Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Stable Model Counting and Its Application in Probabilistic Logic Programming

Rehan Abdul Aziz, Geoffrey Chu, Christian Muise and **Peter Stuckey**
National ICT Australia, Victoria Laboratory *
Department of Computing and Information Systems
The University of Melbourne

Abstract

Model counting is the problem of computing the number of models that satisfy a given propositional theory. It has recently been applied to solving inference tasks in probabilistic logic programming, where the goal is to compute the probability of given queries being true provided a set of mutually independent random variables, a model (a logic program) and some evidence. The core of solving this inference task involves translating the logic program to a propositional theory and using a model counter. In this paper, we show that for some problems that involve inductive definitions like reachability in a graph, the translation of logic programs to SAT can be expensive for the purpose of solving inference tasks. For such problems, direct implementation of stable model semantics allows for more efficient solving. We present two implementation techniques, based on unfounded set detection, that extend a propositional model counter to a stable model counter. Our experiments show that for particular problems, our approach can outperform a state-of-the-art probabilistic logic programming solver by several orders of magnitude in terms of running time and space requirements, and can solve instances of significantly larger sizes on which the current solver runs out of time or memory.

1 Introduction

Consider the counting version of graph reachability problem: given a directed graph, count the number of subgraphs in which node t is reachable from node s (Valiant 1979). This problem can be naturally modeled as a logic program under stable model semantics (Gelfond and Lifschitz 1988). Let us say that the input is given by two predicates: $node(X)$ and $edge(X, Y)$. For each node, we can introduce a decision variable in that models whether the node is in the subgraph. Furthermore, we can model reachability ($reach$) from s as an inductive definition using the following two rules: $reach(s) \leftarrow in(s)$ and $reach(Y) \leftarrow in(Y), reach(X), edge(X, Y)$. The first rule says that s itself is reachable if it is in the subgraph. The second rule is the inductive case, specifying that a node Y is reachable if it is in the subgraph and there is a reachable node X that has an edge to it. Additionally, say there are arbitrary constraints in our problem, e.g., only consider subgraphs where a certain y is also reachable from s etc. This can be done using the integrity constraint: $\leftarrow \neg reach(y)$. The number of stable models of this program is equal to the number of solutions of the problem.

There are at least two approaches to counting the stable models of a logic program. The first is to translate a given logic program to a propositional theory such that there is a one-to-one correspondence between the propositional models of the translated program and the stable models of the original program, and use SAT model counting (Gomes, Sabharwal, and Selman 2008). We show that this approach does not scale well in practice since such translations, if done *a priori*, can grow exponentially with the input size. The second approach is to use an *answer set programming* (ASP) solver like CLASP (Gebser et al. 2007) or DLV (Leone et al. 2006) and enumerate all models. This approach is extremely inefficient since model counting algorithms have several optimizations like caching and dynamic decomposition that are not present in ASP solvers. This motivates us to build a stable model counter that can take advantage of state-of-the-art ASP technology which combines partial translation and lazy unfounded set (Van Gelder, Ross, and Schlipf 1988) detection. However, we first show that it is not correct to naively combine partial translation and lazy unfounded set detection with SAT model counters due to the aforementioned optimizations in model counters. We then suggest two approaches to properly integrate unfounded set propagation in a model counter.

We show that we can apply our algorithms to solve probabilistic logic programs (Raedt and Kimmig 2013). Consider the probabilistic version of the above problem, also called the *graph reliability* problem (Arora and Barak 2009). In this version, each node can be in the subgraph with a certain probability $1 - p$, or equivalently, fail with the probability p. We can model this by simply attaching probabilities to the in variables. We can model observed *evidence* as constraints. E.g., if we have evidence that a certain node y is reachable from s, then we can model this as the unary

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

constraint (not rule): $reach(y)$. The goal of the problem is to calculate the probability of node t being reachable from node s given the evidence. The probabilistic logic programming solver PROBLOG2 (Fierens et al. 2011) approaches this inference task by reducing it to weighted model counting of the translated propositional theory of the original logic program. We extend PROBLOG2 to use our implementation of stable model counting on the original logic program and show that our approach is more scalable.

