

Prolog for First-Order Bayesian Networks: A Meta-interpreter Approach

Hendrik Blockeel

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
`hendrik.blockeel@cs.kuleuven.ac.be`

Abstract. Several extensions of bayesian belief networks to the first order logic or relational framework have been proposed. Many of these have in common that they are embedded in some kind of probabilistic or other extension of logic programming. In this paper we take yet another approach, which could be called a meta-interpreter approach. We discuss the representation of “first order” bayesian belief networks in standard Prolog. The representation formalism we propose is very simple, does not make use of any extensions to logic programming, allows inference using a simple interpreter written in Prolog, and the formalism has an expressiveness similar to other relational variants of bayesian belief networks. Due to the simplicity of the framework, we believe it may be a suitable reference point to compare other approaches to.

1 Introduction

The integration of probabilistic reasoning with first order logic is of increasing interest in the knowledge representation community (e.g., [5, 14, 11]), and also in machine learning and data mining, increasing attention is being paid to probabilistic models. A particular point of interest is formed by combinations of relational learning (including inductive logic programming) and probabilistic learning methods (e.g., [15]).

Bayesian belief networks are a popular representation formalism for representing probabilistic knowledge. The formalism has been extended to relational contexts in several ways [3, 6, 8, 2]. The relationship between all these different extensions is still under investigation.

In this paper we introduce a new approach, and a new perspective in which existing approaches can be viewed. It is a meta-interpreter approach; that is, bayesian networks are described in standard Prolog, and a meta-interpreter (also programmed in Prolog) is used to perform inference in them. This approach is quite similar to Bratko’s approach of representing bayesian networks in Prolog and programming inference for them [1]. It extends it, in that a typical chunk of knowledge describes a set of bayesian networks (or: a “first order” bayesian network) rather than a single one. The networks in such a set share certain similarities, at different possible levels. Consequently, certain types of inference can happen at the level of the whole set of networks instead of its elements.

This paper presents work in progress. We discuss mainly the representation of first order bayesian networks; inference and learning are only touched upon, although it is highly unlikely that any of these would be problematic (given the rather close relationship with existing approaches where concrete algorithms have been proposed). We compare our approach with several existing approaches.

In Section 2, we give some preliminaries and motivate our approach. In Section 3 we present a proposal for representing first order bayesian belief networks in Prolog. In Section 4 we briefly discuss inference and learning. In Section 5 we compare with a number of related approaches, and in Section 6 we conclude.

2 Preliminaries and Motivation

We take a look at bayesian networks from the point of view of the knowledge they represent. A bayesian network is in fact a compact representation of a joint distribution over a set of random variables $X_i, i = 1 \dots n$. For ease of discussion we focus on variables with finite domains D_i . The joint distribution represents all probabilities $P(X_1 = x_1, \dots, X_n = x_n)$ with $x_i \in D_i$. There are $\prod_i |D_i|$ such probabilities, hence the representation of the joint distribution in this form is exponential in the number of variables.

In a bayesian network, additional knowledge is represented about the structure of the joint distribution. More specifically, certain independence assumptions are made; because of these assumptions, the joint distribution can be computed as a product of "local" distributions, that is, marginal and conditional distributions that describe probabilities for all combinations of a subset of variables. The size of these representations is exponential in the number of variables they contain, which is typically much lower than the total number of variables.

A bayesian network consists of a directed graph that represents information on dependencies, as follows. Each node of the graph represents a probabilistic variable. We say that x is a parent of y if there is a directed edge (x, y) in the graph. The knowledge represented by a bayesian network is that each variable is statistically independent from its non-descendants, given its parents.

For instance (example taken from Russell and Norvig [12]), we could have random variables `earthquake`, `burglary`, `alarm`, `john_calls`, `mary_calls`. The graph in Figure 1 represents the knowledge that whether the alarm goes off depends on whether there is an earthquake and also on whether there is a burglary; that burglaries and earthquakes happen independently of each other; and that, given a value for `alarm`, the reaction of the neighbours John and Mary is independent (i.e., any correlation between John's and Mary's calling is explained solely by the influence `alarm` has on them, there is no direct influence of the neighbours on each other).

