A New Algorithm for P-log Inference Engine

Weijun Zhu

Texas Tech University

Abstract. P-log is a recently developed knowledge representation and reasoning language which allows to combine logical and probabilistic reasoning. Even though a naive implementation of P-log inference engine allows some real life applications, in many cases it is still too slow and memory consuming. In this paper we develop and implement a new algorithm for answering P-log queries. The new approach reduces the size of ground P-log program by only grounding rules which are related to the given query. By utilizing the unitary property of the program, this approach also avoids computing all possible worlds of the grounded program. We present some empirical test results of the naive implementation and this new implementation, and discuss the strength and weakness of both systems.

1 Introduction

The language P-log was introduced by [1] to represent logical and probabilistic information in the form of logic program. The semantics of the language is based on the semantics of Answer Set Prolog (ASP) [2]. The probabilistic information is also described by a set of rules whose semantics is built on the theory of Causal Bayesian Network [3].

P-log inherits it's logical expressibility and elaboration tolerance from ASP. Unlike Bayesian networks, which are acyclic graphs, P-log allows users to state recursive definitions without introducing extra atoms or variables. While Bayesian networks are widely used for probabilistic reasoning, data mining and machine learning, they are less suitable for representing some large and dynamic domains. A problem that may require thousands of nodes in its Bayesian networks representation can often be described by a P-log program with dozens of rules. Moreover any change of the size of the domain may cause huge changes on the corresponding graph while in P-log such change can be done simply by replacing some constants.

The first implementation of P-log can be downloaded from *www.cs.ttu.edu/ ∼ wezhu*. Some comparison of performance and ease of representation between this implementation and the system ACE (available at *reasoning.cs.ucla.edu/ace/*) can be found in [4]. The P-log system was successfully used to find best probabilistic diagnoses with some rather complicated systems [5].

This implementation is based on the following approach: the inputs, a P-log program Π and a query Q, are all translated into an ASP program $\tau(\Pi, Q)$ such that answer sets of $\tau(\Pi, Q)$ are in one-to-one correspondence to the possible

worlds of Π. In addition, all the probabilistic information in Π is also encoded by rules with some special atoms such that the measure of each possible world can be computed from the special atoms in the answer sets. We chose *smodels* [6], one of the most efficient answer set solver at that time, as the inference engine for computing answer sets. After all answer sets are computed by *smodels*, the probabilistic information is gathered for computing the probabilities of formulas in the query. In this paper, we refer to such implementation as *plog1.0* inference engine.

This approach suffers from some drawbacks. It is not unusual for a P-log program to have a huge number of possible worlds that make the system too slow. In fact, many queries to a P-log program can be answered by looking only at a smaller portion of the program, which in turn requires computing much smaller number of possible worlds. The procedure of encoding probabilistic information into logical rules significantly increase the size of each possible world. This affects the performance of the procedure of gathering probabilistic information. In some cases, this procedure can take more than 90% of the total running time.

In this paper, we introduce our new approach which partially grounds the input P-log program and uses our own answer set solver to avoid encoding probabilistic information to logical rules. In addition, this new algorithm utilizes the unitary property of our input program to improve the efficiency of the solver even more.

2 Syntax and Semantics of P-log

A probabilistic logic program (P-log program) Π consists of (i) a sorted signature, (ii) a declaration, (iii) a regular part, (iv) a set of random selection rules, (v) probabilistic information part and (vi) a set of observations and actions.

1. **Sorted Signature**: The sorted signature Σ of Π contains a set of constant symbols and term-building function symbols, which are used to form terms in the usual way. Additionally, the signature contains a collection of special function symbols called attributes. Technically, every P-log atom is of the form $a(\bar{t}) = y$, where a is an attribute, $\bar{t}$ is a vector of terms, and y is a term. A literal is an atomic statement $a(\bar{t}) = y$ or its negation, $a(\bar{t}) \neq y$. An extended literal l is a literal or a **not** l, where **not** is the default negation of Answer Set Prolog.
2. **Declaration**: The declaration of a P-log program is a collection of definitions of sorts, attributes and variables.
   - A sort c can be defined by explicitly listing its elements,
   $$c = \{x_1, \dots, x_n\} \quad (1)$$
   If c is a set of continuous integer numbes from m to n. It can be defined as,
   $$c = \{m..n\} \quad (2)$$

