ApproxASP – a Scalable Approximate Answer Set Counter

Mohimenul Kabir[1], Flavio O Everardo[2], Ankit K Shukla[3], Markus Hecher[4], Johannes Klaus Fichte[4], Kuldeep S Meel[1]

[1] National University of Singapore, Singapore
[2] Tec de Monterrey Campus Puebla, Mexico
[3] JKU, Linz, Austria
[4] TU Wien, Vienna, Austria

Abstract

Answer Set Programming (ASP) is a framework in artificial intelligence and knowledge representation for declarative modeling and problem solving. Modern ASP solvers focus on the computation or enumeration of answer sets. However, a variety of probabilistic applications in reasoning or logic programming require counting answer sets. While counting can be done by enumeration, simple enumeration becomes immediately infeasible if the number of solutions is high. On the other hand, approaches to exact counting are of high worst-case complexity. In fact, in propositional model counting, exact counting becomes impractical. In this work, we present a scalable approach to approximate counting for ASP. Our approach is based on systematically adding parity (XOR) constraints to ASP programs, which divide the search space. We prove that adding random XOR constraints partitions the answer sets of an ASP program. In practice, we use a Gaussian elimination-based approach by lifting ideas from SAT to ASP and integrate it into a state of the art ASP solver, which we call ApproxASP. Finally, our experimental evaluation shows the scalability of our approach over existing ASP systems.

Introduction

Answer Set Programming (ASP) (Brewka, Eiter, and Truszczyński 2011; Gebser et al. 2012) is a form of declarative programming, which is based on the stable model semantics of logic programming (Gelfond and Lifschitz 1991). Two decades of progress in the theory and practice of solving ASP offers rich modeling and solving framework that has well-known applications (Balduccini, Gelfond, and Nogueira 2006; Niemelä, Simons, and Soininen 1999; Guziolowski et al. 2013; Schaub and Woltran 2018) in the area of knowledge representation and reasoning, and artificial intelligence.

The problem of counting the number of solutions to a given ASP program, known as #ASP, is a computationally intriguing problem that has applications in probabilistic reasoning (Lee, Talsania, and Wang 2017), planning (Fichte, Gaggl, and Rusovac 2022). The computational complexity of #ASP for disjunctive programs is $\# \cdot$ coNP-complete (Fichte et al. 2017), which further increases to $\# \cdot \Sigma_2^P$-complete (Fichte and Hecher 2019) if counting with respect to a projection.

Counting solutions is a well-studied theoretical problem in mathematics and computer science since its introduction in the late 1970s (Durand, Hermann, and Kolaitis 2005; Hemaspaandra and Vollmer 1995; Valiant 1979). For the propositional model counting problem, #SAT for short, $\#\cdot$P-completeness was established quite early (Valiant 1979). Later Toda (1991) showed that polynomially many calls to a #SAT oracle can be used to capture the entire polynomial hierarchy. Recently, there has been growing interest in the application side of counting solutions to problems (Chakraborty, Meel, and Vardi 2016; Domshlak and Hoffmann 2007; Gomes, Sabharwal, and Selman 2009; Chavira and Darwiche 2008; Darwiche 2020).

Approximate counting (ApproxMC) (Chakraborty, Meel, and Vardi 2016; Soos and Meel 2019; Soos, Gocht, and Meel 2020) showed to be particularly successful in the recent competition (Fichte, Hecher, and Hamiti 2021). The core idea is hashing-based frameworks, which partition the solution space into *roughly equal small* cells of solutions by employing pairwise independent hashing functions using XOR-constraints, and then the count is obtained by enumerating solutions in one of the randomly chosen cells. Motivated by the development of scalable techniques for propositional model counting, there has been a surge of interest in designing scalable counting techniques for problems whose decision problem lies beyond NP (Bendík and Meel 2020, 2021). Inspired by the success of the aforementioned efforts, we investigate the design of scalable techniques for #ASP. Of particular interesting to us are hashing-based frameworks developed in the context of approximate model counting. In this context, one may wonder whether #ASP can be reduced to #SAT, since then invoking a counter such as ApproxMC suffices. However, it is well-known (and observed in practice) that such a reduction might result in exponential blow-up (Lifschitz and Razborov 2006).

In this paper, we present an approach to approximate ASP counting and establish a scalable solver, which is based on an incremental implementation for Gauss-Jordan elimination.

Contributions. The primary contribution of this work is the design of, to the best of our knowledge, the first scalable technique for #ASP that provides rigorous (ε, δ) guarantees. From the technical perspective, we lift the XOR-based hashing framework developed in the context of propositional model counting to ASP. As is witnessed in the development of ApproxMC, designing a scalable counter requires engineering of the underlying ASP solver to handle XOR constraints. To this end, we present the first ASP solver that can natively handle XOR constraints via Gauss-Jordan elimination (GJE). Our experimental study illustrates that for disjunctive logic programs, ApproxASP performs well. ApproxASP solved 185 instances among 200 instances, while the best ASP solver *clingo* solved a total of 177 instances. In addition, on normal logic programs ApproxASP performs on par with state-of-the-art approximate model counter ApproxMC, thereby positioning ApproxASP as the tool of choice in the context of counting for ASP programs.

Related Work. Fichte et al. (2017) established an implementation, called *dynasp*, for counting the number of answer sets. It is based on dynamic programming on tree decompositions and theoretically performs well if the treewidth is low. In practice, a decomposition heuristic is required that can find a tree decomposition of low width fast. Due to theoretical restrictions, an instance can easily have high treewidth simply if it contains a large rule. Since most encodings are not treewidth aware (Hecher 2022), the approach has certain theoretical limitations. Janhunen (2006), Janhunen and Niemelä (2011), and Bomanson, Gebser, and Janhunen (2016) established compilation techniques that transform ASP programs into SAT instances. Most of these techniques preserve the number of answer sets. However, unless the considered program is tight (Kanchanasut and Stuckey 1992; Ben-Eliyahu and Dechter 1994), there can be an exponential overhead (Lee and Lifschitz 2003; Lifschitz and Razborov 2006). Nonetheless, the compilation can be used to transform an input instance and use the existing model counter, e.g., (Chakraborty, Meel, and Vardi 2013). A recent extension to the ASP solver *clingo* (Everardo et al. 2019), called *xorro*, introduces a variety of parity constraints into ASP solving, relying on ASP encodings of parity constraints or theory propagators (Gebser et al. 2016). In contrast, our implementation relies on a dedicated high performant implementation of Gauss-Jordan elimination (GJE).

Preliminaries

We assume familiarity with graph theory (Bondy and Murty 2008) and propositional satisfiability (SAT) (Kleine Büning and Lettman 1999; Biere et al. 2009). Let U be a set of propositional *atoms*. Sometimes we use *variable* instead of atom in the context of propositional satisfiability.