As several authors have mentioned (e.g., Haddawy [4]), the knowledge expressed by this dependency graph can be expressed in propositional logic. Now in many practical situations we may have knowledge that is of a slightly more complex kind. For instance, consider the knowledge that whether a person carries some gene is influenced by whether the person's father and mother carry

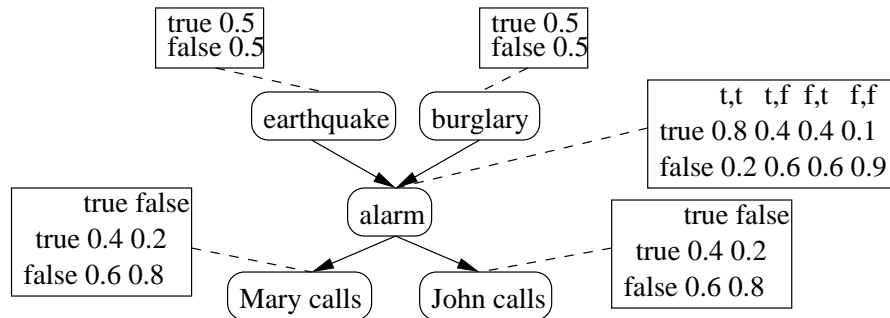


Fig. 1. A simple bayesian network.

the gene. The statement is supposed to hold for all persons and genes in a given universe, whatever that universe is. We can easily express this knowledge in first order logic. In propositional logic, we would have to know the universe in advance, and the dependency has to be stated for each person and gene combination separately; and the same holds for traditional bayesian networks. Our knowledge in this case really describes a set of bayesian networks with similar properties, rather than a single network. The question arises whether we can represent this knowledge in such a way that inference (both deductive and inductive) at the level of both the set and its elements becomes possible. Several existing approaches already demonstrate that this is possible [9, 8, 7, 13].

The difference between the approach we will propose here, and these other approaches, can be summarized as follows. Existing approaches up till now used either a procedural language to model bayesian networks, or a declarative language that is typically an extension of first order logic. Typical for the latter approaches is that the probabilistic reasoning is somehow embedded in the standard inference mechanism of the modelling language. Our approach is closest to the latter, in the sense that it is also declarative; but now, the code *describes* the network, whereas in other approaches the code *is* the network. We use a meta-interpreter to interpret the code, whereas in other approaches the interpretation is part of the inference engine underlying the program.

Our approach has the advantage that one does not need to be familiar with probabilistic extensions of first order logic, constraint logic programming, etc., nor with details of their execution mechanism. A basic understanding of standard Prolog programs is sufficient. We believe this may make the approach suitable as a reference point to compare other approaches to (in a similar way as meta-interpreters in declarative languages can be instructive regarding how inference is performed). Such reference points have been proposed previously, but currently none of them seem to be generally accepted.

3 Representing Bayesian Networks in Prolog

Bratko [1] offers an elegant method for representing standard (propositional) bayesian networks in Prolog. The approach we follow here uses a somewhat different syntax but is not essentially different; it mainly makes some parts of our discussion a bit simpler. We will use the term “first order bayesian network” to refer to a Prolog definition that represents a set of traditional bayesian networks.

We use ground terms to denote random (i.e., stochastic) variables.¹ We define a predicate *depends/3* with the following semantics. Let x be a random variable, l a list of random variables, and t a probability table. *depends*(x, l, t) is true if and only if in the first order bayesian network that is being modelled, x has the variables in l as parents and t is the conditional probability table that lists $P(x|l)$.

We use *input_node*(x, t) as syntactic sugar for *depends*($x, [], t$). Actually, t is in this case represented slightly differently (it is an unconditional distribution rather than a conditional one), but the actual representation of t is an implementation detail and not really relevant here.