2 Preliminaries

We consider propositional variables $\mathcal{V}$. Each $v \in \mathcal{V}$ is (also) a positive literal, and $\neg v, v \in \mathcal{V}$ is a negative literal. Negation of a literal, $\neg l$ is $\neg v$ if $l = v$, and v if $l = \neg v$. An assignment θ is a set of literals which represents the literals which are true in the assignment, where $\forall v \in \mathcal{V}. \{v, \neg v\} \not\subseteq \theta$. If o is a formula or assignment, let $vars(o)$ be the subset of $\mathcal{V}$ appearing in o. Given an assignment θ, let $\theta^+ = \{v \in \theta \mid v \in \mathcal{V}\}$ and $\theta^- = \{\neg v \in \theta \mid v \in \mathcal{V}\}$. Two assignments θ_1 and θ_2 *agree on variables* $V \subseteq \mathcal{V}$, written $\theta_1 =_V \theta_2$, if $vars(\theta_1^+) \cap V = vars(\theta_2^+) \cap V$ and $vars(\theta_1^-) \cap V = vars(\theta_2^-) \cap V$. Given a partial assignment θ and a Boolean formula F, let $F|_\theta$ be the *residual* of F w.r.t. θ. $F|_\theta$ is constructed from F by substituting each literal $l \in \theta$ with $true$ and each literal $\neg l \in \theta$ with $false$ and simplifying the resulting formula. For a formula F, $count(F)$ is the number of assignments to $vars(F)$ that satisfy F.

2.1 DPLL-Based Model Counting

State of the art SAT model counters are very similar to SAT solvers, but have three important optimisations. The first optimisation is to count solution *cubes* (i.e., partial assignments θ, $vars(\theta) \subseteq \mathcal{V}$ whose every extension is a solution) instead of individual solutions. Consider the Boolean formula: $F_1 = \{\neg b \vee a, \neg c \vee \neg a \vee b, \neg d \vee c, \neg e \vee c\}$. Suppose the current partial assignment is $\{a, b, c\}$. The formula is already satisfied irrespective of values of d and e. Instead of searching further and finding all 4 solutions, we can stop and record that we have found a solution cube containing 2^k solutions, where k is the number of unfixed variables.

The second important optimisation is caching. Different partial assignments can lead to identical subproblems which contain the same number of solutions. By caching such counts, we can potentially save significant repeated work. For a formula F and an assignment θ, the number of solutions of F under the subtree with θ is given by $2^{|vars(F)| - |vars(\theta)| - |vars(F|_\theta)|} \times count(F|_\theta)$. We can use the residual as the key and cache the number of solutions the subproblem has. For example, consider F_1 again. Suppose we first encountered the partial assignment $\theta_1 = \{d, c\}$. Then $F_1|_{\theta_1} = \{\neg b \vee a, \neg a \vee b\}$. After searching this subtree, we find that this subproblem has 2 solutions and cache this result. The subtree under θ_1 thus has $2^{5-2-2} \times 2 = 4$ solutions. Suppose we later encounter $\theta_2 = \{\neg d, e, c\}$. We find that $F_1|_{\theta_2}$ is the same as $F_1|_{\theta_1}$. By looking it up in the cache, we can see that this subproblem has 2 solutions. Thus the subtree under θ_2 has $2^{5-3-2} \times 2 = 2$ solutions.

The last optimisation is dynamic decomposition. Suppose after fixing some variables, the residual decomposes into two or more formulas involving disjoint sets of variables. We can count the number of solutions for each of them individually and multiply them together to get the right result. Consider $F_2 = \{a \vee \neg b \vee c, c \vee \neg d \vee e, e \vee f\}$ and a partial assignment $\{\neg c\}$. The residual program can be decomposed into two components $\{a \vee \neg b\}$ and $\{\neg d \vee e, e \vee f\}$ with variables $\{a, b\}$ and $\{\neg d \vee e, e \vee f\}$ respectively. Their counts are 3 and 5 respectively, therefore, the number of solutions for F_2 that extend the assignment $\{\neg c\}$ is $3 \times 5 = 15$. The combination of the three optimisations described above into a DPLL style backtracking algorithm has been shown to be very efficient for model counting. See (Bacchus, Dalmao, and Pitassi 2003; Gomes, Sabharwal, and Selman 2008; Sang et al. 2004) for more details.

2.2 Answer Set Programming

We consider $\mathcal{V}$ split into two disjoint sets of variables *founded* variables ($\mathcal{V_F}$) and *standard* variables ($\mathcal{V_S}$). An ASP-SAT *program* P is a tuple $(\mathcal{V}, R, C)$ where R is a set of *rules* of form: $a \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \neg c_1 \wedge \ldots \wedge \neg c_m$ such that $a \in \mathcal{V_F}$ and $\{b_1, \ldots, c_m\} \subseteq \mathcal{V}$ and C is a set of *constraints* over the variables represented as disjunctive clauses. A rule is *positive* if its body only contains positive founded literals. The *least assignment* of a set of positive rules R, written $Least(R)$ is one that that satisfies all the rules and contains the least number of positive literals. Given an assignment θ and a program P, the reduct of θ w.r.t. P, written, P^θ is a set of positive rules that is obtained as follows: for every rule r, if any $c_i \in \theta$, or $\neg b_j \in \theta$ for any *standard* positive literal, then r is discarded, otherwise, all negative literals and standard variables are removed from r and it is included in the reduct. An assignment θ is a stable model of a program P iff it satisfies all its constraints and $\theta =_{\mathcal{V_F}} Least(P^\theta)$. Given an assignment θ and a set of rules R, the residual rules $R|_\theta$ are defined similarly to residual clauses by treating every rule as its logically equivalent clause. A program is *stratified* iff it admits a mapping $level$ from $\mathcal{V_F}$ to non-negative integers where for each rule in the program s.t., referring to the above rule form, $level(a) > level(c_i)$ whenever $c_i \in \mathcal{V_F}$ for $1 \leq i \leq m$ and $level(a) \geq level(b_i)$ whenever $b_i \in \mathcal{V_F}$ for $1 \leq i \leq n$. In ASP terms, standard variables, founded variables and constraints are equivalent to *choice* variables, regular ASP variables, and integrity constraints resp. We opt for the above representation because it is closer to SAT-based implementation of modern ASP solvers.

