi) = \{\lambda(C_0), \ldots, \lambda(C_7)\}$ to $U = \{\lambda(C_1), \lambda(C_6)\}$. In consequence,

$$\begin{aligned}|\mathcal{S}(\Pi[L] \cup U)| &= |\mathcal{S}(\Pi[L])| - |\mathcal{S}(\Pi[L \cup B(\lambda_1)])| - |\mathcal{S}(\Pi[L \cup B(\lambda_6)])| \\ &\quad + |\mathcal{S}(\Pi[L \cup B(\{\lambda_1, \lambda_6\})])| \\ &= 0 - 0 - 0 + 0 = 0 = |\mathcal{AS}(\Pi_4[L])|.\end{aligned}$$

△

5 Empirical evaluation

To demonstrate the capability of our approach, we implement the functionality into a tool that we call `iascar` (incremental answer set counter with anytime refinement and counting graph compressor). Our prototypical system is publicly available.[1] Below, we outline implementation details and illustrate the results of a series of practical experiments, which aim at evaluating the feasibility of our approach and its limitations. We explain the design of experiments, our expectations, and examine our expectations within a set of instances originating in an AI problem, a prototypical ASP problem, standard combinatorial puzzles, and graph problems.[2]

[1] The latest version can be found on github at `https://github.com/drwadu/iascar` .
[2] Experimental data, including a Linux binary and the source code of the evaluated version of `iascar`, is available at `https://doi.org/10.5281/zenodo.10091992` (Fichte *et al.* 2023).

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Design of Experiments. We design an empirical evaluation to study the questions:

1. Can we obtain sd-DNNFs for supported model counting by modern knowledge compilers?
2. Are these resulting sd-DNNFs feasible for our incremental answer set counting?
3. How does incremental counting on sd-DNNFs compare to translating ASP instances into CNFs and run state-of-the-art model counters?
4. Since our technique aims at improving counting multiple times and under varying assumptions, do we benefit from the potentially expensive construction of sd-DNNFs when counting multiple times?
5. What are the qualitative effects of the inclusion–exclusion-based approach to reduce the over-counting that initially occurs when only supported models are constructed but reduced gradually?

Implementation Details. Our system `iascar` is written in Rust and builds upon well-established tools, namely, `gringo` for constructing ground instances (Gebser *et al.* 2011), the Aalto ASP Tools for converting extended rules (Bomanson *et al.* 2016) and constructing Clark's completion (Gebser *et al.* 2011), and `c2d` to compile CNFs into a DNNF (Darwiche 2004, 1999). In more detail, we implement Algorithms 1 and 2, which first construct a CCG and then count based on the inclusion–exclusion technique. We assume the input program to be ground, if not we use `gringo` to construct a propositional instance (Gebser *et al.* 2011). To obtain a CCG from a propositional program, we first convert extended rules of the ground input program into normal rules using the tool `lp2normal` (Bomanson *et al.* 2016). Then, we construct a positive dependency graph from the propositional program and encode simple cycles, that is, only the first and last vertex repeat, as unsupported constraints. According to Corollary 3, we need to take all cycles into account to obtain the exact number of answer sets of an instance. Separately, we store the completion of the resulting program as a CNF using `lp2sat` (Janhunen 2006). Afterward, we compile the resulting CNF into an (sd-D)NNF by employing `c2d` (Darwiche 2004, 1999).

Model Counters for Comparison. Later, we compare our system to existing tools for counting. Natural approaches for counting are: (a) We employ answer set counters. (b) We enumerate answer sets by a recent answer set solver. (c) Alternatively, we translate the propositional input program into a propositional formula and run state-of-the-art preprocessors and model counters on the resulting formula. We require a one-to-one correspondence between the answer sets and the satisfying assignments for the translation. Unfortunately, existing answer set counters focus on extended functionality like probabilistic reasoning (Fichte *et al.* 2022c), algebraic semi-rings (Eiter *et al.* 2021), or are tailored toward approximate counting (Kabir *et al.* 2022) or certain structural restrictions of the instance (Fichte *et al.* 2017). Therefore, we omit tools listed in (a) from an evaluation. For (b), we use the answer set solver `clingo` (Gebser *et al.* 2009) to enumerate answer sets. To speed up solving, we do not output the answer sets. Since there have been recent advances on enumerating answer sets (Alviano *et al.* 2023), we also include the solver `wasp`, where we state only the number of answer sets and report only one configuration, since we observe no notable difference.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

For repeated counting with `clingo`, one could store the enumerated answer sets and implement fast data structures to test whether an element belongs to a set (Bloom 1970; Weaver *et al.* 2012) or count (Meel *et al.* 2018). To our knowledge, there is no implementation that follows this direction and we did not implement it ourselves. For (c), we turn the input program into a propositional program using `gringo`, convert extended rules (Bomanson *et al.* 2016) into normal rules (`lp2normal`), construct Clark's completion (Gebser *et al.* 2011) (`lp2sat`), and add level mappings (`lp2atomic`). Then, we apply bipartition and elimination as a preprocessing step using `b+e` (Lagniez and Marquis 2017b) and evaluate leading solvers of the model counting competition (Fichte and Hecher 2023; Fichte *et al.* 2021a) using different conceptual techniques. Therefore, we take `c2d` (Darwiche 2004), `d4` (Lagniez and Marquis 2017a), and `sharpsat-td` (Korhonen and Järvisalo 2021). Each solver counts satisfying assignments on propositional formulas given as CNF. We consider approximate counting (Chakraborty *et al.* 2014), which is interesting for projected counting or settings where we cannot expect a solution from exact model counters. Since we observe no notable performance gain in this setting, we omit it below.