Example 1. We again refer to Figure 1. The representation of the bayesian network in Figure 1, using the above predicates, is as follows:

```
depends(alarm, [earthquake, burglary],
  [ [ [t,t], [t,f], [f,t], [f,f]],
    [t, 0.8, 0.4, 0.4, 0.1],
    [f, 0.2, 0.6, 0.6, 0.9] ]).
depends(marycalls, [alarm],
  [ [ [t], [f]],
    [t, 0.4, 0.2],
    [f, 0.6, 0.8] ]).
depends(johncalls, [alarm],
  [ [ [t], [f]],
    [t, 0.4, 0.2],
    [f, 0.6, 0.8] ]).
input_node(earthquake, [t:0.5,f:0.5]).
input_node(burglary, [t:0.5,f:0.5]).
```

The mapping between the Prolog program and the graphical representation with probability tables is obvious. Note that no logical variables were used in the above code; this implies that the above Prolog program represents a traditional bayesian network. By introducing logical variables, more general kinds of knowledge can be represented.

Example 2. This example is inspired by the horse breeding farm example also used by Kersting and De Raedt [8]. The Prolog program

¹ This is a first important difference with other logic-based approaches, e.g., *CLP(BN)* [13] uses skolem constants, and *BLPs* [8] use literals, to denote stochastic variables.

```

depends(has_gene(X), [has_gene(Y), has_gene(Z)],
  [ [ [t,t], [t,f], [f,t], [f,f]],
    [t, 0.8, 0.4, 0.4, 0.1],
    [f, 0.2, 0.6, 0.6, 0.9] ])
:- father(Y,X), mother(Z,X).
input_node(has_gene(X), [t:0.5, f:0.5])
:- (father(Y,X), mother(Z,X) -> fail; true).

```

represents the following information: “The probability that a horse carries a certain gene depends on its father and mother having that gene. If information on its parents is incomplete, we estimate the probability of it carrying the gene as 0.5 ”.

This program can be seen as a bayesian network generator. By adding the information

```

father(apollo, chloe).
mother(blaze, chloe).
father(dexter, flash).
mother(ember, flash).
father(flash, galaxy).
mother(chloe, galaxy).

```

the network structure in Figure 2 is generated (for simplicity we write only names in the figure, instead of `has_gene(name)`).

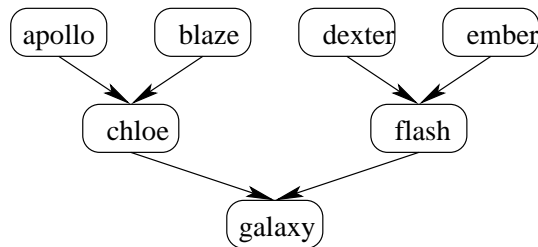


Fig. 2. A “horse farm” bayesian network.

Note that the Prolog program is more structured than the bayesian network. In a traditional bayesian network, a separate conditional probability table is associated with each node. In the Prolog program, such a table is associated with a clause, which defines a set of nodes. We just define which kind of nodes have which kind of probability table. For instance, `has_gene(chloe)` and `has_gene(flash)` share the same conditional probability table (CPT).

We could further generalize this piece of knowledge by defining the bayesian network for different genes, allowing for probability tables that vary with the type of gene but not with the horses.

```

depends(has_gene(X,G), [has_gene(Y,G),has_gene(Z,G)], T)
  :- father(Y,X), mother(Z,X), cpt(G, T).
input_node(has_gene(X,G), T)
  :- (father(Y,X), mother(Z,X) -> fail; true), distr(G, T).

cpt(gene_a, [ [ [t,t], [t,f], [f,t], [f,f]],
              [t, 0.8, 0.4, 0.4, 0.2],
              [f, 0.2, 0.6, 0.6, 0.8] ]]).
distr(gene_a, [t:0.5, f:0.5]).