Propositional Satisfiability. A *literal* is an atom or its negation. A *clause* is a finite set of literals, a (CNF) formula is a finite set of clauses. An *assignment* is a mapping $\tau : X \to \{0, 1\}$ defined for a set $X \subseteq U$ of atoms. For $x \in X$ we put $\tau(\neg x) = 1 - \tau(x)$. By 2^X we denote the set of all truth assignments $\tau : X \to \{0, 1\}$. By default, we assume that $\tau(\top) = 1$ and $\tau(\bot) = 0$ for the constants $\top$ and $\bot$. The *reduct* of a formula F with respect to $\tau \in 2^X$ is the formula $F_\tau := \{c \setminus \tau^{-1}(0) \mid c \in F, c \cap \tau^{-1}(1) = \emptyset\}$. τ *satisfies* F if $F_\tau = \emptyset$ and *dissatisfies* F if $\emptyset \in F_\tau$.

Answer Set Programs

We follow standard definitions of propositional ASP (Brewka, Eiter, and Truszczyński 2011). A *literal* is an atom $a \in U$ or its negation $\neg a$. A *program* P is a set of *rules* of the form

$$a_1 \vee \ldots \vee a_l \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m \qquad (1)$$

where $a_1, \ldots, a_l, b_1, \ldots, b_n, c_1, \ldots, c_m$ are atoms and l, n, m are non-negative integers. We write $H(r) = \{a_1, \ldots, a_l\}$, called *head* of r, $B^+(r) = \{b_1, \ldots, b_n\}$, called *positive body* r, and $B^-(r) = \{c_1, \ldots, c_m\}$, called *negative body* of r. We denote the sets of atoms occurring in a rule r or in a program P by $\mathrm{at}(r) = H(r) \cup B^+(r) \cup B^-(r)$ and $\mathrm{at}(P) = \bigcup_{r \in P} \mathrm{at}(r)$, respectively. A rule r is *disjunctive* if it is of the form as given in Equation 1, r is *normal* if $|H(r)| \leq 1$, and r is an *(integrity) constraint* if $|H(r)| = 0$. We naturally extend properties of rules to programs. In addition, we need definitions of certain cycles of programs. A set $L \subseteq \mathrm{at}(P)$ is a *loop* in P, if it is a directed cycle in the digraph D of a given program P which has as vertices the atoms $\mathrm{at}(P)$ and a directed edge (x, y) between any two atoms $x, y \in \mathrm{at}(P)$ for which there is a rule $r \in P$ with $x \in H(r)$ and $y \in B^+(r)$ (Kanchanasut and Stuckey 1992). A normal program P is *tight* if it does not contain a loop. Multiple definitions of stable models form the basis for answer set programs (Lifschitz 2010). In the following, we state two common definitions, where the second one is more complex, but commonly used in solvers.

Minimal Model Characterization. A set M of atoms *satisfies* a rule r if $(H(r) \cup B^-(r)) \cap M \neq \emptyset$ or $B^+(r) \setminus M \neq \emptyset$. M is a *model* of P if it satisfies all rules of P, we write $M \models P$ for short. The *Gelfond-Lifschitz (GL) reduct* of a program P under a set M of atoms is the program $P^M := \{ H(r) \leftarrow B^+(r) \mid r \in P, M \cap B^-(r) = \emptyset \}$ (Gelfond and Lifschitz 1991). M is an *answer set*, sometimes also just called *stable model*, of a program P if M is a minimal model of P^M. We denote by $\mathrm{AS}(P)$ the set of all answer sets of P.

It is folklore that we cannot obtain new answer sets from introducing integrity constraints.

Observation 1. *Let P be a program, X be a set of integrity constraint rules, and $M \subseteq \mathrm{at}(P)$. Moreover, let M satisfy P, but there is a set $N \subsetneq M$ such that N satisfies P^M. Then, if M satisfies $P \cup X$, there is also a set $N' \subsetneq M$ such that N' satisfies $(P \cup X)^M$.*

Proof. Let P, X, and M be as given above, in particular, assume that M satisfies both P and $P \cup X$, but $N \subsetneq M$ is a model of P^M. Since M also satisfies $P \cup X$ by assumption, for every rule $r \in (P \cup X)$ either (i) $B^-(r) \cap M \neq \emptyset$ and hence $r \notin (P \cup X)^M$ or (ii) $B^-(r) \cap M = \emptyset$ and $r \in (P \cup X)^M$. In Case (i) the rule is not relevant when considering whether N satisfies $(P \cup X)^M$, hence we can ignore that case. In Case (ii), if (iia) $r \in P^M$, we have that N satisfies such r by assumption. If (iib) $r \in X^M$, clearly it is true that $H(r) = B^-(r) = \emptyset$ as r is a constraint. Since M satisfies $(P \cup X)$ and in particular the rule $r \in X$, we have that $B^+(r) \setminus M \neq \emptyset$. Since $N \subsetneq M$, we have in particular that $B^+(r) \setminus N \neq \emptyset$. Hence, N also satisfies $r \in P^M$. As we have considered all cases, we can conclude that the observation is true. □

Corollary 2. *Let P be a program, X be a set of constraint rules, and $M \subseteq \mathrm{at}(P)$. Then, $\mathrm{AS}(P \cup X) \subseteq \mathrm{AS}(P)$.*

Proof. Since for every model was not minimal with respect to the GF-reduct of the program, we can still construct a set that prohibits M from being a minimal model of the GF-reduct. $\square$

Unfounded Set Characterization. An alternative characterization of answer sets is based on so called *unfounded sets* (Van Gelder, Ross, and Schlipf 1988), which is widely used in state-of-the-art ASP solvers (Alviano et al. 2013; Gebser, Kaufmann, and Schaub 2013). Therefore, let P be a program and $\mathrm{def}(P) \coloneqq \{\, a \mid a \in H(r), r \in P \,\}$ the set of atoms that occur in any head of the program. The *completion formula* of a program P is defined as CNF-formula as follows: $\mathrm{compl}(P) \coloneqq \{\, \{a\} \cup \neg B^+(r) \cup B^-(r) \mid r \in P \,\} \cup \{\, \neg a, b \mid b \in B^+(r), r \in P \,\} \cup \{\, \neg a, \neg c \mid c \in B^-(r), r \in P \,\}$ (Clark 1978; Fages 1994). Moreover, let $I \subseteq \mathrm{lit}(P)$. A set $U \subseteq \mathrm{at}(P)$ is an *unfounded set* wrt. I if, for each rule $r \in P$, we have (i) $H(r) \notin U$ (ii) $B^+(r) \cap I^- \neq \emptyset$ or $B^-(r) \cap I^+ \neq \emptyset$, or (iii) $B^+(r) \cap U \neq \emptyset$. Then, M is an *answer set* of a program P if (U1) M satisfies $\mathrm{compl}(P)$ and (U2) no loop contained in M is unfounded. ASP solvers use a slightly varying characterization based on no-goods of unfounded sets; that express (ii) as a no-good (Gebser, Kaufmann, and Schaub 2013).

Answer Set Counting. Given program P, the problem of counting answer sets, #ASP for short, asks to compute the number of answer sets of P. In general, #ASP is $\#\cdot$co-NP-complete (Fichte et al. 2017). If we restrict the problem #ASP to normal programs, the complexity drops to $\#\cdot$P-complete, which is easy to see from standard reductions (Janhunen 2006).