Platform, Measure, and Restrictions. We evaluated our system on two platforms (a) laptop for a user-tailored evaluation on instances with more detailed interest and (b) a systematic evaluation on a larger set of benchmark instances. For (a), we ran the experiments on an 8-core intel I7-10510U CPU 1.8 GHz with 16 GB of RAM, running Manjaro Linux 21.1.1 (Kernel 5.10.59-1-MANJARO). For (b), we used a high-performance cluster consisting of 12 nodes. Each node of the cluster is equipped with two Intel Xeon E5-2680v3 CPUs, where each of these 12 physical cores runs at 2.5 GHz clock speed and has access to 64 GB shared RAM. Results are gathered on Linux RHEL 7 powered on kernel 3.10.0-1127.19.1.el7 with hyperthreading disabled. Transparent huge pages are set to system default (Fichte *et al.* 2020). We follow standard guidelines for empirical evaluations (van der Kouwe *et al.* 2018; Fichte *et al.* 2021b) and measure runtime using `perf` and enforce limits using `runsolver` (Roussel 2011). We mainly compare wall clock time. Run times larger than 900 s count as timeout and main memory (RAM) was restricted to 8 GB. We chose a small timeout due to the interest in fast counting and fast counting multiple times as outlined in the design of experiments. We ran jobs exclusively on one machine, where solvers were executed sequentially with exclusive access and at most four other runs were executed on the same node.

Instances. For our experiment, we select instances that result in varying NNF sizes, CCG sizes, and the number of simple cycles, answer sets, and supported models. We expect prototypical problems for counting multiple times to be found in probabilistic settings. However, this area is entirely unexplored for ASP. Gradually investigating the search space of an ASP instance, so-called navigation is an application for counting multiple times on the same instance under assumptions. Nevertheless, there are no standard ASP benchmark sets and ASP competitions (Gebser *et al.* 2017; Dodaro *et al.* 2019) are either tailored for modeling problems or solving decision or optimization problems. Therefore, we consider different types of instances. Set (S1) contains 242 instances that solve a problem in AI. Set (S2) consists of 936 instances of a prototypical ASP problem. Set (S3) includes a very small set of instances of combinatorial problems. The instances

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

in sets (S1) and (S2) have been used in previous works on ASP and counting (Eiter *et al.* 2021; Besin *et al.* 2021; Hecher 2022). Set (S1) encodes finding extensions of an argumentation framework (Fichte *et al.* 2022b; Dvořák *et al.* 2020; Gaggl *et al.* 2020). While there have been various iterations of the argumentation competition ICCMA, we focused on instances from 2017 (Gaggl *et al.* 2020), and encode conflict-free sets of abstract argumentation instances. These instances have a relatively high number of answer sets and are cycle-free. In contrast, the 2019 instances are easy to enumerate (Bistarelli *et al.* 2020). The 2021 instances have only a relatively small number of solutions (Mailly *et al.* 2021). The ASP encoding for conflict-free sets originates in the abstract argumentation system ASPARTIX (Dvořák *et al.* 2020). More insights on counting and abstract argumentation frameworks and their varying semantics are available in the literature (Dewoprabowo *et al.* 2022). Set (S2) consists of instances that encode a prototypical ASP domain with reachability and use of transitive closure containing cycles. While the previous set can be done by encoding ASP instances into SAT without the use of level mappings, this set provides us with a domain to distinguish the effect of cycles. Reachability on these instances is considered on quite large real-world graphs of public transport networks from all over the world, Dell *et al.* (2017). We select graphs that either incorporate no particular means of public transport or all of them. Further, we omit unsatisfiable instances thereof. Set (S3) contains the well-known n-queens problem for $n \in \{8, 10, 12\}$; a sudoku sub-grid (3x3_grid) that has to be filled uniquely with numbers from 1 to 9; the 3-coloring problem on a graph (3_coloring) and an encoding that ensures arbitrary 2-coloring for the same graph (arb_2_coloring). These instances admit no simple cycles.

Setup. Since instances from the sets (S1) and (S2) contain many instances, we evaluate these on a cluster and summarize the details in Table 1. In addition, we report on interesting instances in more detail in Table 2. There, we omit (S1) due to absence of cycles. For counting under assumptions, we select from the given instance uniform at random three atoms and set them randomly to true or false. By setting few assumptions, we ensure that only few solutions are cut. For considered solvers, we count answer sets and supported models and repeat two times counting under up to three random assumptions. For `iascar` we run varying alternation depth until we reach a fixed-point as by Lemma 4.2.