- Attributes can be declared by a statement of the form:

$$a : c_1 \times \ldots \times c_n \rightarrow c_0 \tag{3}$$

- Variables $V_1, \ldots V_n$ that can take values from elements of a sort c is declared as:

$$\#domain\ c(V_1; \ldots; V_n) \tag{4}$$

3. **Regular Part**: The regular part of a P-log program consists of a collection of rules r of Answer Set Prolog (without disjunction) formed using literals of Σ

$$l_0 \leftarrow l_1, \ldots, l_m, \mathbf{not}\ l_{m+1}, \ldots, \mathbf{not}\ l_n \tag{5}$$

4. **Random Selection Rules**: A random selection rule is a rule of the form

$$[r]\ random(a(\bar{t}) : \{X : p(X)\}) \leftarrow B \tag{6}$$

where r is a term used to name the rule and B is a collection of extended literals of Σ. Statement (6) says that if B holds, the value of $a(\bar{t})$ is selected at random from $\{X : p(X)\} \cap range(a)$, unless this value is fixed by a deliberate action.

5. **Probabilistic Information**: Information about probabilities of random attributes taking a particular value is specified by the following statement (*pr rules*)

$$pr_r(a(\bar{t}) = y|_c B) = v \tag{7}$$

where r is the term used to name the *pr rules*, B is a collections of extended literals, *pr* is a special symbol not belonging to Σ and v is a term built from arithmetic functions whose resulting value is a rational number between 0 and 1.

6. **Observations and Actions**: Observations and actions are statements of the respective forms: $obs(l)$ and $do(a(\bar{t}) = y)$, where l is a literal of Σ and $a(\bar{t}) = y$ is an atom of Σ.

Formal semantics of P-log are explained in [1]. In short, the P-log program is mapped to an A-Prolog program $\tau(\Pi)$ and random selection rules are translated into choice rules in *smodels*. The answer sets of $\tau(\Pi)$ are one-to-one correspondence to the possible worlds of Π. Probabilities are assigned to logically possible outcomes of random selctions. The probability value is determined either by *pr* ruels or by applying equal probability principles. The unnormalized probability, $\hat{\mu}(W)$, of a possible world W induced by Π is

$$\hat{\mu}_\Pi(W) = \prod_{a(\bar{t},y) \in W} P(W, a(\bar{t}) = y) \tag{8}$$

where $P(W, a(\bar{t}) = y)$ is the probability assigned to the random atom $a(\bar{t}) = y$ w.r.t. the possible world W.

A query Q to a P-log program Π of signature Σ has the form:

$$\{f_1, \ldots, f_k\} | obs(l_1), \ldots, obs(l_m), do(a_1(\bar{t}_1) = y_1), \ldots, do(a_n(\bar{t}_n) = y_n) \qquad (9)$$

where $f_1, \ldots, f_k$ are formulas of Σ, $l_1, \ldots, l_m$ are literals and $a_1(\bar{t}_1) = y_1, \ldots, a_n(\bar{t}_n) = y_n$ are atoms. By $formula(Q)$, we mean the set $\{f_1, \ldots, f_k\}$ and by Π_Q, we mean a P-log program consisting of *do* and *obs* statements in the query Q.

The answer to a query Q w.r.t. a P-log program Π of signature Σ is a set F of formulas such that

$$F = \arg \max_{f \in formula(Q)} P_{\Pi \cup \Pi_Q}(f) \qquad (10)$$

i.e., F is a subset of $formula(Q)$ such that for every element f of F, the probability $P_{\Pi \cup \Pi_Q}(f)$ has the largest value, or say f is one of the most likely formulas being true w.r.t. the program $\Pi \cup \Pi_Q$.

The algorithm described in this paper requires the probabilty distribution defined by the P-log programs follow the general probability principles. Furthermore, the P-log program should be causally ordered, which means no two random selection rules recursively depend on each other. In addition, we assume that the regular part of the P-log program is stratified, hence truth value of non-random attribute atoms in regular part of the P-log program can be derived through an iteration methods, described in [7]. In this paper, we call such P-log programs *strongly causally ordered unitary (scou)* P-log program. The formal definition of *scou* P-log program can be found at *www.cs.ttu.edu/ ∼ wezhu*.