3 SAT-Based Stable Model Counting

The most straight forward approach to counting the stable models of a logic program is to translate the program into propositional theory and use a propositional model counter. As long as the translation produces a one-to-one correspondence between the stable models of the program and the solutions of the translated program, we get the right stable model count. Unfortunately, this is not a very scalable approach. Translations based on adding loop formulas (Lin and Zhao 2004) or the *proof-based* translation used

in PROBLOG2 (Fierens et al. 2011) require the addition of an exponential number of clauses in general (see (Lifschitz and Razborov 2006) and (Vlasselaer et al. 2014) respectively). Polynomial sized translations based on *level rankings* (Janhunen 2004) do exist, but do not produce a one to one correspondence between the stable models and the solutions and thus are inappropriate for stable model counting.

Current state of the art SAT-based ASP solvers do not rely on a full translation to SAT. Instead, they rely on lazy unfounded set detection. In such solvers, only the rules are translated to SAT. There is an extra component in the solver which detects unfounded sets and lazily adds the corresponding loop formulas to the program as required (Gebser, Kaufmann, and Schaub 2012). Such an approach is much more scalable for solving ASP problems. However, it cannot be naively combined with a standard SAT model counter algorithm. This is because the SAT model counter requires the entire Boolean formula to be available so that it can check if all clauses are satisfied to calculate the residual program. However, in this case, the loop formulas are being lazily generated and many of them are not yet available to the model counter. Naively combining the two can give the wrong results, as illustrated in the next example.

Example 1. *Consider a program P_1 with founded variables $\{a, b\}$, standard variables $\{s\}$ and rules: $\{a \leftarrow b, b \leftarrow a, a \leftarrow s\}$. There are only two stable models of the program $\{a, b, s\}$ and $\{\neg a, \neg b, \neg s\}$. If our partial assignment is $\{a, b\}$, then the residual program contains an empty theory which means that the number of solutions extending this assignment is 2 (or $2^{|\{s\}|}$). This is clearly wrong, since $\{a, b, \neg s\}$ is not a stable model of the program.*

Now consider P_2 which is equal to P_1 with these additions: founded variable c, standard variables $\{t, u\}$ and two rules: $c \leftarrow a \wedge t$ and $b \leftarrow u$. Consider the partial assignment $\{u, a, b, s\}$, the residual program has only one rule: $c \leftarrow t$. It has two stable models, $\{c, t\}$ and $\{\neg c, \neg t\}$. Now, with the partial assignment $\{\neg u, a, b\}$, we get the same residual program and the number of solutions should be: $2 \times 2^{|\{s\}|} = 4$ which is wrong since s cannot be false in order for a, b to be true when u is false, i.e., $\{\neg u, c, t, a, b, \neg s\}$ and $\{\neg u, \neg c, \neg t, a, b, \neg s\}$ are not stable models of P_2.

In order to create a stable model counter which can take advantage of the scalability of lazy unfounded set detection, we need to do two things: 1) identify the conditions for which the ASP program is fully satisfied and thus we have found a cube of stable models, 2) identify what the residual of an ASP program is so that we can take advantage of caching and dynamic decomposition.

3.1 Searching on Standard Variables for Stratified Programs

The first strategy is simply to restrict the search to standard variables. If the program is stratified, then the founded variables of the program are functionally defined by the standard variables of the program. Once the standard variables are fixed, all the founded variables are fixed through propagation (unit propagation on *rules* and the unfounded set propagation). It is important in this approach that the propagation on the founded variables is only carried out on the rules of the program, and not the constraints. Constraints involving founded variables should only be *checked* once the founded variables are fixed. The reader can verify that in Example 1, if we decide on standard variables first, then none of the problems occur. E.g., in P_1, if s is fixed to either true or false, then we do not get any wrong stable model cubes. Similarly, in P_2, if we replace the second assignment with $\{\neg u, s\}$ which propagates $\{a, b\}$, we still get the same residual program, but in this case, it is correct to use the cached value. Note that stratification is a requirement for all probabilistic logic programs under the distribution semantics (Sato 1995). For such programs given an assignment to standard variables, the well-founded model of the resulting program is the unique stable model.