Expectations. Before we state the results, we formulate expectations from the design of experiment and our theoretical understanding.

(E1.1): When counting multiple times, `iascar` outperforms existing systems.
(E1.2): When counting once, `iascar` is notably slower due to the overhead caused by compilation and compression.
(E1.3): Compiling sd-DNNFs from formulas that encode answer sets takes much longer than when compiling supported models. Most of the time is spend on the compilation for `iascar` if the number of cycles is small.
(E2.1): Compressing the counting graph can significantly reduce its size and works fast.
(E2.2): The runtime of `iascar` depends on the number of cycles and size of the CCG due to the structural parameter of the underlying algorithm.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Table 1. *Comparing runtimes of different solvers when directly counting answer sets by enumeration (`clingo`, `wasp`), counting answer sets on a translation to SAT (`c2d`, `sharpsat-td`, `d4`), using incremental answer-set counting (`iascar`), or using incremental answer-set counting (`iascar-d2`) of depth two. `iascar*` and `iascar-d2*` refer to runs where, regardless of the timeout, a bound (anytime count) was obtained. We omit `iascar-d2` due to relevance for (S1) and (S3). (S1) consists of 242 instances, (S2) of 936 instances, and (S3) of 6 instances. # refers to the number of solved instances within the timeout of 900 s. The average time of the compilation phase for solved instances comprises both sd-DNNF[s] (average time for translating into CNF and sd-DNNF compilation) and ccg[s] (average time for counting graph compression and encoding unsupported constraints). a[s] refers to the average runtime of the counting step. #AS contains the count in* $\log_{10}$ *notation, which equals the number of answer sets for all solvers except `iascar-d2`, `iascar*` and `iascar-d2*`.*

Set	Solver	#	sd-DNNF[s]	ccg[s]	a[s]	#AS
S1	`sharpsat-td`	183	–	–	33.6	104.4
	`c2d`	182	–	–	41.5	104.9
	`iascar`	180	24.1	32.0	**0.1**	106.0
	`d4`	174	–	–	8.3	30.8
	`clingo`	96	–	–	4.4	4.3
	`wasp`	78	–	–	12.7	3.7
S2	`clingo`	397	–	–	21.2	2.2
	`d4`	352	–	–	70.1	1.6
	`iascar*`	343	5.7	33.4	524.2	12.7
	`iascar-d2*`	343	5.7	32.1	266.6	13.0
	`wasp`	341	–	–	9.3	1.5
	`sharpsat-td`	330	–	–	66.5	1.6
	`c2d`	318	–	–	105.2	1.5
	`iascar-d2`	241	3.1	2.3	46.5	6.5
	`iascar`	131	0.9	2.8	14.8	0.2
S3	`iascar`	6	30.0	29.8	**0.2**	10.8
	`d4`	6	–	–	8.8	10.8
	`sharpsat-td`	6	–	–	45.8	10.8
	`c2d`	6	–	–	15.8	10.8
	`clingo`	4	–	–	2.9	3.6
	`wasp`	3	–	–	12.5	3.0

(E2.3): If the instance has few cycles, counting works fast. Otherwise, depth restriction makes our approach utilizable.

(E3): There are instances on which simple cycles are not sufficient for counting answer sets.

Observations and Results. We summarize our results in Tables 1 and 2. We exclude (S1) from Table 2 due to absence of cycles. Experimental data and instances are publicly available (Fichte *et al.* 2023).

(O1): In Tables 1 and 2, we see that `iascar` can compute the answer sets fast if the number of cycles is small or only few cycles are present. When taking a look onto

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Table 2. *For selected interesting instances from the considered sets, we compare runtimes of* **iascar** *for compiling the input program to an NNF when directly counting answer sets (cnf), counting supported models (sup), converging to the answer set count (A) under assumptions with specified alternation depth (d) of several instances with varying numbers of simple cycles (#SC), compressing counting graphs (T), and supported models (#$\mathcal{S}$), sd-DNNF sizes (sd-DNNF size) and CCG sizes (CCG size). Depths marked with * indicate restricting alternation depths to the corresponding value.*

