IASCAR: Incremental Answer Set Counting by Anytime Refinement

Johannes Klaus Fichte[1], Sarah Alice Gaggl[2], Markus Hecher[1], and Dominik Rusovac[2(✉)]

[1] Institute of Logic and Computation, TU Wien, Vienna, Austria
{johannes.fichte,markus.hecher}@tuwien.ac.at
[2] Logic Programming and Argumentation Group, TU Dresden, Dresden, Germany
{sarah.gaggl,dominik.rusovac}@tu-dresden.de

Abstract. Answer set programming (ASP) is a popular declarative programming paradigm with various applications. Programs can easily have so many answer sets that they cannot be enumerated in practice, but counting still allows to quantify 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*. But navigating through parts of the solution space requires counting many times, which is expensive in theory. There, *knowledge compilation* compiles instances into representations on which counting works in polynomial time. However, these techniques exist only for CNF formulas and compiling ASP programs into CNF formulas can introduce an exponential overhead. In this paper, we introduce a technique to iteratively count answer sets under assumptions on knowledge compilations of CNFs that encode supported models. Our anytime technique uses the principle of inclusion-exclusion to systematically improve bounds by over- and undercounting. 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

1 Introduction

Answer set programming (ASP) [11] is a widely used declarative problem modeling and solving paradigm with many applications in knowledge representation, artificial intelligence, planning, and many more. It is widely used to solve difficult search problems while allowing compact modeling [7]. In ASP, a problem is represented as a set of rules, called *logic program*, over atoms. Models of a program under the stable semantics 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., [6,13] or encoding Bayesian networks and their inference [12]. Interestingly, counting also facilitates more fine-grained reasoning modes between brave

G. Gottlob et al. (Eds.): LPNMR 2022, LNAI 13416, pp. 217–230, 2022.
https://doi.org/10.1007/978-3-031-15707-3_17

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 [5]. However, already for ground normal programs, answer set counting is #·P-complete, making it harder than decision problems. Recall that brave reasoning is just NP-complete, but by Toda's Theorem we know that $\mathrm{PH} \subseteq \mathrm{P}^{\#\cdot\mathrm{P}}$ where $\bigcup_{k\in\mathbb{N}} \Delta_k^P = \mathrm{PH}$ and $\mathrm{NP} \subseteq \Delta_2^P = \mathrm{P}^{\mathrm{NP}}$. Approximate counting is in fact easier, i.e., approx-$\#\cdot\mathrm{P} \subseteq \mathrm{BPP}^{\mathrm{NP}} \subseteq \Sigma_3^P$, and approximate answer set counters have very recently been suggested [8]. 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 [3].

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 and directly apply techniques from propositional formulas, it is widely known that one can easily run into an exponential blowup [10] or introduce level mappings 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 [7].

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 efficiently compute the number of supported models. The resulting counting graph can be quite large, but can be evaluated in parallel. Counting supported models provides us only with an upper bound on the number of answer sets. We suggest a combinatorial technique to systematically improve bounds by over- and undercounting while incorporating the external support, whose absence can be seen as 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 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. We can quickly (re-)count after every search space navigation step.

Related Works. Previous work [1] 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 CNFs directly. Note that common reasoning problems on answer set programs without negation can be solved in polynomial time. Model counting can significantly benefit from preprocessing techniques, which eliminate variables. Widely used propositional knowledge compilers are c2d [2] and d4 [9].

2 Preliminaries

We assume familiarity with propositional satisfiability, graph theory, propositional ASP [7]. 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 $\neg a$, we assume $\neg\neg a = a$, 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 $\mathcal{M}(\varphi)$.