Universal Hashing

For basics on hashing function, we refer to introductory literature, e.g., on randomized algorithms (Motwani and Raghavan 1995). An approximate counting tries to compute the number of solutions using a probabilistic algorithm approximately (Karp, Luby, and Madras 1989; Chakraborty, Meel, and Vardi 2013). Therefore, assume that $\mathrm{Sol}(I)$ consists of the set of solutions for a problem instance I. The approximation algorithm takes instance I, a real $\varepsilon > 0$ called *tolerance*, and a real δ with $0 < \delta \leq 1$ called *confidence* as input. The output is a real cnt, which estimates the cardinality of $\mathrm{Sol}(I)$ based on the parameters ε and δ following the inequality

$$\Pr\left[\frac{|\mathrm{Sol}(I)|}{(1+\varepsilon)} \leq \mathrm{cnt} \leq (1+\varepsilon)\cdot|\mathrm{Sol}(I)|\right] \geq 1-\delta. \quad (2)$$

Intuitively, a random hash function h is *k-wise independent* if for all distinct elements $x_1, \ldots, x_k$, the values $h(x_1), \ldots, h(x_k)$ are independent. Formally, let $\mathcal{H}(n,m) = \{h : \{0,1\}^n \to \{0,1\}^m\}$ be a family of hash functions. We call $\mathcal{H}$ *k-wise independent* if for any distinct $x_1, \ldots, x_k \in \{0,1\}^n$, and any $y_1, \ldots, y_k \in \{0,1\}^m$, we have

$$\Pr_{h\in\mathcal{H}}[h(x_1) = y_1 \wedge \ldots \wedge h(x_k) = y_k] = \left(\frac{1}{2^m}\right)^k \quad (3)$$

XOR Hash function. A 3-wise independent hash family $\mathcal{H}_{xor}(n,m)$ is based on random XOR constraints (Gomes, Sabharwal, and Selman 2007). The hash function $\mathcal{H}_{xor}(n,m)$ can be defined as $\mathbf{Ax}+\mathbf{B}$, where $\mathbf{x}$ is one-dimensional matrix representation of $at(P)$, $|at(P)| = n$, $\mathbf{A} \in \{0,1\}^{m\times n}$, $\mathbf{B} \in \{0,1\}^{m\times 1}$, each entry of $\mathbf{A}$ and $\mathbf{B}$ are generated according to Bernoulli distribution with a probability of 0.5.

Independent Support. Approximate counting and sampling widely use independent support (Ivrii et al. 2016) of a theory. For an ASP program P, we say a set $I \subseteq at(P)$ of atoms is an *independent support* if for any answer sets $M_1, M_2 \in \mathrm{AS}(P)$ we have that $M_1 \cap I = M_2 \cap I$, then $M_1 = M_2$. Intuitively, assigning atoms of independent support I uniquely defines an answer set. Moreover, $at(P)$ is an independent support, which is called *trivial independent support*.

Example 3. *Consider the program* $P = \{a_i \vee \bar{a_i}.\ b_i \leftarrow a_i.\ c_i \leftarrow\sim\ a_i.\}$, *where* $i = 1, \ldots, 10$. *Observe that some independent supports of program* P *are* $\{a_1, \ldots, a_{10}\}$, $\{\bar{a_1}, \ldots, \bar{a_{10}}\}$, $\{b_1, \ldots, b_{10}\}$, *and* $\{c_1, \ldots, c_{10}\}$.

Partitioning and Sampling Counts

In the course of approximate model counting, we divide the search space by so-called parity constraints. Intuitively, a parity constraint makes sure that atoms occurring in it occur only in an even or odd number in an answer set of the program. Corollary 2 shows that adding simple constraints to a program will not introduce new answer sets, which is crucial to split the search space. We could use this property and include a counter for each of the model and integrity constraints. The integrity constraints ensure that the counter is even (resp. odd) for even (resp. odd) parity. In fact, the ASP-Core-2 language already allows for representing parity constraints using aggregate rules (Faber et al. 2008)[1]. Unfortunately, such a construction is quite impractical for two reasons. First, it would require us to add far too many auxiliary variables for each new parity constraint, and second, we would suffer from the known inadequacy to scale due to grounding (Everardo et al. 2019). Hence, we aim for an incremental "propagator"-based implementation when solving parity constraints (Gebser et al. 2016).

First, we formally define parity constraints and show basic properties needed for partitioning the search space. The meaning of parity constraints follows previous works on the topic (Everardo et al. 2019). For atoms a_1 and a_2, we denote the *exclusive-or* between these two atoms by $a_1 \oplus a_2$. Let M be a set of atoms. Then, M *satisfies* $a_1 \oplus a_2$ if $M \cap \{a_1\} \cup M \cap \{a_2\} \neq \emptyset$ and $M \cap \{a_1, a_2\} \neq \{a_1, a_2\}$, i.e., *either* a_1 *or* a_2 is in M; but not both. Generalizing to n distinct atoms $a_1, \ldots, a_n$, we obtain an exclusive-or constraint r of the form $(((a_1 \oplus a_2) \ldots) \oplus a_n)$ by applying $\oplus$

[1]The ASP-Core-2 language would allow to represent it in the form of:
`:-#count{1: p(1)} = N, N\2 !=1.` and
`:-#count{X: p(X), X>1} = N, N\ 2 !=0.` resp. Note that N\ 2 is N modulo 2

consecutively. Then, r is satisfied if and only if $|M \cap \mathrm{at}(r)|$ is odd and we refer to r as *odd parity constraint*. Due to associativity, we simply write $a_1 \oplus a_2 \oplus \ldots \oplus a_n$. Analogously, an *even parity constraint*, XOR *constraint* for short, is of the form $a_1 \oplus \ldots \oplus a_n \oplus \top$ and satisfied if and only if $M \cap \mathrm{at}(r)$ is even. Parity constraints can in principle contain literals. But since we can easily rewrite them, we omit such definitions. For example, constraint $\neg a_1 \oplus a_2$ is equivalent to $a_1 \oplus a_2 \oplus \top$ and $\neg a_1 \oplus \neg a_2$ is equivalent to $a_1 \oplus a_2$, i.e., pairs of negated literals cancel parity inversion. We extend the definitions on answer set programs from the preliminaries above to include parity constraints. Following standard ASP syntax, one would expect that parity constraint rules are of the form $\leftarrow a_1 \oplus \ldots \oplus a_n \oplus \top$ and $\leftarrow a_1 \oplus \ldots \oplus a_n$, respectively. However, to simplify the presentation, we write parity constraint rules as above and call them simply *parity constraint*. We let a *parity-constrained program P* be a set of rules or parity constraints and we denote by $R(P)$ the rules and by $C(P)$ the parity constraints of P.

Semantics for Search Space Partitioning. A straightforward approach to include parity constraints in ASP would be to extend the GL-reduct as follows. The reduct of a parity-constrained program P under a set M of atoms is the program $P_M := \{\, H(r) \leftarrow B^+(r) \mid r \in R(P), B^-(r) \cap M = \emptyset \,\} \cup C(P)$. However, our next example shows that extending the popular ASP-definition and including parity constraints into the GL-reduct from above is not enough.