Set	Instance	cnf[s]	sup[s]	A[s]	T[s]	#$\mathcal{S}$	#AS	#SC	d	sd-DNNF size	CCG size
S2	nrp_autorit	6.6	0.4	0.0	0.0	1.6×10^{01}	4.0×10^{01}	5	5	166	123
S2	**nrp_hanoi**	280.2	4.1	**0.3**	0.0	1.0×10^{14}	3.2×10^{12}	**77**	***2**	4119	**3128**
S2	**nrp_berkshire**	**311.3**	**2.7**	**5.0**	0.0	1.2×10^{13}	0.0×10^{00}	**206**	***2**	**10,626**	**7914**
S2	nrp_bart	105.1	2.1	0.1	0.0	2.3×10^{07}	5.8×10^{06}	46	*2	1645	1223
S2	nrp_aircoach	253.8	3.2	1.6	0.0	8.6×10^{11}	0.0×10^{00}	130	*2	8874	6667
S2	nrp_kyoto	0.0	0.0	0.0	0.0	2.0×10^{00}	0.0×10^{00}	2	2	57	38
S3	8_queens	5.2	4.5	0.0	0.0	9.2×10^{01}	0.0×10^{00}	0	0	48,791	3490
S3	10_queens	9.7	6.9	0.0	0.0	7.2×10^{02}	1.2×10^{01}	0	0	532,645	31,172
S3	**12_queens**	95.6	46.0	0.1	**0.7**	1.4×10^{04}	7.5×10^{01}	0	0	**12,529,332**	**649,354**
S3	3x3_grid	5.7	4.5	0.0	0.1	3.6×10^{05}	7.2×10^{02}	0	0	788,711	210,893
S3	3_coloring	8.5	7.2	0.0	0.0	1.0×10^{17}	3.0×10^{16}	0	0	6677	2839
S3	arb_2_coloring	0.4	0.4	0.0	0.0	5.2×10^{33}	6.5×10^{32}	0	0	1061	446

Table 2, we see that instances such as 3_coloring or arb_2_coloring can be solved fast despite the high number of solutions. This confirms our Expectation (E1.1).

(O2): We observe in Table 1 that while the ASP solver `clingo` suffers as soon as the number of instances is high, dedicated model counters can compute the number of answer sets quite fast on the considered instances. In fact, the overall time is faster than the overall time for `iascar`, which confirms our Expectation (E1.2). When inspecting the number of cycles as well, it confirms our Expectation (E2.3).

(O3): In Table 1, we can see that `iascar` spends a notable time during the phase of constructing sd-DNNFs of a CNF if the instance has few or no cycles. Interestingly, in our experiments we have seen that constructing an sd-DNNF of a CNF can vary notably ranging from 0.1 s to 472.0 s for (S1) and ranges within a few seconds for (S2). When we encode answer sets instead of supported models into a CNF, we obtain significantly higher runtimes for compiling the CNF into sd-DNNF. In contrast, `iascar` might allow fast compilation, but can result in extremely high runtimes when applying the inclusion–exclusion principle. This only partially confirms our Expectation (E1.3). Table 2 provides a more detailed observation for selected instances. We see that on smaller instances such as 8_queens, 3x3_grid, or arb_2_coloring, we can compile and count answer sets in reasonable time. Whereas on instances such as nrp_hanoi or nrp_berkshire we observe a high runtime; in particular, there we see that sd-DNNFs can become quite large.

(O4): In Table 2 column T[s], we can see that there are instances where compressing the counting graph can significantly reduce its size. On many instances, we see a reduction by one order, for example, 10_queens by factor 17.1 and 12_queens by 19.3. Still, for 3x3_grid, we see a reduction by 3.7. This confirms Expectation (E2.1), but there we cannot necessarily expect an improvement, which is not unsurprising due to the nature of this simplification step. In fact, compressing instances with a large number of cycles, such as nrp_berkshire, is less effective than on those with a small number of cycles, such as nrp_kyoto and 12_queens.

(O5): By correlating Observation (O3) with column #SC in Table 2, we can see that instances, which can be solved fast, have no simple cycles. This pattern still holds, if we take a look on Table 1 for more instances. When considering only a few cycles as in `iascar-d2`, which considers only depth two, we can see that instances for (S2) result in significantly more solved instances, but a high over-count. This matches with our expectation (E2.2) and the knowledge on how CNFs are generated from a program as cycles are a primary source of hardness in ASP. Unsurprisingly, compiling CNFs without level mappings/loop formulas, as stated in column sup[s], works much faster. This is particularly visible for instances nrp_hanoi, nrp_berkshire, nrp_bart, or nrp_aircoach.

(O6): From columns #SC, depth, and A[s] in Table 2, we can see that the runtime on the illustrated instances depends on both parameters. A medium number of simple cycles and depth effects the runtime; similar to high number of simple cycles and small depth. Still, with a high number of simple cycles and a small depth, we can obtain the count under assumption sufficiently fast. This partially confirms our Expectation (E2.2). Interestingly, the size of the CCG itself has a much less impact than anticipated, see instance 12_queens.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

(O7): Consider Table 2. The runtime, as stated in column A[s], indicates that we can still obtain a reasonable count for instances, which ran with restricted depth, marked by *; see for example nrp_hanoi, nrp_aircoach, or nrp_berkshire.

(O8): Finally, note that in Table 2 there is one instance, namely, nrp_autorit, for which we over-counted by 3 when restricting to simple cycles, which confirms Expectation (E3). However, on all other instances, we obtained the exact count.