3.2 Modifying the Residual Program

In ASP solving, it is often very useful to make decisions on founded variables as it can significantly prune the search space. For this reason, we present a more novel approach to overcome the problem demonstrated in Example 1.

The root problem in Example 1 in both cases is the failure to distinguish between a founded variable being true and being *justified*, i.e., can be inferred to be true from the rules and current standard and negative literals. In the example, in P_1, a and b are made true by search (and possibly propagation) but they are not justified as they do not necessarily have externally supporting rules (they are not true under stable model semantics if we set $\neg s$). In ASP solvers, this is not a problem since the existing unfounded set detection algorithms guarantee that in *complete* assignments, a variable being true implies that it is justified. This is not valid for *partial* assignments, which we need for counting stable model cubes. Next, we show that if we define the residual rules (not constraints) of a program in terms of justified subset of an assignment, then we can leverage a propositional model counter augmented with unfounded set detection to correctly compute stable model cubes of a program. In order to formalize and prove this, we need further definitions.

Given a program $P = (\mathcal{V}, R, C)$ and a partial assignment θ, the *justified assignment* $JA(P, \theta)$ is the subset of θ that includes all standard and founded negative literals plus all the positive founded literals implied by them using the rules of the program. More formally, let $J_0(\theta) = \theta^- \cup \{v \in \theta | v \in \mathcal{V}_\mathcal{S}\}$. Then, $JA(P, \theta) = J_0(\theta) \cup \{v \in \mathcal{V}_\mathcal{F} | v \in \theta, v \in Least(R|_{J_0(\theta)})\}$.

Definition 1. *Given a program $P = (\mathcal{V}, R, C)$ and a partial assignment θ, let $J = JA(P, \theta)$ and $U = vars(\theta) \setminus vars(J)$. The justified residual program of P, w.r.t. θ is written $P|^j_\theta$ and is equal to (W, S, D) where $S = R|_J$, $D = C|_\theta \cup \{u | u \in U\}$ and $W = vars(S) \cup vars(D)$.*

Example 2. *Consider a program P with founded variables $\{a, b, c, d, e, f\}$, standard variables $\{s, t, u, x, y, z\}$ and the following rules and constraints:*

$$\begin{array}{llll} a \leftarrow b. & c \leftarrow d. & e \leftarrow \neg f. & \neg s \vee \neg t \\ b \leftarrow a. & d \leftarrow u. & f \leftarrow \neg e. & a \vee b \\ a \leftarrow s. & & & f \vee x \\ b \leftarrow t. & & & \end{array}$$

Let $\theta = \{a, b, d, u, \neg e, c, f\}$. Then, $J_0(\theta) = \{u, \neg e\}$ and $JA(P, \theta) = J_0(\theta) \cup \{d, f, c\}$. The justified residual program w.r.t. θ has all the rules in the first column and has the constraints: $\{\neg s \vee \neg t,\ a,\ b\}$.

Theorem 1. *Given an ASP-SAT program $P = (\mathcal{V}, R, C)$ and a partial assignment θ, let $P|^j_\theta = (W, S, D)$ be denoted by Q. Let the remaining variables be $\mathcal{V}_r = \mathcal{V} \backslash (W \cup vars(\theta))$ and π be a complete assignment over W. Assume any founded variable for which there is no rule in S is false in θ.*

1. *If π is a stable model of Q, then for any assignment θ_r over the remaining variables, $\theta \cup \pi \cup \theta_r$ is a stable model of P.*
2. *For a given assignment θ_r over remaining variables, if $\theta \cup \pi \cup \theta_r$ is a stable model of P, then π is a stable model of Q.*

Corollary 2. *Let the set of rules and constraints of Q decompose into k ASP-SAT programs $Q_1 = (W_1, S_1, D_1), \ldots, Q_k = (W_k, S_k, D_k)$ where $W_i = vars(S_i) \cup vars(D_i)$ s.t. for any distinct i, j in $1 \ldots k$, $W_i \cap W_j = \emptyset$. Let the remaining variables be: $\mathcal{V}_r = \mathcal{V} \backslash (W_1 \cup \ldots \cup W_k \cup vars(\theta))$ and let $\pi_1, \ldots, \pi_k$ be complete assignments over $W_1, \ldots, W_k$ respectively.*

1. *If $\pi_1, \ldots, \pi_k$ are stable models of $Q_1, \ldots, Q_k$ resp., then for any assignment θ_r over the remaining variables, $\theta \cup \pi_1 \cup \ldots \cup \pi_k \cup \theta_r$ is a stable model of P.*
2. *For a given assignment θ_r over remaining variables, if $\theta \cup \pi_1 \cup \ldots \cup \pi_k \cup \theta_r$ is a stable model of P, then π_i is a stable model of Q_i for each $i \in 1 \ldots k$.*

The first part of Theorem 1 shows that we can solve the justified residual program independently (as well as cache the result) and extend any of its stable model to a full stable model by assigning any value to the remaining variables of the original program. The second part of the theorem establishes that any full stable model of the original program is counted since it is an extension of the stable model of the residual program. The corollary tells us that if the justified residual program decomposes into disjoint programs, then we can solve each one of them independently, and multiply their counts to get the count for justified residual program.