Answer Set Programming (ASP). In the context of ASP, we usually say atom instead of variable. A *(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* of Π is the formula $\mathrm{comp}(\Pi) \coloneqq \{a \leftrightarrow \bigvee_{r \in \Pi, H(r) = a} BF(r) \vee \bot \mid a \in \mathit{at}(\Pi)\}$ where $BF(r) \coloneqq \bigwedge_{b \in B^+(r)} b \wedge \bigwedge_{c \in B^-(r)} \neg c \wedge \top$. An interpretation I is a *supported model*

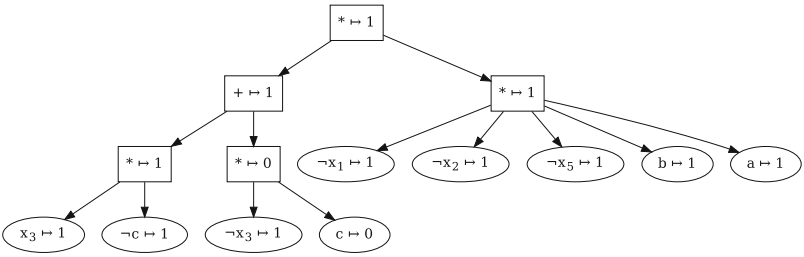

Fig. 1. Counting graph $\mathcal{G}(\varphi \wedge \neg c)$ labeled with literals, operations and val.

of Π, if it is a model of the formula comp(Π). Let $\mathcal{S}(\Pi)$ be the set of all supported models of Π. It holds that $\mathcal{AS}(\Pi) \subseteq \mathcal{S}(\Pi)$, but not vice-versa. If Π is tight, then $\mathcal{AS}(\Pi) = \mathcal{S}(\Pi)$. 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.

Assumptions. We define $\neg L \coloneqq \{\neg a \mid a \in L\}$ for a set L of literals. 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 (cont'd). Since $\mathcal{AS}(\Pi_1) = \{\{a, b\}\}$, we see that if $L \subseteq \{a, b, \neg c\}$, we obtain $\mathcal{AS}(\Pi_1) = \mathcal{AS}(\Pi_1^L)$, and otherwise $\mathcal{AS}(\Pi_1^L) = \emptyset$.

3 Counting Supported Models

In our applications mentioned in the introduction, we are interested in counting multiple times under assumptions. Therefore, we extend known techniques from knowledge compilation [3]. 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 $\Phi_{\mathrm{comp}(\Pi)}$ in a normal form, so-called sd-DNNF by existing knowledge compilers. Then, (iii) on the sd-DNNF $\Phi_{\mathrm{comp}(\Pi)}$ counting can be done in polynomial time in the size of $\Phi_{\mathrm{comp}(\Pi)}$.We can even count under a set L of propositional assumptions by a technique used 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 Sect. 4, (iv) we present a technique to incrementally reduce the overcount. First, 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.

Knowledge Compilation [3] and Counting on Formulas in sd-DNNF. Let φ be a formula. φ is in *NNF (negation normal form)* if negations ($\neg$) occur only directly in front of variables and the only other operators are conjunction ($\wedge$) and disjunction ($\vee$). 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 $\text{vars}(\psi_i) \cap \text{vars}(\psi_j) = \emptyset$. 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. φ is in *sd-DNNF*, if all disjunctions in ψ are smooth, meaning for $\psi = \psi_1 \vee \psi_2$ we have $\text{vars}(\psi_1) = \text{vars}(\psi_2)$. Determinism and smoothness permit traversal operations on sd-DNNFs to count models of φ in linear time in $|\varphi|$. 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 $\text{val}(N) := 1$, if N consists of a literal; labeled by $\text{val}(N) := \Sigma_i \, \text{val}(N_i)$, if N is an $\vee$-node with children N_i; labeled by $\text{val}(N) := \Pi_i \, \text{val}(N_i)$, if N is an $\wedge$-node. By $\text{val}(\mathcal{G}(\varphi))$ we refer to $\text{val}(N)$ 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 $\text{val}(\mathcal{G}(\varphi))$ equals the model count of φ. For a set L of literals, counting of $\varphi^L := \varphi \wedge \bigwedge_{\ell \in L} \ell$ can be carried out by *conditioning* of φ on L [2]. Therefore, the function val on the counting graph is modified by setting $\text{val}(N) = 0$, 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 Φ_{Π^L} an equivalent sd-DNNF of $\text{comp}(\Pi^L)$ 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 $(\mathcal{G}_\Pi)^L$.

Example 3. Consider 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 Fig. 1 that its DAG has 14 nodes, 7 variables and 13 edges, so that $|\varphi_1| = 13$. By conditioning, each variable in L will be removed from $\mathcal{G}(\varphi_1)$ and $\varphi_1 \wedge \neg c = ((x_3 \wedge \neg\bot) \vee (\neg x_3 \wedge \bot)) \wedge (\neg x_1 \wedge \neg x_2 \wedge \neg x_5 \wedge a \wedge b)$. From Fig. 1, we observe that the model count $\text{val}(\mathcal{G}(\varphi \wedge \neg c))$ of $\varphi \wedge \neg c$ is 1.