```

We can define several levels of similarity for the elements of a set of bayesian networks: they can all have the same structure and the same probability table; the same structure but different probability tables; or even different structures. The latter is the case, e.g., if the success of a student for some study program depends on the courses taken, and the number of such courses may be variable. Clearly in such cases the computation of the CPT becomes more complex. Nevertheless our representation formalism can easily represent such cases:

```

depends(succeeds(X), L, T) :-
  findall(Course, registered(X, Course), L),
  cpt_for_succeeds(L, T).

cpt_for_succeeds(L, T) :- ...
  % would typically depend on some weighted average

```

Obviously, not any possible definition of `depends` is consistent with its interpretation (defining a bayesian network). For instance, the definition of `depends` must be such that for each node exactly one list of parents and one CPT is generated; also, the generated directed graph must be acyclic. For now we are assuming it is the responsibility of the programmer of the bayesian network to ensure this.

4 Inference and Learning

We briefly discuss inference in first order bayesian networks and how they can be learned from data. This discussion is very brief and no concrete methods are proposed; however, our approach is sufficiently close to some other approaches (see next section) to believe that the methods proposed there can easily be adapted to our framework.

Inference in a first order bayesian network is simple, as it really represents just a large bayesian network. A Prolog meta-interpreter for inference in bayesian networks is easy to write, see, e.g., Bratko's program [1] which handles propositional networks. A simplified meta-interpreter for first order bayesian networks is provided in the appendix; although it performs a limited kind of inference only, it may be instructive regarding the simplicity of this approach.

Learning the parameters of first order bayesian networks is as easy as for propositional networks. For instance, consider the case where CPT entries are computed from counts (e.g., $P(X = x|Y = y) = |D_{X=x,Y=y}|/|D_{Y=y}|$, with D referring to some data set and $|D_c|$ denoting the number of cases fulfilling condition c). The CPT of a clause, which may describe multiple nodes in the network, can be computed by defining as “stochastic variables” not the original stochastic variables but sets of them. That is, instead of having a variable $x(a)$ and $x(b)$, for each of which we could count how many times it takes a certain value, we just have a variable x for which these counts are the sum of the separate counts for $x(a)$ and $x(b)$. In the case of “parameterized” CPT’s (as for the `gene_a` example), the stochastic variables are just generalized a bit less; that is, `has_gene(apollo, gene_a)`, `has_gene(blaze, gene_a)`, etc., are generalized to a variable `has_gene(gene_a)` but not to a single variable `has_gene`.

More precisely, the procedure detailed in Figure 3 can be followed. Basically, the procedure looks at each `depends` clause separately. For each such clause, it determines the logical variables in its body that uniquely determine the CPT. Then for each possible value combination v for these variables, all observations for *Node* and *Parents* are taken into account to compute the CPT associated with v .

```

For each clause depends(Node,Parents,CPT) :- Body:
   $V :=$  all logical variables in Body that determine the CPT
   $S :=$  all  $(Node, Parent, V)$  instantiations generated by Body
  for each different value  $v$  for  $V$  occurring in  $S$ :
     $S_v := \{(N, P, V) \in S | V = v\}$ 
    compute  $CPT_v$  from  $S_v$ 

```

Fig. 3. Learning the CPT’s associated with a clause.

Example 3. Consider the clause

```

depends(has_gene(X,G), [has_gene(Y,G),has_gene(Z,G)], T)
  :- father(Y,X), mother(Z,X), cpt(G, T).

```

According to this clause, the CPT associated with `has_gene(X,G)` depends on G , but not on X . Therefore, a different CPT should be constructed for each different value of G that occurs (`gene_a, ...`). The CPT for `gene_a` will be constructed from all observed values of `has_gene(..., gene_a)`.

Learning the structure of a first order bayesian network boils down to learning the conditional clauses of the networks. These can be learned using ILP approaches, in exactly the same way as proposed by Kersting and De Raedt [7] and Santos Costa et al. [13].