Summary. The evaluation indicates that our approach clearly pays off on instances containing reasonably many cycles. In particular, we see promising results when counting under assumptions, clearly benefiting from knowledge compilation. Compression of the counting graph works reasonably fast and can significantly reduce its size. Overall, the drawn experiments allowed us to confirm our expectations we stated before running the experiments. However, we see that our approach shows only benefits if the number of cycles is sufficiently small and whenever we are interested in counting multiple times. We expect that additional preprocessing pays off, if we can either exclude cases where there are no answer sets possible or where we can reduce the instance size notably, as with preprocessing of propositional formulas. Further, since knowledge compilation might consume larger parts of our overall runtime, we immediately expect better performance with the availability of improved and optimized knowledge compilers.

6 Conclusion

We establish a novel technique for counting answer sets under assumptions combining ideas from knowledge compilation and combinatorial solving. Knowledge compilation and known transformations of ASP programs into CNF formulas already provide a basic toolbox for counting answer sets. However, compilations suffer from overhead when constructing CNFs. Our approach is similar to propagation-based solving when searching for one solution. We construct compilations that allow reasoning for supported models and apply a combinatorial principle to count answer sets. Our approach gradually reduces the over-counting we obtain when considering supported models. Further, we introduce domain-specific simplification techniques for counting graphs.

We expect our technique to be useful for navigating answer sets or answering probabilistic questions on ASP programs, requiring repeated counting questions under assumptions. Thereby, we see particular potential of our quantitative technique in the study and analysis of existing solving approaches and heuristics, especially through the lense of answer set navigation, where we expect synergies. For instance, feasible repeated counting might yield useful counting-based metrics in the context of searching diverse answer sets (Böhl *et al.* 2023; Böhl and Gaggl 2022). Another interesting application could be to augment visual representations of answer sets (Dachselt *et al.* 2022; Hahn *et al.* 2022) with designated quantitative characteristics, such as relative frequencies obtained by repeated counting under assumptions.

For future work, we plan to investigate techniques to reduce the size of compilations for supported models, which can, in fact, already be a bottleneck due to the added clauses modeling the support of an atom. There, domain-specific preprocessing or an alternative compilation could be promising. Furthermore, fast identification of unsatisfiable cases by incremental SAT solving could be interesting to evaluate. From the practical side, it is

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

seems also be interesting whether we can speed up counting by GPUs (Fichte *et al.* 2021c) or database technology (Fichte *et al.* 2022e) in the ASP navigation setting. From the theoretical side, questions on the effectiveness of knowledge compilations in ASP might be interesting and similar to considerations for formulas (Darwiche and Marquis 2002). Finally, we believe that verifiable results would also be interesting when exact bounds are required, similar to techniques that have recently been developed in propositional counting (Fichte *et al.* 2022d; Beyersdorff *et al.* 2023; Bryant *et al.* 2023).

References

Alviano, M., Dodaro, C., Fiorentino, S., Previti, A. and Ricca, F. 2023. ASP and subset minimality: Enumeration, cautious reasoning and MUSes. *Artificial Intelligence*, *320*, 103931, 1–25.

Apt, K. R., Blair, H. A. and Walker, A. 1988. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*. Elsevier, 89–148.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, UK.

Besin, V., Hecher, M. and Woltran, S. 2021. Utilizing treewidth for quantitative reasoning on epistemic logic programs. *Theory and Practice of Logic Programming*, *21*, 5, 575–592.

Beyersdorff, O., Hoffmann, T. and Spachmann, L. N. 2023. Proof Complexity of Propositional Model Counting. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT'23)*, M. Mahajan and F. Slivovsky, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 271. Dagstuhl Publishing, 2:1–2:18.

Bistarelli, S., Kotthoff, L., Santini, F. and Taticchi, C. 2020. A first overview of iccma'19. In *Proceedings of the Workshop on Advances in Argumentation in Artificial Intelligence 2020 co-located with the 19th International Conference of the Italian Association for Artificial Intelligence (AIxIA'20)*, B. Fazzinga, F. Furfaro and F. Parisi, Eds. CEUR Workshop Proceedings, vol. 2777. CEUR-WS.org, 90–102.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13(7)*, 7, 422–426.

Bogaerts, B. and den Broeck, G. V. 2015. Knowledge compilation of logic programs using approximation fixpoint theory. *Theory and Practice of Logic Programming*, *15*, 4-5, 464–480.

Böhl, E. and Gaggl, S. A. 2022. Tunas - fishing for diverse answer sets: A multi-shot trade up strategy. In *Proceedings of the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22)*, G. Gottlob, D. Inclezan and M. Maratea, Eds. Lecture Notes in Computer Science, vol. 13416. Springer, 89–102.

Böhl, E., Gaggl, S. A. and Rusovac, D. 2023. Representative answer sets: Collecting something of everything. In *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI'23)*, K. Gal, A. Nowé, G. J. Nalepa, R. Fairstein and R. Radulescu, Eds. FAIA, vol. 372. IOS Press, 271–278.