Counting Supported Models. Using the techniques as described above, we can compile the formula $\text{comp}(\Pi)$ 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 $\text{comp}(\Pi_1)$ in CNF, we obtain 10 clauses with 4 new auxiliary variables x_1, x_2, x_3, x_5. We can compile it into an sd-DNNF Φ_{Π_1} which is logically equivalent to $\text{comp}(\Pi_1)$. For illustration purposes, we chose 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 $\text{val}(\mathcal{G}_{\Pi_1})$.

3.1 Counting Supported Models Under Assumptions

Since assumptions of formulas and programs slightly differ, it is not immediately clear that we can use conditioning to obtain the number of supported models of a program under given assumptions. However, supported models of Π under assumptions L coincide with models of Φ_{Π^L}.

Observation 1. *$\mathcal{M}(\Phi_{\Pi^L}) = \mathcal{S}(\Pi^L)$ for program Π and assumptions L.*

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

Lemma 1 ($\star$[1]). *$\mathcal{M}((\Phi_\Pi)^L) = \mathcal{S}(\Pi^L)$ for program Π and assumptions L.*

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.

Corollary 1. *$\mathrm{val}((\mathcal{G}_\Pi)^L) = |\mathcal{M}((\Phi_\Pi)^L)| = |\mathcal{S}(\Pi^L)|$ for program Π and assumptions L. If Π is tight, also $\mathrm{val}((\mathcal{G}_\Pi)^L) = |\mathcal{AS}(\Pi^L)|$ holds. Furthermore, counting can be done in time linear in $|\Phi_\Pi|$.*

Example 5 (cont'd). Π_1 has two supported models $\{a, b\}$ and $\{a, b, c\}$. Without setting $\mathrm{val}(c)$ to 0 in Fig. 1, we would obtain 2, which corresponds to these two models. By assumption $\neg c$, we set $\mathrm{val}(c)$ to 0, which results in 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 the well-known Tseitin transformation. It is well-known that there is a one-to-one correspondence, however, auxiliary variables are introduced. 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, which we describe by Algorithm 1. Algorithm 1 takes as input an sd-DNNF Φ_Π, and literals $\mathcal{L}(\Pi)$; and returns the compressed counting graph $\tau(\mathcal{G}_\Pi)$. In Line 2, 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.

[1] Statements marked by "$\star$" are proven in appendix https://tinyurl.com/iascar-p.

Algorithm 1. Counting Graph Compression

In: Program Π, sd-DNNF Φ_Π; **Out**: $\tau(\mathcal{G}_\Pi)$

```
1: initialize array t and traverse nodes N ∈ Φ_Π bottom-up such that
2:   if N contains a literal ℓ ∈ L(Π) then label N with val(N)
3:   else if N contains a literal ℓ ∉ L(Π) 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 ← C and mark N as ignored
7:     else v ← 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
```

In Line 6, the node needs to be absorbed, as it has only one child meaning that 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.

Lemma 2 ($\star$). *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,* $\mathrm{val}(\tau(\mathcal{G}_\Pi)) = \mathrm{val}(\mathcal{G}_\Pi)$ *and $\tau(\mathcal{G}_\Pi)$ can be constructed in time $\mathcal{O}(2 \cdot |\Phi_\Pi|)$.*

Corollary 2. *If Π is tight, then* $\mathrm{val}(\tau(\mathcal{G}_\Pi)) = |\mathcal{AS}(\Pi)|$.

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 cyclic atoms $C \in cycles(\Pi)$ 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[2] *of $C \in cycles(\Pi)$, whenever $H(r) \subseteq C$ and $a \notin C$. 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.

[2] Note that external supports are sets of literals. However, we can simulate such a set by introducing an auxiliary atom; hence one atom, as in this definition, is sufficient [7].

Example 6. Let $\Pi_2 = \{a \leftarrow b; b \leftarrow a; a \leftarrow c; c \leftarrow \neg d; d \leftarrow \neg c\}$. 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 $\mathcal{AS}(\Pi_2) = \{\{a, b, c\}, \{d\}\}$.