5 Related Work

5.1 Bayesian Logic Programs

Kersting and De Raedt’s [8] bayesian logic programs formed in a sense the inspiration for this work. Part of their motivation for introducing BLP’s is that they generalize over bayesian networks in the same way first order logic generalizes over propositional logic. In addition, however, BLP’s form a kind of unifying framework over Prolog and bayesian networks.

Instead of unifying the logical and probabilistic level, our approach explicitly aims to keep them on several levels. In our opinion this clean separation simplifies the semantics of the knowledge chunks that are represented.

A specific example that illustrates some of the complexity that arises when the logical and probabilistic levels are unified, is the following BLP clause:

```
has_gene(X) | father(Y,X), mother(Z,X), has_gene(Y), has_gene(Z)
```

Kersting and De Raedt define the bayesian networks that such a clause gives rise to, as follows: for all ground instantiations of the clause, a bayesian network exists in which the instantiated head of the clause has as parents the instantiated body literals of the clause. Note that this puts the `father` literal and the `has_gene` literal at the same level, whereas in our approach the `father` literal defines the *structure* of the bayesian network and hence is at the logical level, whereas the `has_gene` literals correspond to random variables, and hence are terms in our approach.

It is easy to see that the bayesian networks generated by BLP’s do not generate the propositional bayesian networks that would normally be used if one hand-coded them. Moreover, the semantics of the resulting bayesian networks are quite complex. Consider the instantiation

```
has_gene(a) | father(b,a), mother(c,a), has_gene(b), has_gene(c)
```

and consider it as a bayesian network. The network is drawn in Figure 4, together with a similar network that would be generated by our approach. The bayesian network expresses that `has_gene(a)` is dependent on `has_gene(b)` (and its other parents). However, what is really the case is that `has_gene(a)` depends on `has_gene(b)` *only if* `father(b,a)` has the value *true*. Indeed, if `father(b,a)` has the value *false*, no meaningful value can be filled in for the conditional probability of `has_gene(a)` given `has_gene(b)`. That is, the conditional probability tables in this approach contain empty fields. Apparently Kersting and De Raedt’s implementation handles this situation correctly by “compiling away” the variables that in fact indicate structural properties. Yet, their syntax does not hint at these complexities, and thus seems deceptively simple. In the approach we propose, the fact that `has_gene(a)` depends on `has_gene(b)` only if `father(b,a)` has the value *true*, is exactly what is represented by our Prolog clauses.

It is not obvious how inference should be performed in BLPs if the “structural” variables are not compiled away. In such a situation, the BLP approach

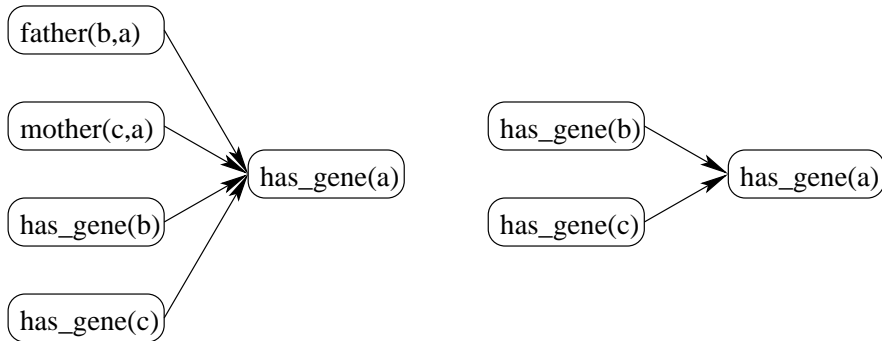


Fig. 4. Left: Bayesian network represented by the instantiation of a bayesian logic program. Right: Bayesian network generated by our approach.

can also be interpreted as follows: by placing the structural and probabilistic information at the same level, one gains the ability to learn the structure of the bayesian net together with its parameters; in fact, a bayesian net with structure s and parameters p is mapped on to a second bayesian net with fixed structure and parameters the value of which determine both s and p . This is an effect that is orthogonal to the upgrade to first order logic. In that sense, it appears BLP's extend bayesian networks in two different ways.