Example 4. *Consider the program $P = \{a \leftarrow {\sim} b; b \leftarrow {\sim} c; c \leftarrow {\sim} a; a \oplus b \oplus c\}$, which contains an odd constraint. Furthermore, take the set $M = \{a, b, c\}$. It is easy to see that M satisfies each rule of P_M, in particular, the parity constraint. Now, while $N = \emptyset$ satisfies each rule $r \in R(P)_M$, the set N does not satisfy the parity constraints. Hence, M would be an answer set of the parity-constrained program P. In other words, adding a parity constraint to a program might yield a new, counterintuitive answer set.*

From the previous example, we can conclude that simply extending the GL-reduct is not enough when introducing parity-constraints into answer set programming (for approximate counting). Hence, we suggest the following definition:

Definition 5 (Answer Sets of Parity-Constrained Programs)**.** *A set $M \subseteq \mathrm{at}(P)$ is an* answer set *of P if (i) M satisfies each rule $r \in P$ (in particular, also the parity constraints) and (ii) M is the minimal model of $(R(P))^M$. Again, we denote by $\mathrm{AS}(P)$ the set of all answer sets of P.*

In the following observation, we show that parity-constraints according to our definition do not introduce new answer sets, which is crucial to search space partitioning.

Observation 6. *Let P be a program and X a set of parity constraints. Then, $\mathrm{AS}(P \cup X) \subseteq \mathrm{AS}(P)$.*

Proof. Assume that $M \subseteq \mathrm{at}(P)$ is an answer set of $P \cup X$. Since M satisfies every non-parity rule in $P \cup X$, M satisfies every rule $r \in P$. Since parity constraints do not occur in the program $(R(P))^M$ by Definition 5, for every $M \subseteq \mathrm{at}(P)$ satisfying P and $N \subsetneq M$ we have that N satisfies $(R(P))^M$. In other words, a set N that shows that M is not minimal with respect to the program $(R(P))^M$, still shows that M is not minimal when parity constraints are added. In turn, we cannot accidentally introduce new answer sets by adding the constraints, which establishes the observation. □

Well-Defined for Unfounded Set Characterization. Since we have seen certain pitfalls above and we aim for using parity constraints in an ASP solver such as clasp, we quickly review the behavior of parity constraints under an unfounded set characterization. Therefore, we extend the definitions for programs: a set $M \subseteq \mathrm{at}(P)$ is an *answer set* of a *parity-constrained program P under the unfounded set semantics* if (UP1) M satisfies $\mathrm{compl}(R(P))$, (UP1a) M satisfies $C(P)$, and (UP2) no loop contained in M is unfounded. We denote by $\mathrm{AS}_{\mathrm{uf}}(P)$ the set of all answer sets of P according to the unfounded set definition from above. Since parity constrains only apply to the model part, we observe that answer sets remain the same for both definitions. So, parity constraints only restrict answer sets, but never introduce new ones.

Observation 7. *Let P be a program and X a set of parity constraints. Then, $\mathrm{AS}_{uf}(P \cup X) \subseteq \mathrm{AS}_{uf}(P)$.*

Proof. The main argument is that propositional logic is monotonic and we conclude that Condition (UP1a) will only remove models. However, we can also show the following stronger statement $\mathrm{AS}_{\mathrm{uf}}(P \cup X) \subseteq \mathrm{AS}_{\mathrm{uf}}(P)$. Assume that M is an answer set of $P \cup X$. By Condition (UP1) we have that M satisfies $\mathrm{compl}(R(P))$. Since $P = R(P \cup X)$, M satisfies $\mathrm{compl}(P)$ and Condition (U1) is trivially satisfied. Since for every rule $r \in C(X)$ we have that $H(r) = \emptyset$, we can conclude that the positive dependency digraphs remains the same, i.e., $D^+_P = D^+_{P \cup X}$. Thus, the loops of $P \cup X$ and P are the same. In consequence, since Condition (UP2) holds for M, it allows us to conclude that Condition (U2) is also satisfied. Finally, from the folkore that propositional logic is monotonic and hence Condition (UP1a) will only remove models from $P \cup X$, we conclude that our claim $\mathrm{AS}_{\mathrm{uf}}(P \cup X) \subseteq \mathrm{AS}_{\mathrm{uf}}(P)$, which in turn establishes the observation. □

Approximate Counting

The central idea for approximate counting is to employ hash functions for sampling the search space uniformly (Motwani and Raghavan 1995). First, one partitions the set $\mathrm{Sol}(I)$ of solutions of an input instance I into roughly equally small cells. Then, one picks a random cell, counts the number s of solutions in the cell, and scales s by the number c of cells to obtain an ε-approximate estimate of the count c. A priori, we do not know the distribution of the set $\mathrm{Sol}(I)$ of solutions. Hence, we have to hash without knowledge of the distribution of the solutions, i.e., partition $\mathrm{Sol}(I)$ into cells uniformly and independently. Interestingly, this can be resolved by using a k-wise independent hash function (Gomes, Sabharwal, and Selman 2007), for which we refer to the preliminaries. In the context of approximate sampling and counting techniques, the most exploited hash family is $\mathcal{H}_{xor}$ (Soos, Gocht, and Meel 2020). We use an already evolved algorithm from the case of propositional satisfiability (Chakraborty, Meel, and Vardi 2013, 2016) and lift it to ASP.

Listing 1: Approximate Counting (ApproxASP)

```
Data: Program $P$, Independent support $I$,
      tolerance $\varepsilon$, confidence $\delta$
Result: Approximate number of answer sets
1  $C \leftarrow \{\}$ ;                         // Sampled counts
2  $n_c \leftarrow 2$ ;                         // Number of Cells
3  $p \leftarrow 1 + \lceil 9.84 \cdot (\frac{\varepsilon}{1+\varepsilon}) \cdot (1 + \frac{1}{\varepsilon})^2 \rceil$ ;     // Pivot
   /* Try to enumerate $p + 1$ many
      answer sets projected to $I$         */
4  $S =$ Enum-k-AS$(P, p + 1, I)$;
   /* Enumerated less answer sets?    */
5  if $|S| \leq p$ then return $|S|$ ;
6  for $i \leftarrow 0$ to $\lceil 17 \cdot \log_2 \frac{3}{\delta} \rceil$ by $1$ do
       /* Divide search space and sample
          at most $p + 1$ solutions          */
7      $(n_c, s) \leftarrow$ DivideNSampleCell$(P, I, p + 1, n_c)$ ;
       /* Keep estimate if search space
          was actually divided             */
8      if $n_c > 0$ then $C \leftarrow C \cup \{n_c \cdot s\}$;
9  end
10 return median$(C)$ ;
```