Bomanson, J., Gebser, M. and Janhunen, T. 2016. Rewriting optimization statements in answer-set programs. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP'16)*, M. Carro, A. King, N. Saeedloei and M. D. Vos, Eds. OpenAccess Series in Informatics (OASIcs), vol. 52, Dagstuhl, Germany. Dagstuhl Publishing, 5:1–5:15.

Bondy, J. A. and Murty, U. S. R. 2008. *Graph Theory*. Graduate Texts in Mathematics. Springer.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM*, *54*, 12, 92–103.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Bryant, R. E., Nawrocki, W., Avigad, J. and Heule, M. J. H. 2023. Certified knowledge compilation with application to verified model counting. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT'23)*, M. Mahajan and F. Slivovsky, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 271. Dagstuhl Publishing, 6:1–6:20.

Chakraborty, S., Fremont, D. J., Meel, K. S., Seshia, S. A. and Vardi, M. Y. 2014. Distribution-aware sampling and weighted model counting for SAT. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, C. E. Brodley and P. Stone, Eds., Québec City, QC, Canada. The AAAI Press, 1722–1730.

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*. Springer, 293–322.

Dachselt, R., Gaggl, S. A., Krötzsch, M., Méndez, J., Rusovac, D. and Yang, M. 2022. NEXAS: A visual tool for navigating and exploring argumentationsolution spaces. In *Proceedings of the 9th International Conference on Computational Models of Argument (COMMA'22)*, F. Toni, Ed. FAIA, vol. 220146. IOS Press, 116–127.

Darwiche, A. 1999. Compiling knowledge into decomposable negation normal form. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence, (IJCAI'99)*, T. Dean, Ed. Morgan Kaufmann, 284–289.

Darwiche, A. 2001. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, *11*, 1-2, 11–34.

Darwiche, A. 2004. New advances in compiling CNF to decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, R. López De Mántaras and L. Saitta, Eds., Valencia, Spain. IOS Press, 318–322.

Darwiche, A. and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research*, *17*, 229–264.

Dell, H., Komusiewicz, C., Talmon, N. and Weller, M. 2017. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *Proceedings of the 12th International Symposium on Parameterized and Exact Computation, IPEC'17*, D. Lokshtanov and N. Nishimura, Eds. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl Publishing, 30:1—30:13.

Dewoprabowo, R., Fichte, J. K., Gorczyca, P. J. and Hecher, M. 2022. A practical account into counting dung's extensions by dynamic programming. In *Proceedings of the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22)*, G. Gottlob, D. Inclezan and M. Maratea, Eds. Springer, 387–400.

Dimopoulos, Y., Nebel, B. and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning (ECP'97)*, S. Steel and R. Alami, Eds. Springer, 169–181.

Dodaro, C., Redl, C. and Schüller, P. 2019. The answer set programming challenge. https://sites.google.com/view/aspcomp2019/.

Dvořák, W., Gaggl, S. A., Rapberger, A., Wallner, J. P. and Woltran, S. 2020. The ASPARTIX system suite. In *Proceedings of the 8th International Conference on Computational Models of Argument (COMMA'20)*, H. Prakken, S. Bistarelli, F. Santini and C. Taticchi, Eds. FAIA, vol. 326. IOS Press, 461–462.

Eiter, T., Hecher, M. and Kiesel, R. 2021. Treewidth-aware cycle breaking for algebraic answer set counting. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR'21)*, M. Bienvenu, G. Lakemeyer and E. Erdem, Eds. IJCAI Organization, 269–279.

Fages, F. 1994. Consistency of clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, *1*, 1, 51–60.

Ferraris, P., Lee, J. and Lifschitz, V. 2006. A generalization of the lin-zhao theorem. *Annals of Mathematics and Artificial Intelligence*, *47*, 79–101.

Fichte, J. K., Gaggl, S. A., Hecher, M. and Rusovac, D. 2022a. IASCAR: Incremental answer set counting by anytime refinement. In *Proceedings of the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22)*, G. Gottlob, D. Inclezan and M. Maratea, Eds. Lecture Notes in Computer Science, vol. 13416. Springer, 217–230.

Fichte, J. K., Gaggl, S. A., Hecher, M. and Rusovac, D. 2023. IASCAR: Incremental answer set counting by anytime refinement (experiments).

Fichte, J. K., Gaggl, S. A. and Rusovac, D. 2022b. Rushing and strolling among answer sets – navigation made easy. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI'22)*. AAAI Press, 5651–5659.

Fichte, J. K. and Hecher, M. 2023. The model counting competitions 2021–2023. https://mccompetition.org/past_iterations.

Fichte, J. K., Hecher, M. and Hamiti, F. 2021a. The model counting competition. *ACM Journal of Experimental Algorithmics*, *26*a, 1–26.

Fichte, J. K., Hecher, M., McCreesh, C. and Shahab, A. 2021b. Complications for computational experiments from modern processors. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming, (CP'21)*, L. D. Michel, Ed. Leibniz International Proceedings in Informatics (LIPIcs), vol. 210. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:21.