5.2 $CLP(\mathcal{BN})$ and PRM's

The existing approach that is closest to the one we propose, is the recently proposed $CLP(\mathcal{BN})$ by Santos Costa et al. [13]. Their approach is much closer to ours than that of Kersting and De Raedt, in that they cleanly separate structural and probabilistic information, and have clauses the meaning of which is very similar to what we use. A difference is that they do not use a meta-interpreter, but use the inference engine of a constraint logic programming system to perform inference. Thus, executing their code is equivalent to performing inference, whereas in our approach, executing the code just generates the bayesian network; to perform inference the meta-interpreter has to be used. Also, Santos Costa et al. use the concept of skolem constants to model stochastic variables, while we just use ground terms. The need for the concept of skolems follows from their embedding of bayesian inference into the constraint logic inference engine. From the point of view of explaining how the approach works, this complicates matters.

We illustrate the similarity between our approach and $CLP(\mathcal{BN})$ with an example. Santos Costa et al. [13] provide the following $CLP(\mathcal{BN})$ program as an example:

```

satisfaction(Reg, Sat) :-
    reg(Reg, Course, _),
    professor(Course, Prof),

```

```

ability(Prof,Abi),
grade(Reg, Grade),
sat_table(Abi, Grade, Table),
{Sat = satisfaction(Reg) with Table}.

```

The program states that the satisfaction associated with a student-course registration (how satisfied is the student with the course?) depends on the ability of the professor and the grade the student got for the course.

This would typically be represented in our framework as follows:

```

depends(satisfaction(Reg), [ability(Prof),grade(Reg)], Table) :-
  reg(Reg, Course, _),
  professor(Course, Prof),
  sat_table(Table).

```

This illustrates that our representation typically has about the same complexity as the representation used by the $CLP(BN)$ approach.

Santos Costa et al. compare their approach with Probabilistic Relational Networks (PRMs) [2], showing that $CLP(BN)$ is at least as powerful. This suggests that our meta-interpreter approach could also be used to describe PRMs.

5.3 Further Comparison with BLPs and $CLP(BN)$

We end this section with a small table that summarizes some of the differences between the meta-interpreter approach (MIA), BLPs, and $CLP(BN)$.

	BLP	$CLP(BN)$	MIA
stochastic var.	literal	skolem	term
dependency		:-	depends
prob. inference	engine	engine	meta-interpreter
queries	prob. inf.	prob. inf.	description of bayesian net

The table may clarify somewhat how the meta-interpreter approach treats bayesian nets at a meta-level; it describes a bayesian net, allowing not only standard probabilistic inference (queries about specific probabilities) but also queries regarding the structure of the bayesian net itself. Which variables depends on which other variables is explicitly described by the **depends** predicate, instead of implicitly through the structure of the program. Terms denote stochastic variables, and a meta-interpreter performs probabilistic inference, whereas in the other approaches the inference engine of the programming language takes care of this. Nevertheless, as suggested above, the meta-interpreter approach generalizes bayesian networks to the first order context as easily as the other approaches.

6 Conclusions

The contributions of this paper are as follows. We have proposed a simple formalism for representing first order bayesian networks in Prolog. We have discussed

the expressiveness that this method provides; it is clear that it can at least easily handle the examples with which these other approaches are motivated. The formalism has a simple semantics : the connection between a clause and the knowledge it represents is obvious. Understanding the inference that happens in a first order bayesian network, is decoupled into understanding the semantics of a Prolog program (which defines the structure of the bayesian net) and understanding inference in traditional bayesian networks. In other approaches, both are much more intertwined (because of the embedding of the probabilistic inference in the logical inference engine). We believe that, due to its simplicity, this approach holds some promise with respect to clarifying the relationship between the many different approaches that currently exist.

This work is obviously preliminary. Further work is needed on defining the semantics of our representation formalism more formally; implementing a full meta-interpreter for bayesian inference; implementing learning procedures; and finally, comparing the approach to other approaches. This comparison should include the more procedural approaches, where some kind of programming language is used to program probabilistic models, e.g., IBAL [10].