Example 7. Let $a \leftarrow b$, $b \leftarrow a$, and $b \leftarrow c, \neg d$ be rules. Then the external support of cyclic atoms $\{a, b\}$ is $\{c, \neg d\}$. 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 $ES(\{a, b\}) = \{b_r\}$, see Footnote (see Footnote 2).

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, i.e., 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 $|X_1 \cup X_2 \cup X_3| = |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_2| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3|$. This principle can be used to count answer sets via supported model counting.

We define the *unsupported constraint* $\lambda(C)$ for a set $C = \{c_0, \ldots, c_n\} \in \mathit{cycles}(\Pi)$ of cyclic atoms and its resp. external supports $ES(C) = \{s_0, \ldots, s_m\}$ by $\lambda(C) \coloneqq \bot \leftarrow c_0, \ldots, c_n, \neg s_0, \ldots, \neg s_m$. The unsupported constraints as defined here contain the whole set C, which is slightly weaker than constraints (nogoods) defined in related work [7], but sufficient for characterizing answer sets.

Lemma 3 ($\star$). *For any given program Π where $C_i \in \mathit{cycles}(\Pi)$ and $1 \leq i \leq n$, we have that $\mathcal{AS}(\Pi) = \mathcal{S}(\Pi \cup \{\lambda(C_1), \ldots, \lambda(C_n)\})$.*

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 $n \coloneqq |\mathit{cycles}(\Pi)|$. Now, we define the approach of $|\mathcal{AS}(\Pi^L)|$ by a_d^L, using the combinatorial principle of inclusion-exclusion as follows:

$$a_d^L \coloneqq \sum_{i=0}^{d} (-1)^i \sum_{\Gamma \in \Lambda_i(\Pi)} |\mathcal{S}(\Pi^L \cup \Gamma)| = |\mathcal{S}(\Pi^L)| - \sum_{\Gamma \in \Lambda_1(\Pi)} |\mathcal{S}(\Pi^L \cup \Gamma)| + \sum_{\Gamma \in \Lambda_2(\Pi)} |\mathcal{S}(\Pi^L \cup \Gamma)| - \cdots + (-1)^d \sum_{\Gamma \in \Lambda_d(\Pi)} |\mathcal{S}(\Pi^L \cup \Gamma)|$$

By subtracting $|\mathcal{S}(\Pi^L \cup \Gamma)|$ 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 $|\mathcal{S}(\Pi^L \cup \Gamma)|$

Algorithm 2. Incremental Counting by Anytime Refinement

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

1: `count` $\leftarrow \mathrm{val}(\tau(\mathcal{G}_\Pi)^L)$ 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: $c' \leftarrow \mathrm{val}(\tau(\mathcal{G}_\Pi)^{L \cup L'})$ where L' is the set of literals appearing in $\Gamma_j \in \Lambda_i(\Pi)$
7: `count` $\leftarrow$ `count` $- c'$ **if** i is odd **otherwise** `count` $\leftarrow$ `count` $+ c'$
8: **return** `count`

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, $cycles(\Pi) = \{C_1, \ldots, C_n\}$, and further $U \coloneqq \{\lambda(C_1), \ldots, \lambda(C_n)\}$ be the set of all unsupported constraints of Π. Then, $|\mathcal{S}(\Pi^L \cup U)| = \sum_{i=0}^{n} (-1)^i \sum_{\Gamma \in \Lambda_i(\Pi)} |\mathcal{S}(\Pi^L \cup \Gamma)|$ for assumptions L.*

Finally, one can count answer sets correctly.

Corollary 3 (⋆). *Let $n = |cycles(\Pi)|$. Then, $a_n^L = |\mathcal{AS}(\Pi^L)|$ for program Π and assumptions 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 (⋆). *Let Π be a program and L be assumptions. If $a_i^L = a_{i+1}^L$ for some integer $i \geq 0$, then $a_i^L = |\mathcal{AS}(\Pi^L)|$.*

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 approaching the answer set count 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 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 as a termination criterion in Line 4.