Example 3. *In Example 2, the justified residual program has only two stable models: $\pi_1 = \{s, a, b, \neg t\}$ and $\pi_2 = \{t, a, b, \neg s\}$. It can be verified that the only stable assignments extending θ of P are $\theta \cup \pi_1 \cup \theta_{xyz}$ and $\theta \cup \pi_2 \cup \theta_{xyz}$ where θ_{xyz} is any assignment on the standard variables x, y, z. Therefore, the total number of stable models below θ is $2 \times 2^{|\{x,y,z\}|} = 16$.*

Now say we have another assignment $\theta' = \{a, b, c, d, u, e, \neg f, x\}$. It can be seen that it produces the same justified residual program as that produced by θ for which we know the stable model count is 2. Furthermore, the set of remaining variables is $\{y, z\}$. Therefore, the number of stable assignments below θ' is $2 \times 2^{|\{y,z\}|} = 8$.

In order to convert a model counter to a stable model counter, we can either modify its calculation of the residual program as suggested by Theorem 1, or, we can modify the actual program and use its *existing* calculation in a way that residual of the modified program correctly models the *justified* residual program. Let us describe one such approach and prove that it is correct. We post a copy of each founded variable and each rule such that the copy variable only becomes true when the corresponding founded variable is justified. More formally, for each founded variable v, we create a standard variable v', add the constraint $\neg v' \vee v$, and for each rule $a \leftarrow f_1 \wedge \ldots f_n \wedge sn_1 \wedge \ldots \wedge sn_m$ where each f_i is a positive founded literal and each sn_i is a standard or negative literal, we add the clause $a' \vee \neg f'_1 \vee \ldots \neg f'_n \vee \neg sn_1 \vee \ldots \vee \neg sn_m$. Most importantly, we do not allow search to take decisions on any of these introduced copy variables. Let this transformation of a program P be denoted by $copy(P)$.

We now show that it is correct to use the above approach for stable model counting. For the following discussion and results, let $P = (\mathcal{V}, R, C)$ be an ASP-SAT program, $copy(P) = (W, R, D)$ be denoted by Q. Let π, π_1, π_2 be assignments over W and θ, θ_1, θ_2 be their projections over non-copy variables ($\mathcal{V}$). Let $Q|_\pi$ (similarly for π_1, π_2) be a shorthand for $(vars(R|_\theta) \cup vars(D|_\theta), R|_\theta, D|_\theta)$. The results assume that assignments π, π_1, π_2 are closed under unit propagation and unfounded set propagation, i.e., both propagators have been run until fixpoint in the solver.

To prove the results, we define a function prj that takes the copy program Q and π and maps it to the justified residual program w.r.t. to the projection of that π on non-copy variables and then argue that $Q|_\pi$ correctly models the justified residual program. Formally, $prj(Q, \pi)$ is an ASP-SAT program $P' = (\mathcal{V}', R', C')$ constructed as follows. Add every constraint in $D|_\pi$ that does not have a copy variable in C'. For every constraint $v' \vee \neg f'_1 \vee \ldots \neg f'_n \vee \neg sn_1 \vee \ldots \vee \neg sn_m$ in $D|_\pi$, add the rule $v \leftarrow f_1 \wedge \ldots \wedge f_n \wedge sn_1 \wedge \ldots \wedge sn_m$ in R'. Let U be the set of founded variables v such that v is true but v' is unfixed in π. For every v in U, add the constraint v in C'. Define $\mathcal{V}'$ as variables of R' and C'. Proposition 3 proves that we cannot miss any stable model of the original program if we use the copy approach.

Proposition 3. *If π cannot be extended to any stable model of Q, then θ cannot be extended to any stable model of P.*

Theorem 4 establishes that we can safely use $Q|_\pi$ to emulate the justified residual program $P|^j_\theta$. Corollary 5 says that if we detect a stable model cube of $Q|_\pi$, then we also detect a stable model cube of the same size for the justified residual program. This corollary and Proposition 3 prove that the stable model count of the actual program is preserved.

Theorem 4. $P|^j_\theta = prj(Q, \pi)$.

Corollary 5. *If $Q|_\pi$ has no rules or constraints and there are k unfixed variables, then θ is a stable model cube of $P|^j_\theta$ of size 2^k.*

The next two corollaries prove that the copy approach can be used for caching dynamic decomposition respectively.