Most probabilistic problems, including all the problems which can be modeled by Bayesian networks, can be represented as a *scou* P-log program. The restrictions on *scou* P-log prgram does not limit much on the usefulness of the algorithm described in this paper. Real life problems, including all the examples shown in this paper, should be natually represented by *scou* P-log programs. Furthermore, in most cases, a syntax based check can determine whether a given P-log program is a *scou* P-log program or not.

3 Grounding Algorithm

In Bayesian networks, conditional independence among variables is the key concept for concise representation of probabilistic models and for fast computation of probabilities of random events. The similar idea can be used for P-log inference engine, that is to find a subset of P-log program such that for those rules that are not included in this subset, the answer to the query is independent from them.

Our approach is based on the idea of *dependent set* of literals. To start, let us consider a ground *scou* P-log program, shown in Example 1, and our query is what is the probability of l being true.

Example 1. A ground *scou* P-log program

```
l, a, b, c, d, e : boolean.
1. l :- a, not b.
2. [a] random(a).
3. pr(a=true|c=true)=1/4.
4. [c] random(c).
5. [d] random(d).
6. [e] random(e).
```

Let S be a set of ground literals of a signature Σ. By $Term(S)$, we denote the set of ground attribute terms occurring in the literals from S. For example, if $S = \{a = true, b = false, c \neq true\}$, then $Term(S) = \{a, b, c\}$.

Definition 1. *[Dependent Set]*
Let Π be a ground P-log program with signature Σ and l be a ground literal of Σ. We define the **dependent set** *of l, written as $Dep_{\Pi}(l)$, as the minimal set of ground attribute terms from Σ satisfying the following conditions:*

- $Term(\{l\}) \subseteq Dep_{\Pi}(l)$;
- *For every regular rule $l_0 : -B$ of Π if $Term(\{l_0\}) \subseteq Dep_{\Pi}(l)$, then $Term(B) \subseteq Dep_{\Pi}(l)$);*
- *For a random selection rule of the form (4): if $a(\bar{t}) \in Dep_{\Pi}(l)$, then $Term(B \cup p(y)) \subseteq Dep_{\Pi}(l)$;*
- *For a pr rule in the form (5) if $a(\bar{t}) \in Dep_{\Pi}(l)$, then $Term(B) \subseteq Dep_{\Pi}(l)$.*

In the previous example, the dependent set of literal l is $\{l, a, b, c\}$. Only literals in the dependent set of literal l directly or indirectly affect the probability of l being true. Therefore, we can safely remove rules which contain no such literals. For example, rules 5 and 6 can be removed from the P-log program Π without changing the results of the query. Notice that with some extra rules, every formula f can be associated with a literal l such that l is true if and only if f is true. Therefore, the theorem we address here only consider the probability of a literal l and the proof can be found at *www.cs.ttu.edu/ ∼ wezhu*.

Theorem 1. *Let Π be a unitary and strongly causally ordered P-log program with no do and obs statements and Σ be its signature. Let l be a literal of Σ and α_l be the attribute term used to form l and $\Pi' = \{r | Term(Head(r)) \in Dep(\alpha_l), r \in \Pi\}$, then $prob_{\Pi'}(l) = prob_{\Pi}(l)$.*

For a non-ground P-log program, we can build the dependent set and ground the program at the same time. This is done by using a *head-to-body* approach, which is described in the Algorithm 1.

In Algorithm 1, we use the set S_{term} for storing ground terms which belong to the dependent set of l and use the set $S_{finished}$ for recording ground terms that have been processed. Inside the *while* loop, the ground program Π' is built by calling function $GroundRule(\Pi, a(\bar{t}))$. This function checks each rule of Π, if the rule can be grounded to rules whose head are formed by $a(\bar{t})$, then all such ground instances will be added to program Π'. Since some rules generated by

Input: a strongly causally ordered P-log program Π with signature Σ and a ground literal l of Σ.
Output: a ground P-log program Π' with signature Σ, such that $P_{\Pi}(l) = P_{\Pi'}(l)$
$S_{term} := Term(\{l\});$
$S_{finished} := \emptyset;$
Π' := declarations in Π;
while S_{term} *is not empty* **do**
  Let $a(\bar{t})$ be an element of S_{term};
  $\Pi' := \Pi' \cup GroundRule(\Pi, a(\bar{t}))$;
  Reduce the program Π';
  Update S_{term} and $S_{finished}$;
end
return Π'

Algorithm 1: *Ground*

this function may be useless (whose bodies are never satisfied by any possible worlds), we use a *reduction* function to eliminate any such rules. This *reduction* function is critical to help the algorithm keep unrelated ground terms out of the set S_{term} and to avoid grounding useless rules.