Table 1. Runtimes of compiling input program to an NNF when directly counting answer sets (sat), counting supported models (comp), compressing counting graphs (T) and approaching the answer set count (A) under assumptions with specified alternation depth (d) of several instances with varying numbers of simple cycles (#SC) and supported models ($\#\mathcal{S}$), sd-DNNF sizes (NNF size) and CCG sizes (CCG size). Depths marked with * indicates restricting alternation depths. `IASCAR#` corresponds to the approximation of the number of answer sets.

Instance	sat[s]	comp[s]	NNF size	T[s]	CCG size	$\#\mathcal{S}$	#SC	d	IASCAR#	A[s]
8_queens	5.2	4.5	48,791	0.0	3,490	$9.200 \cdot 10^{1}$	0	0	$0.000 \cdot 10^{0}$	0.0
10_queens	9.7	6.9	532,645	0.0	31,172	$7.240 \cdot 10^{2}$	0	0	$1.200 \cdot 10^{1}$	0.0
12_queens	95.6	46.0	**12,529,332**	**0.7**	**649,354**	$1.420 \cdot 10^{4}$	0	0	$7.500 \cdot 10^{1}$	0.1
3x3_grid	5.7	4.5	788,711	0.1	210,893	$3.629 \cdot 10^{5}$	0	0	$7.200 \cdot 10^{2}$	0.0
AF_stable	3.0	2.9	11,141	0.0	3,284	$7.696 \cdot 10^{3}$	0	0	$3.080 \cdot 10^{3}$	0.0
3_coloring	8.5	7.2	6,677	0.0	2,839	$1.026 \cdot 10^{17}$	0	0	$3.028 \cdot 10^{16}$	0.0
arb_2_coloring	0.4	0.4	1,061	0.0	446	$5.193 \cdot 10^{33}$	0	0	$6.490 \cdot 10^{32}$	0.0
simple	1.3	0.1	90	0.0	59	$1.400 \cdot 10^{1}$	3	3	$0.000 \cdot 10^{0}$	0.0
nrp_accenture	6.0	0.3	119	0.0	84	$6.000 \cdot 10^{0}$	5	5	$0.000 \cdot 10^{0}$	0.0
nrp_autorit	6.6	0.4	166	0.0	123	$1.600 \cdot 10^{1}$	5	5	$4.000 \cdot 10^{1}$	0.0
nrp_california	12.8	0.5	201	0.0	**133**	$8.000 \cdot 10^{0}$	**15**	**15**	$0.000 \cdot 10^{0}$	**0.5**
nrp_hanoi	280.2	4.1	4,119	0.0	**3,128**	$1.017 \cdot 10^{14}$	**77**	***2**	$3.197 \cdot 10^{12}$	**0.3**
nrp_berkshire	**311.3**	**2.7**	**10,626**	0.0	**7,914**	$1.162 \cdot 10^{13}$	**206**	***2**	$0.000 \cdot 10^{0}$	**5.0**
nrp_benton	20.4	0.7	642	0.0	446	$5.200 \cdot 10^{1}$	38	*2	$0.000 \cdot 10^{0}$	0.0
nrp_bart	105.1	2.1	1,645	0.0	1,223	$2.295 \cdot 10^{7}$	46	*2	$5.767 \cdot 10^{6}$	0.1
nrp_aircoach	253.8	3.2	8,874	0.0	6,667	$8.563 \cdot 10^{11}$	130	*2	$0.000 \cdot 10^{0}$	1.6
nrp_a1210993	64.7	1.6	1,280	0.0	954	$3.642 \cdot 10^{5}$	29	*2	$0.000 \cdot 10^{0}$	0.0
nrp_kyoto	0.0	0.0	57	0.0	38	$2.000 \cdot 10^{0}$	2	2	$0.000 \cdot 10^{0}$	0.0