Corollary 6. *If $Q|_{\pi_1} = Q|_{\pi_2}$, then $P|^j_{\theta_1} = P|^j_{\theta_2}$.*

Corollary 7. *If $Q|_\pi$ decomposes into k disjoint components $Q_1, \ldots, Q_k$, then $P|^j_\theta$ decomposes into k disjoint components $P_1, \ldots, P_k$ such that $P_i = prj(Q_i, \pi_i)$ where π_i is projection of π on $vars(Q_i)$.*

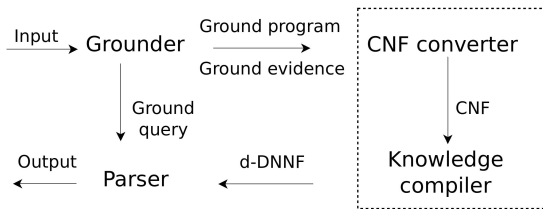

Figure 1: Execution of PROBLOG2

4 PROBLOG2 via Stable Model Counting

In this section, we describe how we apply stable model counting in the probabilistic logic programming solver PROBLOG2 (Fierens et al. 2013). A probabilistic logic program is a collection of mutually independent *random* variables each of which is annotated with a probability, *derived* variables, *evidence* constraints and rules for the derived variables. The *distribution semantics* (Sato 1995) says that for a given assignment over the random variables, the values of the derived variables is given by the well-founded model. Furthermore, the *weight* of that world is equal to the product of probabilities of values of the random variables. In our setting, it is useful to think of random variables, derived variables, evidence constraints, and rules as standard variables, founded variables, constraints and rules respectively. PROBLOG2 handles various inference tasks, but the focus of this paper is computing the marginal probability of query atoms given evidence constraints. The probability of a query atom is equal to the sum of weights of worlds where a query atom and evidence are satisfied divided by the sum of weights of worlds where the evidence is satisfied.

Figure 1 shows the execution of a PROBLOG2 program. The input is a non-ground probabilistic logic program which is given to the grounder that cleverly instantiates only parts of the program that are relevant to the query atoms, similar to how *magic set transformation* (Bancilhon et al. 1985) achieves the same goal in logic programming. The ground program and the evidence is then converted to CNF using the proof based encoding that we discussed earlier. This CNF is passed on to a *knowledge compiler* like DSHARP (Muise et al. 2012). DSHARP is an extension of SHARPSAT (Thurley 2006) where the DPLL-style search is recorded as d-DNNF. The d-DNNF produced by the knowledge compiler is given to the parser of PROBLOG2 along with the ground queries and probabilities of the random variables. The parser evaluates the probability of each query by crawling the d-DNNF as described in (Fierens et al. 2013).

Our contribution is in the components in the dotted box in Figure 1. We have implemented stable model counting by extending the propositional model counter SHARPSAT as described in the previous section. Since SHARPSAT is part of the knowledge compiler DSHARP, our extension of SHARPSAT automatically extends DSHARP to a stable model knowledge compiler. The CNF conversion component in PROBLOG2 chain is replaced by a simple processing of the ground program and evidence to our desired input format. In the first approach where the search is restricted to standard variables, the evidence needs to be passed on to our stable model counter which posts a nogood (the current assignment of standard variables) each time an evidence atom is violated. In approach given in Section 3.2, however, we post each evidence as a unit clause, much like PROBLOG2 does in its CNF conversion step. Including evidence in constraints in the second approach is safe since our residual program relies on the justified assignment only, and propagation on founded literals that makes them true due to constraints does not change that. Outside the dotted box in the figure, the rest of the PROBLOG2 logic remains the same.

5 Experiments

We compare the two approaches based on implementation of unfounded set detection as explained in Section 3 against the proof based encoding of PROBLOG2. We use two well-studied benchmarks: *SmokersFriends* (Fierens et al. 2011) problem and the graph reliability problem (*GraphRel*) (Arora and Barak 2009) with evidence constraints.

In both problems, the graph is probabilistic. In *GraphRel*, the nodes are associated with probabilities while in *SmokersFriends*, the edges have probabilities. Naturally, for n nodes, the number of random variables is in $O(n)$ and $O(n^2)$ for *GraphRel* and *SmokersFriends* respectively. Due to this, *GraphRel* has significantly more loops per random variables in the dependency graph which makes it more susceptible to the size problems of eager encoding. We refer to the fixed search approach of Section 3.1 as ASPROBLOGS and the proper integration of unfounded set detection through the use of copy variables of Section 3.2 as ASPROBLOG. All experiments were run on a machine running Ubuntu 12.04.1 LTS with 8 GB of physical memory and Intel(R) Core(TM) i7-2600 3.4 GHz processor.