Listing 2: DivideNSampleCell

```
Data: Program $P$, Independent support $I$, pivot p,
      number $n'_c$ of cells from the previous round
Result: Number $n_c$ of cells that where divided;
        Count $c$ for the sampled cell
   /* Randomly choose $|I|$ many
      constraints $X \in \mathcal{H}_{xor}(|I|, |I| - 1)$ of
      length $|I - 1|$ over variables          */
1  $X \leftarrow$ ChooseXOR$(I, |I|, |I| - 1)$;
   /* Enumerate $p$ answer sets               */
2  $S =$ Enum-k-AS$(P \cup X, p, I)$ ;
   /* Enumerated less answer sets?          */
3  if $|S| \geq p$ then return $(0, 0)$;
   /* Estimate size of cells                */
4  $m \leftarrow$ LogASPSearch$(P, I, X, p, \log_2 n'_c)$ ;
   /* Pick $k$ XOR constraints                */
5  $Y \leftarrow$ Choosek$(X, m)$ ;
   /* Enum answer sets for one cell         */
6  $S \leftarrow$ Enum-k-AS$(P \cup Y, p, I)$ ;
7  return $(2^m, |S|)$
```

The algorithm is given in Listings 1 and 2. The ApproxASP algorithm takes as input an ASP program P, an independent support I for P, a tolerance ε $(0 < \varepsilon \leq 1)$, and a confidence δ with $0 < \delta \leq 1$. Note that we can always use a trivial independent support I consisting of all atoms of the program. First, sampled counts and the number of cells are initialized. Then, in Line 3, we compute a threshold pivot p that depends on ε to determine the chosen value of the size of a small cell. The seemingly magic constant originates in a probabilistic analysis using Chernoff and Chebyshev inequalities for counting sets (Chakraborty, Meel, and Vardi 2016). Then, it checks if the input program P has at least a pivot number of answer sets projected to I (**Enum-k-AS**); otherwise, we are trivially done and return the answer set count $|S|$. Subsequently, the algorithm continues and calculates a value $r := 17 \cdot \log_2 {}^3\!/\!_\delta$ that determines how often we need to sample for the requested confidence δ. The value again originates in a probabilistic analysis (Chakraborty, Meel, and Vardi 2013). Next, we divide the search space and sample a cell at most r times using the function **DivideNSampleCell**. If the attempt to split into at least 2 cells worked, represented by a non-zero number of cells, we store the count and estimate the total count by taking the number of cells times the count of the sampled cell. The final estimate of the count returned by ApproxASP is the median of the estimates stored in C, computed in Line 10.

Function **DivideNSampleCell** takes as input an ASP program P, an independent support I, a pivot p, and the number n'_c of cells from the previous round. It returns the number of constructed cells from the chosen XOR constraints and an ε-approximate estimate of the count of the answer sets of program P. Then, we check (lower bound) whether the program together with the XOR-constraint ($P \cup X$) has at most p answer sets by simply enumerating at most p answer sets of $P \cup X$. Intuitively, if the cell contains more answer sets than the pivot, we have selected XOR-constraints unfavorable as we cannot count the number of answer sets in the considered cell. We proceed with finding the "right" number of XOR-constraints to take from the chosen XOR-constraints (**LogASPSearch**), which has an underlying idea that the partitioned cell is large enough, but not too large; using exponential search. The function is somewhat involved, but follows previous work on propositional model counting (Chakraborty, Meel, and Vardi 2016). Then, we pick k of the XOR-constraints in Line 5. We enumerate at most p solutions projected to I for the program P together with the selected k XOR-constraints. Finally, we return the number of cells, which is 2^m, and the number of answer sets of the sampled cell. Overall, the number of satisfiability calls is $\mathcal{O}(p \cdot \log |C|)$.

Interestingly, our algorithm relies only on the property that adding constraints does not increase the number of solutions, which is needed for Line 4 to 6. In fact, in Observation 6, we already established the property and the remaining construction directly works due to results on propositional satisfiability where the construction is based on basics of probability theory that works for sets $S \subseteq \{0, 1\}^n$ (Chakraborty, Meel, and Vardi 2016).

Implementation Details

Most of the computation of approximate model counting is involved in satisfiability checking. Similar to ApproxMC, the consistency checking of ApproxASP consists of two subroutines, the answer set solving and the XOR solving. For the answer set solving, ApproxASP uses *clingo* as the underlying solver. The ASP solver addresses the answer set computation while invoking the XOR solving subroutine.

Everardo et al. (2019) discussed the impact of eagerly or lazily translating XOR constraints into ASP encodings. An eager translation is practically infeasible due to an exponential blowup in the number of constraints. Alternatively, lazy translation relies on non-trivial XOR solving techniques in ASP using theory propagators. We go beyond this approach and implement a dedicated, sophisticated theory propagator. We use an implementation for Han and Jiang's *Gauss-Jordan elimination (GJE)* (Han and Jiang 2012) that has been integrated into the state-of-the-art SAT solver *Cryptominisat* to extend *clingo* by full XOR solving. There, *clingo* searches for an answer set while providing the XOR subroutine a (partial) assignment such that the GJE deduces its satisfiability. An answer set is reached if both sub-routines are satisfied.

Gauss-Jordan Elimination. Han and Jiang (2012) proposed a framework for Gauss-Jordan elimination representing a set of XOR constraints as matrix $M = [A|b]$, where A is an $m \times n$ matrix coupled with a parity constraint b. Each row and column in A represents an XOR and a variable, respectively. This method suits perfectly to remove linearly dependent equations while incrementally updating M.

The framework uses the *well-known* two-watched literals scheme per XOR constraint, where one watched literal is called *basic* and the other one is *non-basic*. Basic variables are on the diagonal of the matrix in reduced row-echelon form. Instead of removing a column when the corresponding variable is assigned, GJE computes the state of the XOR constraint after one of the watched-literals is assigned. The state of an XOR constraint is exactly one of the following: (i) conflicted, (ii) propagated, (iii) satisfied, and (iv) new watch variable assigned. An XOR constraint is propagating if all except one literal are assigned. It is satisfied or conflicted, respectively, if all of its variables are assigned, and the truth value of the literals satisfies or dissatisfies, respectively, the XOR-constraint. If an XOR constraint has more than one unassigned literal, it must watch a new literal, i.e., there is no determination of the satisfiability of the XOR.

To compute a conflict and propagation clause, we start with an empty clause and scan forward the corresponding row of the matrix. For all set bits, we insert the corresponding variable into the clause with the negative or positive phase if it is assigned false or true, respectively. In the case of propagation, we insert the unassigned variable with the appropriate phase, such that the number of positive phases is odd or even if the parity of the XOR is even or odd, respectively.

Example 8. *Let* $[10111]$ *be the matrix representation of an* XOR *where the columns correspond to variables* x_1, x_2, x_3, x_4, x_5 *and the parity is* 1. *If the assignment is* $[?0110]$, *then the* XOR *constraint is propagating, and the clause is* $x_1 \vee \neg x_3 \vee \neg x_4 \vee x_5$. *If the assignment is* $[00101]$, *then the* XOR *constraint is conflicting, and the conflict clause is* $x_1 \vee \neg x_3 \vee x_4 \vee \neg x_5$.