7 Acknowledgements

The author is a post-doctoral fellow of the Fund for Scientific Research of Flanders (FWO-Vlaanderen), Belgium. He thanks Luc De Raedt, Kristian Kersting, Marc Denecker, Joost Vennekens and Sašo Džeroski for useful comments and interesting discussions.

References

1. I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2001. 3rd Edition.
2. L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 7–34. Springer-Verlag, 2001.
3. P. Haddawy. Generating Bayesian networks from probability logic knowledge bases. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 262–269, 1994.
4. P. Haddawy. An overview of some recent developments in bayesian problem solving techniques. *AI Magazine*, Spring 1999.
5. J.Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1989.
6. M. Jaeger. Relational Bayesian networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 266–273. Morgan Kaufmann Publishers, 1997.
7. K. Kersting and L. De Raedt. Adaptive Bayesian logic programs. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 11th international conference on inductive logic programming*, pages 104–117. Springer-Verlag, 2001.

8. K. Kersting and L. De Raedt. Towards combining inductive logic programming and Bayesian networks. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 11th international conference on inductive logic programming*, pages 118–131. Springer-Verlag, 2001.
9. D. Koller and A. Pfeffer. Learning probabilities for noisy first-order rules. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1316–1323, 1997.
10. A. Pfeffer. IBAL : A probabilistic rational programming language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 733–740, 2001.
11. D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.
12. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.
13. V. Santos Costa, D. Page, M. Qazi, and J. Cussens. Clp(bn): Constraint logic programming for probabilistic knowledge. In *Proceedings of 19th Conference on Uncertainty in Artificial Intelligence*, 2003. To appear. See also <http://www.cos.ufrj.br/~vitor/Yap/clpbn/>.
14. T. Sato and Y. Kameya. Prism: A symbolic-statistical modeling language. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 1330–1335, 1997.
15. T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research (JAIR)*, 15:391–454, 2001.

A Meta-interpreter for Inference

The following simplified interpreter performs only forward inference, in bayesian networks without undirected cycles.

```
distr(A, Distr) :- input_node(A, Distr).
distr(A, Distr) :- depends(A, B, [H|T]), sumprobs(A,B,H,T, Distr).

sumprobs(_Var, _Parents, _Values, [], []).
sumprobs(Var, Parents, Values, [[Val|Probs]|Rows], [Val:Prob|Rest]) :-
    sumprobs2(Parents, Values, Probs, Prob),
    sumprobs(Var, Parents, Values, Rows, Rest).

sumprobs2(Parents, Values, Probs, Prob) :-
    distrlist(Parents, Values, Distr),
    vecprod(Probs, Distr, Prob).

distrlist(Parents, Values, Distr) :-
    separate_distr(Parents, SepDistr),
    multiply(Values, SepDistr, Distr).
```

```

multiply([], _, []).
multiply([Combination|Rest], SepDistr, [Prob|RestDistr]) :-
    getprobs(Combination, SepDistr, Problist),
    multiply_list(Problist, 1, Prob),
    multiply(Rest, SepDistr, RestDistr).

getprobs([], [], []).
getprobs([A|B], [D1|Rest], [Prob|RestProbs]) :-
    membercheck(A:Prob, D1),
    getprobs(B, Rest, RestProbs).

multiply_list([], Acc, Acc).
multiply_list([A|B], Acc, Res) :- Acc1 is A*Acc, multiply_list(B, Acc1, Res).

vecprod([], [], 0).
vecprod([A|B], [C|D], E) :- vecprod(B,D,F), E is A*C+F.

membercheck(A, [A|_]) :- !.
membercheck(A, [_|C]) :- membercheck(A, C).

separate_distr([], []).
separate_distr([Parent|RestParents], [Distr|RestDistr]) :-
    distr(Parent, Distr),
    separate_distr(RestParents, RestDistr).

```