4 Computing possible worlds and probabilities

We use the following example to show that sometimes computing all possible worlds may not be necessary.

Example 2. [A scou P-log program Π]

```
a, b, f : boolean.
[a] random (a).
[b] random (b).
f :- a.
f :- b.
-f :- not f.
```

The probability of f being true w.r.t. Π can be computed by building a tree in which nodes, N, are labeled by sets, L_N, of literals in the signature of Π and links are labeled by values v $(0 \leq v \leq 1)$. The root of the tree consists of literals that are always true in all possible worlds of Π and the children of a node N are created as follows:

1. Find a random selection rule r such that r is ready to fire w.r.t. L_N (i.e., the body of r is satisfied by L_N and the probability of each possible outcome can be known by looking at corresponding *pr* rules or by applying equal probability principles). Suppose we pick the first random selection rule in Example 2 to fire, we will have two possible outcomes: a and $\neg a$.

2. We then derive as much as possible by using regular rules in the P-log program. As shown in Figure 1(a), the literal f is included in the left child of the root.

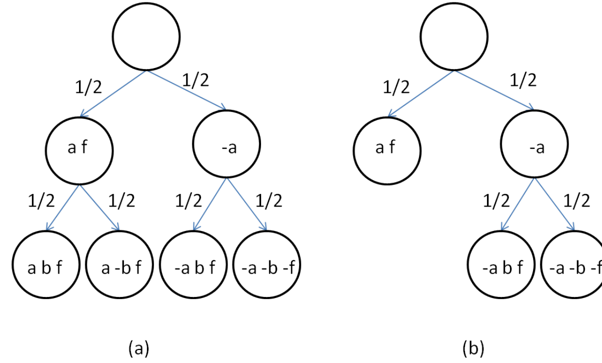

Fig. 1. Possible worlds of Example 2 and two different ways of computation of probability of f being true: (a) $P_{\Pi}(f) = P_{\Pi}(\{a, f, b\}) + P_{\Pi}(\{a, f, \neg b\}) + P_{\Pi}(\neg a, b, f)$; (b) $P_{\Pi}(f) = P_{\Pi}(\{a, f\}) + P_{\Pi}(\{\neg a, b, f\})$.

The weight of each node N, denoted by $weight(N)$, is the product of all the numbers that label the path from root to node N.

If a P-log program is a *scou* P-log program, then the tree built as above is a unitary tree in which the sum of the weights of all the leaf nodes descended from a node N is the weight of N.

To compute the probability of f being true w.r.t. the P-log program in Example 2, we can sum up all the weights of leaf nodes in which f is true. However, from Figure 1 we can see that we don't need to fire random selection rule b for the node $\{a, f\}$. Since the literal f is always true in both possible worlds ($\{a, f, b\}$ and $\{a, f, \neg b\}$) extended from $\{a, f\}$, we can compute the probability of f being true by summing up $P_{\Pi}(\{a, f\})$ and $P_{\Pi}(\{\neg a, b, f\})$, as shown in Fig. 1(b).

With *obs* and *do* statements in the program which play roles of eliminating possible worlds of a P-log program, the tree we created above is no longer a unitary tree. Hence we need some special treatments to make our tactics work. The goal is to look for nodes N in the tree such that the subtree rooted at N is a unitary tree and the truth value of a formula f is known by N. We call such a node as *a partial possible world that knows* f. Let Φ be the set of all these nodes, we define ϕ be a subset of Φ as follows: For any node N and its parent N', $N \in \phi$ if and only if $N' \notin \Phi$.

The set ϕ is called the set of smallest partial possible worlds that knows f. For some P-log programs Π, the size of ϕ can be much smaller than the number of possible worlds. The following theorem says that it is sufficient to compute

the probability of a formula f by summing up the weights of nodes which belong to ϕ.