The communication of *clingo* with the XOR sub-routing occurs during the following two tasks:

1. **Watch Literal Assigned/Reassigned**: If the truth value of one of the watch literals of an XOR is changed, XOR solver computes the XOR constraint's state. The state of the XOR will be (exactly) one of the following: (i) conflicted, (ii) propagated, (iii) satisfied, and (iv) has unassigned variable. We refer to this procedure by $\mathbf{propagate}$.
2. **Propagation Fixpoint**: The XOR solver computes the state of all XOR constraints at the propagation fixpoint or getting a new stable model. In this case, the state of the XOR will be (exactly) one of the following: (i) conflicted, (ii) propagated, (iii) satisfied, and (iv) undetermined. We refer to this procedure by $\mathbf{check}$.

Further Optimization of XOR Solving. We utilize some heuristics further to speed up the XOR solving of ApproxASP.

Heuristic 9. *If the state of an* XOR *is satisfied, the state will be unchanged as long as the ASP solver does not backtrack.*

For each XOR, we keep a separate bit to store whether the XOR is satisfied or not. If the state of an XOR is satisfied, we set the bit. Later, we do not compute the state as long as the corresponding bit is set. However, if the ASP solver backtracks, we clear the corresponding bit. We use the optimization in both $\mathbf{propagate}$ and $\mathbf{check}$.

Heuristic 10. *If both basic and non-basic variables of an* XOR *are unassigned, then the state of the* XOR *is undetermined.*

The $\mathbf{check}$ function is invoked on each propagation fixpoint, which computes the state of each XOR constraint. If an XOR is undetermined, it is unnecessary to compute its state. To reduce the unnecessary computation, we skip computing the state of an XOR if both of its basic and non-basic variables are unassigned.

Heuristic 11. *Constructing* XOR *constraints from independent support.*

Similar to the observations in the context of CNF-XOR solving, the runtime of our implementation degrades with an increase in the size of XOR. From the definition of $\mathcal{H}_{xor}$, each XOR is constructed by randomly choosing each variable with a probability of 0.5. Thus the expected size of XOR is half of the number of variables. Chakraborty et al. (2016) observed that the construction of XORs over the independent support of a formula suffices. As a result, the expected size of XOR is half of the number of variables in the independent support. Subsequently, Ivrii et al. (Ivrii et al. 2016) proposed an efficient technique for computation of independent support for CNF formulas relying on the progress of Minimal Unsatisfiable Subset (MUS) techniques.

Computing Independent Support. Since there is no off the shelf independent support computation technique for general ASP, we focus on the subset of ASP programs (namely, normal programs[2]) for which we can rely on standard translations of ASP programs to SAT (Fages 1994; Janhunen and Niemelä 2011; Bomanson 2017). We employ a tool for independent support computation in the context of CNF formulas.

Experiments

We conducted a preliminary experimental evaluation to assess the run-time performance of ApproxASP and the quality of its approximation. We consider the following questions:

RQ1 How does ApproxASP compare to existing systems?

[2]Unless, we cannot compute independent support for disjunctive programs, the heuristic is only taking effect on normal programs.

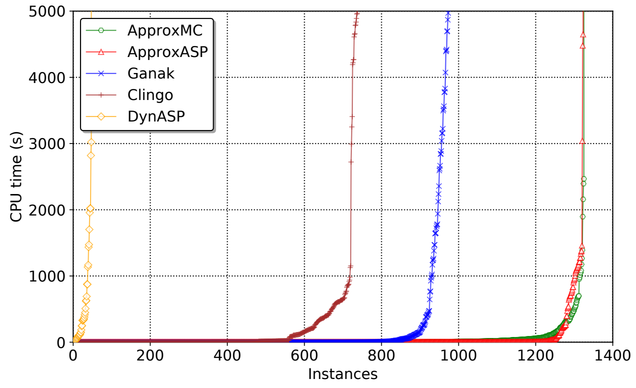

Figure 1: Runtime of various tools for normal programs. The x-axis refers to the number of instances; y-axis depicts the runtime individually sorted in ascending order per solver.

RQ2 How does ApproxASP efficient for handling ASP programs with XOR constraints (ASP+XOR)?

RQ3 Does ApproxASP output approximate counts that are indeed between $(1+\varepsilon)^{-1}$ and $(1+\varepsilon)$ of the exact counts?

Environment. All experiments were carried out on a high-performance computer cluster, where each node consists of an 2xE5-2690v3 CPUs running with 2x12 real cores and 96GB of RAM. The runtime was limited to 1000 and 5000 seconds, for ASP+XOR solving and approximate counting, respectively. We follow standard guidelines for empirical evaluations (van der Kouwe et al. 2018; Fichte et al. 2021)

Other tools. We selected four programs that allow for counting answer sets, namely, DynASP v2.0 (Fichte et al. 2017), *clingo* v5.4.0 (Gebser et al. 2007), Ganak (Sharma et al. 2019), ApproxMC4 (Soos, Gocht, and Meel 2020). We compared the ASP+XOR solver of ApproxASP with *xorro* (Everardo et al. 2019). *clingo* can count answer sets by enumeration. While Ganak and ApproxMC are the state-of-the-art exact and approximate propositional model counters, respectively, they can be used to count answer sets using translations to SAT for certain classes of programs (Fages 1994; Janhunen and Niemelä 2011; Bomanson 2017). We run Ganak with cache size bounded to 2000 MB and compute the sampling set of the SAT instance to run ApproxMC. In line with previous works in approximate counting, we set $\varepsilon = 0.8$ and $\delta = 0.2$ for both ApproxMC and ApproxASP.

Instances. We follow a similar approach on selecting instances as previous works (Aziz et al. 2015; Fichte et al. 2017). We divide the benchmarks into (i) normal programs and (ii) disjunctive programs. For normal programs (i), we chose from different well-known graph problems encoded into ASP programs, where we selected counting variants for the vertex cover, independent set, dominating set, graph reachability, and r-arborescence problem. We randomly generated 1500 graphs, consisting of at most 50 vertices and 250 edges taken equally from each class. For disjunctive programs (ii), we use the projected model counting problem on 2QBFs, which is known to be #·co-NP-complete (Durand, Hermann, and Kolaitis 2005). We generated 200 random instances, where the number of variables and clauses are at most 100 and 200, respectively. All the generated QBF instances are 3-DNFs. To generate ASP+XOR programs, we add as much XORs to the ASP program so that the number of answer sets in a randomly chosen cell is at most pivot p.

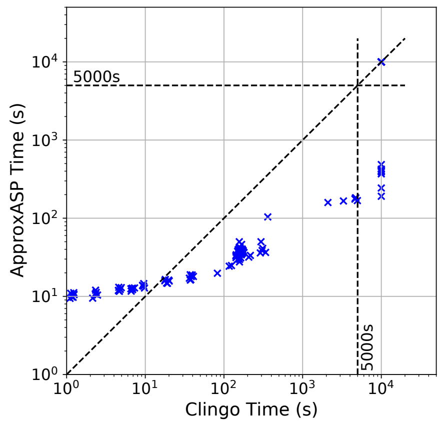

Figure 2: Clingo vs ApproxASP on disjunctive instances. If a point is below the diagonal, then ApproxASP solves it faster. The timeouted instances are shown beyond the 5000s axis.