Table 1 shows the comparison between PROBLOG2, ASPROBLOG and ASPROBLOGS on *GraphRel* on random directed graphs. The instance is specified by N, the number of nodes, and P, the probability of an edge between any two nodes. The solvers are compared on the following parameters: time in seconds (Time), number of variables and clauses in the input program of DSHARP (V and C resp.), number of decisions (D), average decision level of backtrack due to conflict or satisfaction (A), the size in megabytes of the d-DNNF produced by DSHARP (S), and for ASPROBLOG and ASPROBLOGS, the number of loops produced during the search (L). Each number in the table represents the median value of that parameter from 10 random instances of the size in the row. The median is only defined if there are at least (6) output values. A '—' represents memory exhaustion or a timeout of 5 minutes, whichever occurs first. A '—' in columns Time, D, A, L, S means that the solver ran out of memory but the grounding and encoding was done successfully, while a '—' in all columns of a solver means that it never finished encoding the problem. We show the comparison on three types of instances: small graphs with high density, medium graphs with high to medium density, and large graphs with low density.

Clearly ASPROBLOG and ASPROBLOGS are far more scalable than PROBLOG2. While PROBLOG2 requires less search (since it starts with all loop formulae encoded) the overhead of the eager encoding is prohibitive. For all solved

Instance		PROBLOG2						ASPROBLOG							ASPROBLOGS						
N	P	Time	V	C	D	A	S	Time	V	C	D	A	L	S	Time	V	C	D	A	L	S
10	0.5	11.33	2214	7065	**199**	**7.68**	1.21	**1.08**	72	226	233	8.88	**13**	**.057**	1.13	**60**	**171**	333	8.75	124	.10
11	0.5	115.75	6601	21899	**353**	**8.61**	7.62	**1.11**	86	283	382	9.76	**23**	**.10**	1.12	**73**	**216**	354	9.38	107	.10
12	0.5	—	16210	55244	—	—	—	**1.20**	101	348	**675**	10.81	**21**	**.19**	1.32	**87**	**267**	904	**10.47**	405	.28
13	0.5	—	59266	204293	—	—	—	**1.41**	117	414	**1395**	12.16	**44**	**.41**	2.61	**102**	**320**	2737	**11.33**	1272	1.28
15	0.5	—	—	—	—	—	—	**2.05**	142	514	**3705**	13.42	**59**	**1.23**	4.78	**125**	**398**	7542	**12.88**	2028	2.71
20	0.5	—	—	—	—	—	—	**31.82**	246	966	**83091**	18.37	**189**	**38.11**	82.21	**224**	**757**	143188	**18.31**	32945	62.02
25	0.25	—	—	—	—	—	—	**22.44**	225	800	**62871**	**18.70**	**231**	**27.23**	53.63	**198**	**620**	128534	19.55	41811	43.06
30	0.1	—	—	—	—	—	—	**3.71**	168	468	**7347**	**15.89**	**129**	**2.99**	13.22	**137**	**351**	43968	19.31	2833	10.40
31	0.1	—	37992	115934	—	—	—	**2.84**	171	473	**5054**	**15.06**	**52**	**2.23**	12.67	**140**	**356**	19585	17.53	1293	11.18
32	0.1	—	—	—	—	—	—	**7.93**	185	528	**17006**	**17.06**	**173**	**7.75**	35.97	**153**	**398**	108916	21.42	5405	32.10
33	0.1	—	—	—	—	—	—	**25.13**	191	533	**67929**	**18.49**	**343**	**31.06**	—	**157**	**403**	—	—	—	—
34	0.1	—	—	—	—	—	—	**12.97**	201	566	**33338**	**19.41**	**155**	**14.66**	112.27	**165**	**429**	324304	23.20	5502	124.21
35	0.1	—	—	—	—	—	—	**101.40**	222	663	**249512**	**21.78**	**1567**	**123.62**	—	**186**	**503**	—	—	—	—
36	0.1	—	—	—	—	—	—	**100.20**	228	683	**279273**	**21.41**	**1542**	**124.73**	—	**190**	**518**	—	—	—	—
37	0.1	—	—	—	—	—	—	**65.86**	227	659	**159056**	**20.55**	**658**	**77.57**	—	**188**	**499**	—	—	—	—
38	0.1	—	—	—	—	—	—	—	240	712	—	—	—	—	—	**200**	**540**	—	—	—	—

Table 1: Comparison of PROBLOG2, ASPROBLOG, and ASPROBLOGS on the Graph Reliability problem