Example 8. Let $\Pi_3 = \Pi_2 \cup \{b \leftarrow g; f \leftarrow g; e \leftarrow f; f \leftarrow e\}$, which has 2 cycles $C_0 = \{a, b\}$ and $C_1 = \{e, f\}$. Their corresponding external supports are $ES(C_0) = \{c, g\}$ and $ES(C_1) = \{g\}$. Program Π_3 has 6 supported models $\{\{d\}, \{d, e, f\}, \{a, b, d\}, \{a, b, c\}, \{a, b, c, e, f\}, \{a, b, d, e, f\}\}$ of which $\{d\}$ and $\{a, b, c\}$ are answer sets. Suppose we want to determine $a_1^{\{d\}}$, then: $a_1^{\{d\}} = |\mathcal{S}(\Pi^{\{d\}})| - |\mathcal{S}(\Pi^{\{d\}} \cup \{\bot \leftarrow a, b, \neg c, \neg g\})| - |\mathcal{S}(\Pi^{\{d\}} \cup \{\bot \leftarrow e, f, \neg g\})| = 4 - 2 - 2 = 0$. We see that restricting the alternation depth to 1, leads to undercounting. However, not restricting the depth leads to the exact count as: $a_2^{\{d\}} = a_1^{\{d\}} + |\mathcal{S}(\Pi^{\{d\}} \cup \{\bot \leftarrow a, b, \neg c, \neg g; \bot \leftarrow e, f, \neg g\})| = 0 + 1 = 1 = |\mathcal{AS}(\Pi_3^{\{d\}})|$.

5 Preliminary Empirical Evaluation

To demonstrate the capability of our approach, we implemented a prototypical system, called `IASCAR`. The system binary is publicly available for download[3]. Our system counts on CCGs constructed from sd-DNNFs. Therefore, we implement Algorithms 1 and 2, which first construct a CCG and then count based on

[3] See https://tinyurl.com/iascar-b for a Linux binary, instances, and raw data. `gringo` cuts off trivial supported models when grounding, not affecting us here.

the inclusion-exclusion technique. However, for simplicity in our experiments we use `IASCAR` only on simple cycles, i.e., only first and last vertex repeat. While in theory, as stated in Corollary 3, we need to take all cycles into account to obtain an exact result, our use of `IASCAR` *approximates by overcounting*.

In order to obtain the CCGs, we use a chain that consists of (a) constructing a positive dependency graph from ground input program and encoding simple cycles as unsupported constraints for later use separately; (b) converting extended rules of the ground input program (`gringo`) into normal rules (`lp2normal`); (c) constructing as CNF the completion of the resulting program (`lp2sat`); and (d) compiling CNF into an (sd-D)NNF (`c2d`). Alternatively, when converting programs into CNF instances for directly counting the number of answer sets, we insert a Step (b1) after (b) which adds loop formulas (`lp2atomic`) and obtain the count after Step (d) without using `IASCAR`.

We design a small experiment to study the questions: (1) are modern knowledge compilers capable of outputting sd-DNNFs that allow for counting supported models or can we even output sd-DNNFs that allow for counting answer sets; (2) do we benefit from counting on sd-DNNFs when counting many times for counting under assumptions; (3) how much do we benefit from our approach to systematically reduce overcounting.

We take instances that encode a prototypical ASP domain with reachability (nrp_*) and use of transitive closure [4] containing cycles. This problem distinguishes from simple SAT for which we could use knowledge compilers without encoding a program into CNF by using level mappings or loop formulas. Therefore, we take as instances real-world graphs of public transport networks from all over the world, which were used in the PACE'16 and '17 challenges. In addition, we chose 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; an encoding for stable extensions of an argumentation framework instance (AF_stable) [5]; the 3-coloring problem on a graph (3_coloring); an encoding that ensures arbitrary 2-coloring for the same graph (arb_2_coloring). These instances admit no simple cycles. In general, the instances result in varying NNF sizes, CCG sizes, and number of simple cycles, answer sets and supported models. Prototypical problems benefiting from counting many times are probabilistic settings or navigation problems. These domains are quite unexplored due to absence of ASP systems, and to the best of our knowledge, there are no standard benchmark sets for counting under assumptions. For counting under assumptions, we selected a small number of atoms (3) in the program to keep it consistent and having a sufficiently high number of solutions. We selected uniform at random.

We ran the experiments on an 8-core intel I7-10510U CPU 1.8 GHz with 16 GB of RAM on Manjaro Linux 21.1.1 (Kernel 5.10.59-1-MANJARO). We follow standard guidelines for empirical evaluations; runtime is measured by `perf`.