Analysis of RQ1. The results of our experiments are shown in Figure 1 for non-disjunctive programs. The plot shows the runtimes for each counter, and a point (x, y) on the plot indicates that x instances took less than or equal to y seconds to solve. Out of the 1500 instances, ApproxMC managed to solve 1325 instances, whereas ApproxASP solved 1323 instances. Figure 2 shows the scatter plot of Clingo and ApproxASP for disjunctive problems. Although *clingo* is faster on some instances, it is clear that ApproxASP solves sufficient instances in a reasonable time, which takes a huge time to enumerate.

Table 1 shows the performance evaluation of *clingo*, DynASP, Ganak, ApproxMC, and ApproxASP on all instances. The first row shows the total number of instances, the second row shows the number of instances solved by each counter, and the third row presents the PAR-2 score [3].

The time spent in translating ASP to SAT and calculating the independent support is negligible, which is < 1s on average. For normal programs, ApproxMC and ApproxASP solved a similar number of instances. For disjunctive problems, ApproxASP solved 185 instances and *clingo* solved 177 instances. While ApproxASP solved more instances than *clingo*, we observe slightly different behavior than on normal programs. The gap between the runtime performance of ApproxASP and *clingo* is small. However, this is not entirely surprising, since we use trivial independent support for disjunctive problems.

[3]PAR-2 score is penalized average runtime that assigns a runtime of two times the timeout (without "not solved" status) for each benchmark not solved by a tool.

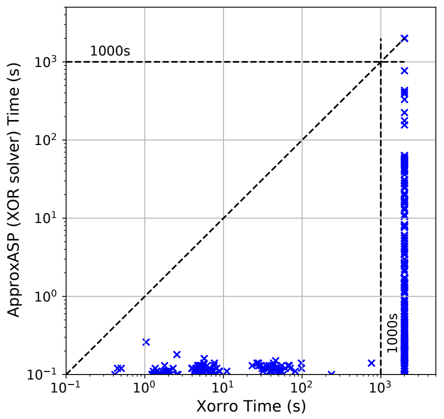

Figure 3: *xorro* vs ApproxASP (XOR solver) on ASP+XOR programs. If a point is below the diagonal, then ApproxASP (XOR solver) solves it faster. The timeouted instances are shown beyond the 1000s axis.

		C	D	G	AMC	AAS
Normal	#Instances	1500	1500	1500	1500	1500
	#Solved	738	47	973	**1325**	1323
	PAR-2	5172	9705	3606	**1200**	1218
Disj.	#Instances	200	200	200	200	200
	#Solved	177	0	0	0	**185**
	PAR-2	1372	10000	10000	10000	**795**

Table 1: The runtime performance comparison of (C) *clingo*, (D) DynASP, (G) Ganak, (AMC) ApproxMC, and (AAS) ApproxASP on all considered instances.

Analysis of RQ2. The results on ASP+XOR programs are shown in Figure 3. From the figure, it is clear that the XOR solver of ApproxASP is significantly faster than *xorro*.

Analysis of RQ3. We compare the number of solutions computed by ApproxASP with solutions returned by exact counters to assess the quality of the approximation. The results of our comparison are shown in Figure 4. We observe that ApproxASP outputs counts within the tolerance for all the instances that are close to the output of *clingo* or Ganak. Moreover, we compute the observed tolerance ε_{obs}, which is defined as $\max({}^{s}/_{|\text{AS}(P)|} - 1, {}^{|\text{AS}(P)|}/_{s} - 1)$, where s is the output given by ApproxASP. We observe a maximum value of $\varepsilon_{obs} = 0.25$ and the arithmetic mean of $\varepsilon_{obs} = 0.037$ across all instances, while the theoretical guarantee is 0.8.

Summary. Our experimental study illustrates that on the selected benchmarks of disjunctive logic programs, ApproxASP performs well. ApproxASP solved 185 instances among 200 instances, while the best ASP solver *clingo* solved a total of 177 instances. On normal programs, ApproxASP performs on par with state-of-the-art approximate model counter ApproxMC. In terms of the quality of approximation,

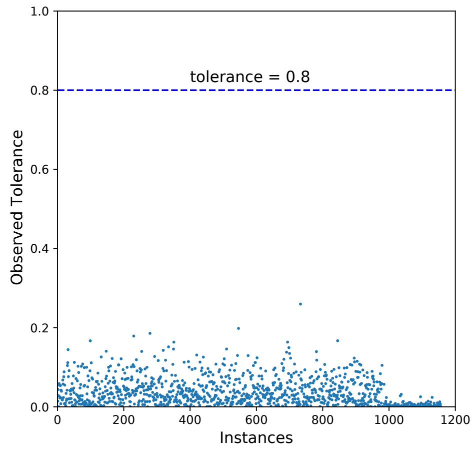

Figure 4: Visualization of the tolerance, which is computed from the estimate of ApproxASP.

ApproxASP outputs counts within $[(1+\varepsilon)^{-1}, (1+\varepsilon)]$ ratio of the exact number of answer sets for all instances.

Conclusion

In this paper, we present ApproxASP, the first scalable approximate counter for ASP programs that employs pairwise independent hash functions, represented as XOR constraints, to partition the solution space, and then invokes an ASP solver on a randomly chosen cell. To achieve practical efficiency, we augment the state of the art ASP solver, *clingo*, with native support for XORs. Our empirical evaluation clearly demonstrates that ApproxASP is able to handle problems that lie beyond the reach of existing counting techniques. Our empirical evaluation shows that ApproxASP is competitive with ApproxMC on the subset of instances that can be translated to CNF without exponential blowup and can handle instances disjunctive programs, which can not be solved via reduction to #SAT without exponential blowup. The empirical analysis, therefore, positions ApproxASP as the tool of choice in the context of counting for ASP programs.

Acknowledgments

This work has been carried out while the last three authors were visiting the Simons Institute and was supported by a Google Fellowship, the National Research Foundation Singapore [NRF- NRFFAI1-2019-0004], the AI Singapore Programme [AISG-RP-2018-005], NUS ODPRT Grant [R-252-000-685-13], the Austrian Science Fund (FWF), Grants Y698 and P32830, the Vienna Science and Technology Fund, Grant WWTF ICT19-065, Ministry of Education Singapore Tier 2 grant [MOE-T2EP20121-0011], and computational resources were provided by the NSCC (https://www.nscc.sg/).

References

Alviano, M.; Dodaro, C.; Faber, W.; Leone, N.; and Ricca, F. 2013. WASP: A Native ASP Solver Based on Constraint Learning. In *LPNMR'13*, 54–66. Springer.

Aziz, R. A.; Chu, G.; Muise, C.; and Stuckey, P. J. 2015. Stable model counting and its application in probabilistic logic programming. In *Twenty-ninth AAAI conference on artificial intelligence*.

Balduccini, M.; Gelfond, M.; and Nogueira, M. 2006. Answer set based design of knowledge systems. *Ann. Math. Artif. Intell.*, 47(1-2): 183–219.