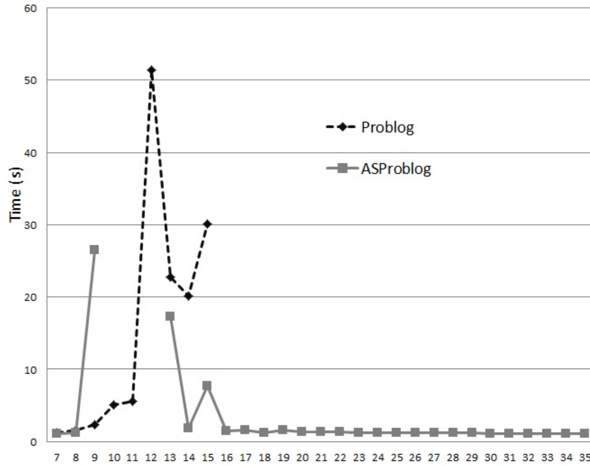

Figure 2: *SmokersFriends* with 31 random variables

instances, ASPROBLOG has the best running time and d-DNNF size, illustrating that the search restriction of ASPROBLOGS degrades performance significantly. While the encoding for ASPROBLOGS is always smallest, the encoding with copy variables and rules of ASPROBLOG is not significantly greater, and yields smaller search trees and fewer loop formulae. It is clearly the superior approach.

Figure 2 compares the performance of PROBLOG2 and ASPROBLOG on *SmokersFriends* when the number of random variables is fixed to 31 and the problem size is increased. In the problem description, there are two sets of random variables, the *stress* and the *influences* variables. The first one exists for each person in the graph, while the latter exists for every edge in the graph. In our setting, for an instance with n persons, the number of *influences* random variables is equal to $31 - n$. The rest of the *influences* variables are fixed to true or false at run time. For the smallest instances of sizes 7 and 8, PROBLOG2 and ASPROBLOG have similar performance. For instances 9 to 12, PROBLOG2 does better than ASPROBLOG where the latter cannot solve instances 11 and 12 due to memory exhaustion. The reason is that the complete encoding in PROBLOG2 propagates better and the extra unfounded set check at each node in the search tree in ASPROBLOG does not pay off. But as the number of people increases and the number of probabilistic edges becomes less, the problem becomes easier for ASPROBLOG but not for PROBLOG2. The reason is that by fixing the probabilistic edges, we are just left with n external rules, and many internal rules, making many founded variables logically equivalent to each other. In the last instance, the number of loop formulas required for the problem is only one! Our lazy approach benefits from this structure in the problem, while PROBLOG2 does not. Our experiments with the same range of instances but with number of random variables fixed to 33 and 35 show similar behaviour of PROBLOG2 and ASPROBLOG where initially, PROBLOG2 does better, followed by hard instances for both, and finally, ASPROBLOG detecting the structure and solving the last few instances in less than 2 seconds.

6 Conclusion

Stable model counting is required for reasoning about probabilistic logic programs with positive recursion in their rules. We demonstrate that the current approach of translating logic programs eagerly to propositional theories is not scalable because the translation explodes when there is a large number of recursive rules in the ground program. We give two methods to avoid this problem which enables reasoning about significantly bigger probabilistic logic programs.

References

Arora, S., and Barak, B. 2009. Computational complexity - a modern approach (Chapter: Complexity of Counting).

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. DPLL with Caching: A new algorithm for #SAT and Bayesian Inference. *Electronic Colloquium on Computational Complexity (ECCC)* 10(003).

Bancilhon, F.; Maier, D.; Sagiv, Y.; and Ullman, J. D. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, 1–15. ACM.

Fierens, D.; Van den Broeck, G.; Thon, I.; Gutmann, B.; and De Raedt, L. 2011. Inference in probabilistic logic programs using weighted CNF's. In Gagliardi Cozman, F., and Pfeffer, A., eds., *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI), Barcelona, Spain, 2011*, 211–220.

Fierens, D.; den Broeck, G. V.; Renkens, J.; Shterionov, D. S.; Gutmann, B.; Thon, I.; Janssens, G.; and Raedt, L. D. 2013. Inference and learning in probabilistic logic programs using weighted boolean formulas. *CoRR* abs/1304.6810.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 386. MIT Press.

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

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

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2008. Model counting.

Janhunen, T. 2004. Representing normal programs with clauses. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004*.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* 7(3):499–562.

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

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

Muise, C. J.; McIlraith, S. A.; Beck, J. C.; and Hsu, E. I. 2012. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Canadian Conference on AI*, 356–361.

Raedt, L. D., and Kimmig, A. 2013. Probabilistic programming concepts. In *Programming Languages, Computer Science, arXiv*.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining component caching and clause learning for effective model counting. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-2004)*.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *ICLP*, 715–729. MIT Press.

Thurley, M. 2006. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *SAT*, 424–429.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. 410–421.

Van Gelder, A.; Ross, K. A.; and Schlipf, J. S. 1988. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 221–230. ACM.

Vlasselaer, J.; Renkens, J.; den Broeck, G. V.; and Raedt, L. D. 2014. Compiling probabilistic logic programs into sentential decision diagrams. In *Workshop on Probabilistic Logic Programming (PLP), Vienna*.