Fichte, J. K., Hecher, M., Morak, M. and Woltran, S. 2017. Answer set solving with bounded treewidth revisited. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, M. Balduccini and T. Janhunen, Eds. Lecture Notes in Computer Science, vol. 10377. Springer, 132–145.

Fichte, J. K., Hecher, M. and Nadeem, M. A. 2022c. Plausibility reasoning via projected answer set counting - a hybrid approach. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence, (IJCAI'22)*, L. D. Raedt, Ed. International Joint Conferences on Artificial Intelligence Organization, 2620–2626.

Fichte, J. K., Hecher, M. and Roland, V. 2021c. Parallel model counting with CUDA: Algorithm engineering for efficient hardware utilization. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP'21)*, L. D. Michel, Ed. Leibniz International Proceedings in Informatics (LIPIcs), vol. 210. Dagstuhl Publishing, 24:1–24:20.

Fichte, J. K., Hecher, M. and Roland, V. 2022d. Proofs for Propositional Model Counting. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT'22)*, K. S. Meel and O. Strichman, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, Dagstuhl, Germany. Dagstuhl Publishing, 30:1–30:24.

Fichte, J. K., Hecher, M., Thier, P. and Woltran, S. 2022e. Exploiting database management systems and treewidth for counting. *Theory and Practice of Logic Programming*, *22*e, 1, 128–157.

Fichte, J. K., Manthey, N., Schidler, A. and Stecklina, J. 2020. Towards faster reasoners by using transparent huge pages. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, H. Simonis, Ed. Lecture Notes in Computer Science. Springer, 304–322.

Fierens, D., den Broeck, G. V., Renkens, J., Shterionov, D. S., Gutmann, B., Thon, I., Janssens, G. and Raedt, L. D. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, *15*, 3, 358–401.

Gaggl, S. A., Linsbichler, T., Maratea, M. and Woltran, S. 2020. Design and results of the second international competition on computational models of argumentation. *Artificial Intelligence*, *279*, 103193.

Gebser, M., Kaminski, R., König, A. and Schaub, T. 2011. Advances in gringo series 3. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, J. P. Delgrande and W. Faber, Eds. Springer, 345–351.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Gebser, M., Kaufmann, B. and Schaub, T. 2009. The conflict-driven answer set solver clasp: Progress report. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, E. Erdem, F. Lin and T. Schaub, Eds. Springer, 509–514.

Gebser, M., Kaufmann, B. and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, *187-188*, 52–89.

Gebser, M., Maratea, M. and Ricca, F. 2017. The design of the seventh answer set programming competition. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, M. Balduccini and T. Janhunen, Eds. Springer, 3–9.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP'88)*, R. A. Kowalski and K. A. Bowen, Eds., vol. 2. MIT Press, 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, *9*, 3/4, 365–386.

Hahn, S., Sabuncu, O., Schaub, T. and Stolzmann, T. 2022. Clingraph: ASP-based visualization. In *Proceedings of the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22)*, G. Gottlob, D. Inclezan and M. Maratea, Eds. Lecture Notes in Computer Science, vol. 13416. Springer, 401–414.

Hecher, M. 2022. Treewidth-aware reductions of normal ASP to SAT – Is normal ASP harder than SAT after all? *Artificial Intelligence*, *304*, 103651.

Janhunen, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, *16*, 1-2, 35–86.

Janhunen, T. and Niemelä, I. 2011. Compact translations of non-disjunctive answer set programs to propositional clauses. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning – Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, M. Balduccini and T. Son, Eds. Lecture Notes in Artificial Intelligence, vol. 6565. Springer, 111–130.

Kabir, M., Everardo, F. O., Shukla, A. K., Hecher, M., Fichte, J. K. and Meel, K. S. 2022. ApproxASP – a scalable approximate answer set counter. *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI'22)*, 5755–5764.

Kleine Büning, H. and Lettmann, T. 1999. *Propositional Logic – Deduction and Algorithms*. Cambridge Tracts in Theoretical Computer Science, vol. 48. Cambridge University Press.

Korhonen, T. and Järvisalo, M. 2021. Integrating tree decompositions into decision heuristics of propositional model counters. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP'21)*, L. D. Michel, Ed. Leibniz International Proceedings in Informatics (LIPIcs), vol. 210, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 8:1–8:11.

Kuiter, E., Krieter, S., Sundermann, C., Thüm, T. and Saake, G. 2023. Tseitin or not tseitin? the impact of cnf transformations on feature-model analyses. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*, Rochester, MI, USA. ACM.

Lagniez, J., Lonca, E. and Marquis, P. 2016. Improving model counting by leveraging definability. In *Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, S. Kambhampati, Ed., New York City, NY, USA. The AAAI Press, 751–757.

Lagniez, J. and Marquis, P. 2014. Preprocessing for propositional model counting. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, C. E. Brodley and P. Stone, Eds., Québec City, QC, Canada. The AAAI Press, 2688–2694.