Ben-Eliyahu, R.; and Dechter, R. 1994. Propositional Semantics for Disjunctive Logic Programs. *Ann. Math. Artif. Intell.*, 12(1): 53–87.

Bendík, J.; and Meel, K. S. 2020. Approximate counting of minimal unsatisfiable subsets. In *CAV'20*, 439–462. Springer.

Bendík, J.; and Meel, K. S. 2021. Counting Maximal Satisfiable Subsets. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5): 3651–3660.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. Amsterdam, Netherlands: IOS Press. ISBN 978-1-58603-929-5.

Bomanson, J. 2017. lp2normal—A Normalization Tool for Extended Logic Programs. In *LPNMR'17*, 222–228. Springer.

Bomanson, J.; Gebser, M.; and Janhunen, T. 2016. Rewriting Optimization Statements in Answer-Set Programs. In *Tech. Comm. of ICLP'16*, volume 52, 5:1–5:15. Dagstuhl Publishing.

Bondy, J. A.; and Murty, U. S. R. 2008. *Graph theory*, volume 244 of *Graduate Texts in Mathematics*. New York, USA: Springer.

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

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2013. A Scalable Approximate Model Counter. In *CP'13*, volume 8124, 200–216. Springer.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Chavira, M.; and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6–7): 772—799.

Clark, K. L. 1978. Negation as failure. *Logic and Data Bases*, 1: 293–322.

Darwiche, A. 2020. Three Modern Roles for Logic in AI. In *PODS'20*, 229–243. Assoc. Comput. Mach., New York.

Domshlak, C.; and Hoffmann, J. 2007. Probabilistic Planning via Heuristic Forward Search and Weighted Model Counting. *J. Artif. Intell. Res.*, 30.

Durand, A.; Hermann, M.; and Kolaitis, P. G. 2005. Subtractive reductions and complete problems for counting complexity classes. *Theoretical Computer Science*, 340(3): 496–513.

Everardo, F.; Janhunen, T.; Kaminski, R.; and Schaub, T. 2019. The Return of xorro. In *LPNMR'19*, 284–297. Springer.

Faber, W.; Pfeifer, G.; Leone, N.; Dell'armi, T.; and Ielpa, G. 2008. Design and implementation of aggregate functions in the DLV system. *Theory Pract. Log. Program.*, 8(5-6): 545–580.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *J. on Methods of Logic in Computer Science*, 1(1): 51–60.

Fichte, J. K.; Gaggl, S. A.; and Rusovac, D. 2022. Rushing and Strolling among Answer Sets – Navigation Made Easy. Forthcoming.

Fichte, J. K.; and Hecher, M. 2019. Treewidth and Counting Projected Answer Sets. In *LPNMR'19*, volume 11481 of *LNCS*, 105–119. Springer.

Fichte, J. K.; Hecher, M.; and Hamiti, F. 2021. The Model Counting Competition 2020. *ACM Journal of Experimental Algorithmics*, 26(13).

Fichte, J. K.; Hecher, M.; McCreesh, C.; and Shahab, A. 2021. Complications for Computational Experiments from Modern Processors. In *CP'21*, volume 210, 25:1–25:21. Dagstuhl Publishing.

Fichte, J. K.; Hecher, M.; Morak, M.; and Woltran, S. 2017. Answer Set Solving with Bounded Treewidth Revisited. In *LPNMR'17*, 132–145. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory Solving Made Easy with Clingo 5. In *Tech. Comm. of ICLP'16*, volume 52, 2:1–2:15. Dagstuhl Publishing.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan & Claypool.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-driven Answer Set Enumeration. In *LPNMR'07*, volume 4483, 136–148. Springer.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2013. Advanced conflict-driven disjunctive answer set solving. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI'13)*, 912–918. The AAAI Press.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Comput.*, 9(3/4): 365–386.

Gomes, C.; Sabharwal, A.; and Selman, B. 2007. Near-Uniform Sampling of Combinatorial Spaces Using XOR Constraints. In *Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems (NIPS'06)*, 481–488. MIT Press.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2009. Chapter 20: Model Counting. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 633–654. Amsterdam, Netherlands: IOS Press.

Guziolowski, C.; Videla, S.; Eduati, F.; Thiele, S.; Cokelaer, T.; Siegel, A.; and Saez-Rodriguez, J. 2013. Exhaustively characterizing feasible logic models of a signaling network using Answer Set Programming. *Bioinformatics*, 29(18): 2320–2326. Erratum see Bioinformatics 30, 13, 1942.

Han, C.-S.; and Jiang, J.-H. R. 2012. When Boolean satisfiability meets Gaussian elimination in a simplex way. In

International Conference on Computer Aided Verification, 410–426. Springer.

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

Hemaspaandra, L. A.; and Vollmer, H. 1995. The Satanic Notations: Counting Classes Beyond #P and Other Definitional Adventures. *SIGACT News*, 26(1): 2–13.

Ivrii, A.; Malik, S.; Meel, K. S.; and Vardi, M. Y. 2016. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1): 41–58.

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*, 111–130. Springer.

Kanchanasut, K.; and Stuckey, P. J. 1992. Transforming normal logic programs to constraint logic programs. *Theoretical Computer Science*, 105(1): 27–56.

Karp, R. M.; Luby, M.; and Madras, N. 1989. Monte-Carlo Approximation Algorithms for Enumeration Problems. *J. Algorithms*, 10(3): 429–448.

Kleine Büning, H.; and Lettman, T. 1999. *Propositional logic: deduction and algorithms*. Cambridge University Press, Cambridge.

Lee, J.; and Lifschitz, V. 2003. Loop Formulas for Disjunctive Logic Programs. In *LP'03*, 451–465. Springer.

Lee, J.; Talsania, S.; and Wang, Y. 2017. Computing LPMLN using ASP and MLN solvers. *TPLP*, 17(5-6): 942–960.

Lifschitz, V. 2010. *Thirteen Definitions of a Stable Model*, 488–503. Berlin, Heidelberg: Springer.

Lifschitz, V.; and Razborov, A. 2006. Why are there so many loop formulas? *ACM Trans. Comput. Log.*, 7(2): 261–268.

Motwani, R.; and Raghavan, P. 1995. *Randomized Algorithms*. Cambridge University Press, Cambridge.

Niemelä, I.; Simons, P.; and Soininen, T. 1999. Stable model semantics of weight constraint rules. In *LPNMR'99*, 317–331. Springer.

Schaub, T.; and Woltran, S. 2018. Special Issue on Answer Set Programming. *KI*, 32(2-3): 101–103.

Sharma, S.; Roy, S.; Soos, M.; and Meel, K. S. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *IJCAI'19*, 1169–1176.

Soos, M.; Gocht, S.; and Meel, K. S. 2020. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *CAV'20*, 463–484. Springer.

Soos, M.; and Meel, K. S. 2019. BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

Toda, S. 1991. PP is as Hard as the Polynomial-Time Hierarchy. *SIAM J. Comput.*, 20(5): 865–877.

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

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.

Van Gelder, A.; Ross, K.; and Schlipf, J. S. 1988. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *PODS'88*, 221—230. Assoc. Comput. Mach., New York.