Before we state the results, we formulate expectations from the design of experiment and our theoretical understanding. (E1): We anticipate limitations of counting answer sets when compiling plain level encodings/loop formulas,

as generating sd-DNNF takes long. (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. (E2.3): Counting works fast on instances with few cycles. Otherwise, depth restriction makes our approach utilizable. (E3): There are instances on which simple cycles are insufficient for counting answer sets.

We summarize our results in Table 1. (O1): From column sat[s], we can see that constructing an sd-DNNF of a CNF, which encodes answer sets of an input program, and subsequent counting varies notably. For example, 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_california, nrp_hanoi, or nrp_berkshire we observe a high runtime; in particular, there we see that sd-DNNFs can become quite large. By correlating this observation with column #S, we can see that instances, which can be solved fast, have no simple cycles. This matches with our expectation E1 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 encodings/loop formulas, as stated in column comp[s], works much faster. This is particularly visible for instances nrp_california, nrp_hanoi, nrp_berkshire, nrp_bart, nrp_aircoach, or nrp_a1210993. (O2): From column T[s], we can see that 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, 12_queens by 19.3, 3x3_grid by 3.7, or AF_stable by 3.4. This confirms Expectation (E2.1). However, 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. (O3): From columns #SC, depth, and A[s], we can see that the runtime 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. (O4): 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. Restricting depth d to 2 led to overcounting for instance nrp_hanoi. (O5): Finally, there is one instance, namely, nrp_autorit, for which we overcounted by 3 when restricting to simple cycles, which confirms Expectation (E3). However, on all other instances, we obtained the exact number of answer sets.

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.

6 Conclusion and Future Work

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 us with a basic toolbox for counting answer sets. However, compilations suffer from an overhead when constructing CNFs. One can view our approach 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 overcounting that we obtain when simply considering supported models. Further, we introduce domain specific simplification techniques on counting graphs.

We expect our technique to be useful for navigating answer sets or answering probabilistic questions on ASP programs, requiring 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. A large scale in-depth analysis of benefits of various counting techniques, such as enumeration on instances with few expected answer sets, approximations if one needs to count only once, or knowledge compilations could be fruitful for users. From the theoretical side, questions on effectiveness of knowledge compilations in ASP might be interesting similar to considerations for formulas [3].

Acknowledgements. Research was funded by the DFG through the Collaborative Research Center, Grant TRR 248 project ID 389792660, the BMBF, Grant 01IS20056_NAVAS, the Vienna Science and Technology Fund (WWTF) grant ICT19-065, and the Austrian Science Fund (FWF) grants P32830 and Y698.

References

1. Bogaerts, B., den Broeck, G.V.: Knowledge compilation of logic programs using approximation fixpoint theory. TPLP **15**(4–5), 464–480 (2015)
2. Darwiche, A.: Compiling knowledge into decomposable negation normal form. In: IJCAI 1999, pp. 284–289. Morgan Kaufmann (1999)
3. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002)
4. Eiter, T., Hecher, M., Kiesel, R.: Treewidth-aware cycle breaking for algebraic answer set counting. In: KR 2021, vol. 18, pp. 269–279 (2021)
5. Fichte, J.K., Gaggl, S.A., Rusovac, D.: Rushing and strolling among answer sets - navigation made easy. In: AAAI 2022 (2022)
6. Fierens, D., et al.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. TPLP **15**(3), 358–401 (2015)
7. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. **187–188**, 52–89 (2012)
8. Kabir, M., Everardo, F., Shukla, A., Fichte, J.K., Hecher, M., Meel, K.: ApproxASP - a scalable approximate answer set counter. In: AAAI 2022 (2022, in Press)

9. Lagniez, J.M., Marquis, P.: An improved decision-DDNF compiler. In: IJCAI 2017, pp. 667–673. The AAAI Press (2017)
10. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Trans. Comput. Log. **7**(2), 261–268 (2006)
11. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K.R., Marek, V.W., Truszczynski, M., Warren, D.S. (eds.) The Logic Programming Paradigm, pp. 375–398. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-642-60085-2_17
12. Sang, T., Beame, P., Kautz, H.: Performing Bayesian inference by weighted model counting. In: AAAI 2005. The AAAI Press (2005)
13. Wang, Y., Lee, J.: Handling uncertainty in answer set programming. In: AAAI 2015, pp. 4218–4219. The AAAI Press (2015)