Theorem 2. *Let Π be a ground scou P-log program and f be a formula. Let ϕ_f be the collection of all the smallest partial possible worlds of Π that know f. Then the probability of f being true w.r.t. Π can be computed by following equation:*

$$P_\Pi(f) = \sum_{I \in \phi_f \text{ and } I \vdash f} weight(I) \tag{11}$$

5 Experimental Results

We conducted our experiments on a machine built with 1.60GHz CPU and 2GB memory. The operating system is ubuntu system (version 10.10). We name the new P-log inference engine, based on the approach described in this paper, as *plog*2.0 and call the old one as *plog*1.0. We compare the performance of *plog*2.0 against *plog*1.0 to see how problems from different domains affect the efficiency of those two P-log inference engines.

We record the running times of *smodels* in *plog*1.0 and report them under the column *Sm* in our tables. The time used for retrieving probability information from output of *smodels* and computing the probability of formulas is recorded under the column *Retr*. The running times of grounding software (*lparse*) is not listed in the tables as they are normally insignificant. For the new system *plog*2.0, grounding time is presented under the column *Ground* and the time used for computing partial possible worlds and probability of formulas is shown under the column *Solv*. The total running times, shown under the column *Total* is measured by the *time command* in Linux system. In our tables, all running times are in seconds.

We tested our systems on a large pool of domains. The four domains we presented in this paper are the most representative examples that address the characters of both systems. Except the *grid* domain, all domains came from other relevant literatures. The detailed descriptions of each domain and the P-log program of each instances can be downloaded at *www.cs.ttu.edu/ $\sim$ wezhu*.

	plog1.0			plog2.0		
Inst	Sm	Retr	Total	Ground	Solv	Total
bld(7,1)	3.73	53.58	59.71	0.02	0.01	0.10
bld(7,2)	2.01	29.03	33.90	0.02	0.05	0.09
bld(7,3)	1.13	15.73	17.59	0.03	0.37	0.41
bld(7,4)	0.57	8.58	9.64	0.03	1.97	2.03
bld(8,4)	6.53	94.76	105.19	0.04	14.91	15.09

Fig. 2. The blood type domain

In Figure 2, we present our results of running *plog*2.0 and *plog*1.0 on problems in *blood type* domain [8]. The problem is to compute the probability of one's blood type w.r.t. the ABO blood type system given some blood type information of his relatives. In each instance $bld(m, n)$, we have total m individuals in the domain, of which n persons are relatives to the person P we are interested in. Notice that the blood types of individuals from other families have no influence on the result of our query. Those individuals can be dropped in the computation of the query. From the table, we can see that *plog*2.0 performs much better than *plog*1.0 does on all the instances we have tested. For many instances, *plog*2.0 is more than 10 times faster than the old one. The table also shows that the procedure of retrieval probabilistic information from answer sets takes about 90% of the total running time of old P-log inference engine. In *plog*2.0, the grounding time normally is small and insignificant comparing to the *solving* time.

The result is not surprising. While both systems have the same ungrounded P-log program as input, the sizes of the ground programs produced by both systems are different. The system *plog*1.0 grounds the whole program as *plog*2.0 only grounds part of it. The system *plog*2.0 will produce a ground program which only contains n individuals, adding new individual who is not a family member of P to this domain has no effect on how many possible worlds will be computed by *plog*2.0. On the other hand, adding any individual increases the number of possible worlds in an exponential rate for old *plog*1.0 system.

	plog1.0			plog2.0		
Inst	Sm	Retr	Total	Ground	Solv	Total
grid_3_4	0.09	1.09	1.34	0.01	0.04	0.07
grid_2_7	0.38	5.35	6.13	0.01	0.06	0.08
grid_3_5	0.70	11.03	12.48	0.02	0.26	0.30
grid_4_4	1.54	23.40	26.50	0.02	0.56	0.59
grid_3_6	6.48	106.06	120.06	0.03	1.69	1.79
grid_4_5	-	-	T.O.	0.03	8.08	8.12

Fig. 3. The grid domain.

In Figure 3, we show the results of instances of the *grid* domain. A grid instance, *grid_m_n*, consists of $m \times n$ nodes. Each node N in the grid receives information from its left node and the node above and passes information to its right node and the node below. If N is a faulty node, then it cannot pass information to its destination. Nodes can randomly be faulty or not. We are interested in the probability of information being successfully passed from the node in the up-left conner to the node at the bottom-right conner. The table shows that the *plog*2.0 performs significantly better than *plog*1.0 does. The total time grows exponentially with respect to the number of total nodes in the domain for both engines, but *plog*2.0 is about 50 times faster than *plog*1.0 on average.