Lagniez, J. and Marquis, P. 2017a. An improved decision-DDNF compiler. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, C. Sierra, Ed., Melbourne, VIC, Australia. The AAAI Press, 667–673.

Lagniez, J. and Marquis, P. 2017b. On preprocessing techniques and their impact on propositional model counting. *Journal of Automated Reasoning*, *58*b, 4, 413–481.

Lautemann, C. 1983. Bpp and the polynomial hierarchy. *Information Processing Letters*, *17*, 4, 215–217.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, L. P. Kaelbling and A. Saffiotti, Eds., vol. 19, Edinburgh, Scotland, UK. Professional Book Center, 503–508.

Lee, J. and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs. In *Proceedings of the 19th International Conference on Logic Programming (LP'03)*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 2916, Mumbai, India. Springer, 451–465.

Lee, J. and Wang, Y. 2015. A probabilistic extension of the stable model semantics. In *2015 AAAI Spring Symposia, Stanford University*. AAAI Press.

Lifschitz, V. 1999. Action languages, answer sets, and planning. In *The Logic Programming Paradigm*. Springer, 357–373.

Lifschitz, V. and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, *7*, 2, 261–268.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, *157*, 1-2, 115–137.

Mailly, J., Lonca, E., Lagniez, J. and Rossit, J. 2021. The fourth international competition on computational models of argumentation (ICCMA'21). http://argumentationcompetition.org/2021/index.html.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczyński and D. S. Warren, Eds. Artificial Intelligence. Springer, 375–398.

Marek, W. and Subrahmanian, V. 1992. The relationship between stable, supported, default and autoepistemic semantics for general logic programs. *Theoretical Computer Science*, *103*, 2, 365–386.

Masina, G., Spallitta, G. and Sebastiani, R. 2023. On CNF Conversion for Disjoint SAT Enumeration. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT'23)*, M. Mahajan and F. Slivovsky, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 271, Alghero, Italy. Dagstuhl Publishing, 15:1–15:16.

Meel, K. S., Shrotri, A. A. and Vardi, M. Y. 2018. On hashing-based approaches to approximate DNF-counting. In *Proceedings of the 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'17)*, S. Lokam and R. Ramanujam, Eds., Leibniz International Proceedings in Informatics (LIPIcs), vol. 93, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 41:1–41:14.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, *25*, 3-4, 241–273.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R. and Barry, M. 2001. An a-prolog decision support system for the space shuttle. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, I. V. Ramakrishnan, Ed., Las Vegas, Nevada, USA. Springer, 169–183.

Pontelli, E., Son, T., Baral, C. and Gelfond, G. 2012. Answer set programming and planning with knowledge and world-altering actions in multiple agent domains. In *Correct Reasoning – Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, E. Erdem, J. Lee, Y. Lierler and D. Pearce, Eds. Lecture Notes in Computer Science, vol. 7265. Springer, 509–526.

Robinson, J. A. and Voronkov, A., Eds. 2001. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press.

Roussel, O. 2011. Controlling a solver execution with the runsolver tool. *Journal on Satisfiability, Boolean Modeling and Computation*, *7*, 139–144.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press

Sang, T., Beame, P. and Kautz, H. 2005. Performing Bayesian inference by weighted model counting. In *AAAI'05*, Pittsburgh, Pennsylvania, USA. The AAAI Press.

Sipser, M. 1983. A complexity theoretic approach to randomness. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC'83)*, Boston, Massachusetts, USA, 330–335.

Spallitta, G., Sebastiani, R. and Biere, A. 2023. Enumerating disjoint partial models without blocking clauses. CoRR, abs/2306.00461.

Stockmeyer, L. 1983. The complexity of approximate counting. In *Proceedings of the 15h Annual ACM Symposium on Theory of Computing (STOC'83)*, New York, NY, USA. Association for Computing Machinery, 118–126.

Stockmeyer, L. J. 1976. The polynomial-time hierarchy. *Theoretical Computer Science*, *3*, 1, 1–22.

Toda, S. 1991. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, *20*, 5, 865–877.

Truszczyński, M. 2011. Trichotomy and dichotomy results on the complexity of reasoning with disjunctive logic programs. *Theory and Practice of Logic Programming*, *11*, 881–904.

Tseytin, G. S. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483.

van der Kouwe, E., Andriesse, D., Bos, H., Giuffrida, C. and Heiser, G. 2018. Benchmarking crimes: An emerging threat in systems security. CoRR, abs/1801.02381, 1–17.

Wang, Y. and Lee, J. 2015. Handling uncertainty in answer set programming. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*, B. Bonet and S. Koenig, Eds., Austin, TX, USA. The AAAI Press, 4218–4219.

Weaver, S. A., Ray, K. J., Marek, V. W., Mayer, A. J. and Walker, A. K. 2012. Satisfiability-based set membership filters. *Journal on Satisfiability, Boolean Modeling and Computation*, *8*, 3-4, 129–148.

https://doi.org/10.1017/S1471068424000036 Published online by Cambridge University Press