Since each node has two possible outcomes: the node is faulty or not, the number of possible worlds of an instance $grid_m_n$ is $2^{m \times n}$. However, the set of all the smallest partial possible worlds that knows the formula (here, the formula is the literal $MessageReceived$ which means the node in the bottom-right conner can receive the information sent from the node at the up-left conner) is much smaller. For example, if the source node is faulty then we can derive that the literal $MessageReceived$ cannot be true regardless the status of other nodes. We find that the idea of smallest partial possible worlds that knows formula is specially useful when dealing with diagnosis problems where often a single faulty component may explain the unexpected observations.

	plog1.0			plog2.0		
Inst	Sm	Retr	Total	Ground	Solv	Total
B10_2	0.05	0.02	0.17	0.24	0.03	0.27
B10_3	0.11	0.07	0.28	0.24	0.09	0.34
B15_1	0.09	0.01	0.36	1.10	0.04	1.15
B15_2	0.34	0.09	0.71	1.09	0.25	1.36
B15_3	1.11	0.45	1.88	1.13	1.04	2.19
B20_1	0.42	0.03	1.01	3.94	0.14	4.11
B20_2	1.99	0.28	2.90	4.01	1.25	5.29
B20_3	6.62	0.53	7.86	4.04	7.32	11.40

Fig. 4. The block map domain.

We present the results of running instances of the *block map* domain [9] with *plog*2.0 and *plog*1.0 in Figure 4. We can see that the new inference engine takes more total time than those reported by *plog*1.0. Unlike previous examples, where grounding time does not play important role on the overall performance, the grounding time for the *block map* domain significantly affects the overall performance of *plog*2.0. As shown in the table, in the worst case, the grounding time could take over 90% of overall running time. The solving time of *plog*2.0 is comparable to the running time of *smodels*. Except the last instance, the solving time of *plog*2.0 is slightly smaller than the running time reported by *smodels*.

Both engines produce the same size of ground programs and compute same number of possible worlds. Therefore, the method we described in this paper does not help. The head-to-body approach in the grounding algorithm takes too many extra considerations with no gains on the resulting ground program. We believe this is the main reason why the overall performance of new system is worse than the old one.

The results of running instances of *poker* domain [10] are shown in Figure 5. An instance $P3_12$ means we randomly pick 3 cards from 12 cards deck. We are interested in the probability of having a single pair in hands. Again, the new system *plog*2.0 performs better than the old system *plog*1.0. On average, *plog*2.0 takes about 1/3 of the total running time of *plog*1.0 does.

	plog1.0			plog2.0		
Inst	Sm	Retr	Total	Ground	Solv	Total
P3_12	0.02	0.07	0.25	0.05	0.02	0.08
P3_16	0.05	0.21	0.37	0.09	0.07	0.17
P3_20	0.11	0.54	0.79	0.14	0.15	0.30
P5_12	0.07	0.39	0.58	0.12	0.15	0.29
P5_16	0.41	2.78	3.54	0.22	1.00	1.24
P5_20	1.67	12.60	14.90	0.33	4.14	4.51

Fig. 5. The poker domain

Similar to the *block map* domain, both engines essentially produce the same size of ground programs and compute the same size of possible worlds for instances in this domain. Comparing to the block map example, the differences are: in the poker domain, the size of the ground program is not very large. With much less logical rules in the poker domain, each possible world is much easier to compute. Meanwhile, the large number of possible worlds makes the retrieval procedure become a bottle neck of the performance of *plog*1.0. It takes more than 80% of the total computing time in several instances. The smaller number of logic rules make the difference between the *solving* time of *plog*2.0 and *smodels* running time of *plog*1.0 become ignorable.

Overall, *plog*2.0 performs better than *plog*1.0 in many types of domains. The improvement can be huge when partially grounding and computing partial possible worlds are effective. Even for domains that have no such advantage can be taken by *plog*2.0, the new system will still perform better when the total number of possible worlds is large and the size of the program is small.

6 Related Work

We surveyed several works related to the inference engine designed for combining logic and probabilistic reasoning. Some research has been conducted to explore the benefits of utilizing logic information in Bayesian networks to improve the performance of exact inference engines for probabilistic reasoning. [11] shows computing with Bayesian networks can be reduced to CNF model counting problems. This approach uses state-of-the-art SAT and model counting engines, dynamic variable branching heuristics and component caching. The results show that when many of the dependencies between variables are entirely or partially deterministic, the efficient logical machine underlying model counting programs stands a good chance of quickly reducing the problem into small subproblems. [9] described an inference system for relational Bayesian networks. The system compiles propositional instances of these models into arithmetic circuits. It exploits determinism and local structure in the Bayesian networks and expand the scales of Bayesian networks that can be handled efficiently by exact inference algorithms.

The closest work to this paper is another implementation of P-log [10], which was developed by using XASP for computing possible worlds and using XSB for retrieving probability information from possible worlds. Because their implementation is also translation based, it has similar disadvantages as our old P-log system. We plan to perform a comparison between *plog*2.0 and their implementation in the near future.

7 Conclusion and Future Work

This paper described the inference engine *plog*2.0. We conclude that our new system *plog*2.0 is more efficient than the old system *plog*1.0. In a summary, we list the major improvements of this new system as follows:

1. The new algorithm selectively grounds the input program with respect to the specific query given by the user. When a smaller ground P-log program is produced, often it will result in a much smaller set of possible worlds and faster computation.
2. The old P-log system involves complicated decoding of probability information from the output of an answer set solver and some heavy IO operations. The new algorithm combines construction of possible worlds and measures their probability at the same time. This may save significant computation time comparing to old P-log system where measuring possible worlds are done after all the possible worlds are built.
3. The new algorithm takes advantage of unitary properties of the input program. Instead of building the set of all possible worlds, it builds the set of smallest partial possible worlds that know the formula. The later set usually is much smaller and easier to compute. The *grid* domain and similar examples have shown that such method can be very effective in reducing the computation time of P-log systems.

We list our future work as follows:

- It is very common that once a P-log program is fixed, user may propose many different queries to the program. Our implementation so far has not taken advantage of this usage patterns. With proper cache techniques introduced, we should be able to avoid repeated computations for different queries. One of our future work is to add this feature so that a sequence of queries can be answered more quickly.
- If the ground program still contains a lot of random selection rules. Evaluating the probability of a formula may become infeasible. Many sophisticated algorithms, such as recursive conditioning [12], allow to compute probabilities with respect to a large scale of Bayesian networks. We are interested in how to combine these techniques with our algorithms so we may be able to solve problems with large number of random selection rules in the program.
- The ground time of the new system is comparatively small in many domains we have tested. However in some other problems, when the ground program

is large, it become the bottle neck of the overall performance of P-log system. We expect some overhead of our grounding algorithm comparing to other systems, say *lparse*. However, we believe a better design of grounding algorithm may help reducing the grounding time on large programs.

References

[1] Baral, C. and Gelfond, M. and Rushton, N.: Probabilistic Reasoning with Answer Sets. Theory and Practice of Logic Programming (2009) 57–144.

[2] Gelfond, M. and Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing (1991) Vol. 9. 365–385.

[3] Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kauffmann Publishers, Inc. (1988)

[4] Gelfond, M. and Rushton, N. and Zhu, W.: Combining Logical and Probabilistic Reseasoning. AAAI Spring Symposium (2006) 50–55.

[5] Nogueira, N. and Balduccini, M. and Gelfond, M. and Watson, R. and Barry, M.: An A-Prolog decision support system for the Space Shuttle. In PADL 2001 (2000) 169–183.

[6] Niemelae, I. and Simons, P.: Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. Lecture Notes in Computer Science (1997) Vol. 1265 420–429.

[7] Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving Cambridge University Press. (2003)

[8] Friedman, N. and Getoor, L. and Koller, D. and Pfeffer, A.: Learning Probabilistic Relation Models. AAAI (1999) 1300–1307.

[9] Chavira, M. and Darwiche, A. and Jeager, M.: Compiling Relational Bayesion Networks for Exact Inference. International Journal of Approximate Reasoning (2006). Vol. 42. 4–20.

[10] Anh, H.T. and Ramli, C.D. and Damasio, C.V.: An Implementation of Extended P-Log Using XASP. In ICLP '08: Proceedings of the 24th International Coference on Logic Programming, pages 739-743, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Sang, T. and Beame, P. and Kautz, H.: Solving Bayesian networkss by Weighted Model Counting. AAAI (2005). 475–482.

[12] Darwiche, A.: Recursive Conditioning. Artificial Intelligence (2001). VOL. 126. 5–41