Interpreting Bayesian Logic Programs

Kristian Kersting and **Luc De Raedt** and **Stefan Kramer**
Institute for Computer Science, Machine Learning Lab
University Freiburg, Am Flughafen 17, D-79110 Freiburg/Brg., Germany
{kersting, deraedt, skramer}@informatik.uni-freiburg.de

Abstract

Various proposals for combining first order logic with Bayesian nets exist. We introduce the formalism of Bayesian logic programs, which is basically a simplification and reformulation of Ngo and Haddawys probabilistic logic programs. However, Bayesian logic programs are sufficiently powerful to represent essentially the same knowledge in a more elegant manner. The elegance is illustrated by the fact that they can represent both Bayesian nets and definite clause programs (as in "pure" Prolog) and that their kernel in Prolog is actually an adaptation of an usual Prolog meta-interpreter.

Introduction

A Bayesian net (Pearl 1991) specifies a probability distribution over a fixed set of random variables. As such, Bayesian nets essentially provide an elegant probabilistic extension of propositional logic. However, the limitations of propositional logic, which Bayesian nets inherit, are well-known. These limitations motivated the development of knowledge representation mechanisms employing first order logic, such as e.g. in logic programming and Prolog. In this context, it is no surprise that various researchers have proposed various first order extensions of Bayesian nets: e.g. probabilistic logic programs (Ngo & Haddawy 1997), relational Bayesian nets (Jaeger 1997) and probabilistic relational models (Koller 1999). Many of these techniques employ the notion of *Knowledge-Based Model Construction* (Haddawy 1999) (KBMC), where first-order rules with associated uncertainty parameters are used as a basis for generating Bayesian nets for particular queries.

We tried to identify a formalism that is as simple as possible. While introducing Bayesian logic programming we employed one key design principle. The principle states that the resulting formalism should be as close as possible to both Bayesian nets and to some well-founded first order logic knowledge representation mechanism, in our case, "pure" Prolog programs. Any formalism designed according to this principle should be easily accessible and usable by researchers in both communities.

This paper is laid out as follows. The next three sections present the authors' solution: Bayesian logic programs. Section 5 shows a kernel implementation of them in Prolog. In the following section we give suggestions for learning Bayesian logic programs.

We assume some familiarity with Prolog or logic programming (see e.g. (Sterling & Shapiro 1986)) as well as with Bayesian nets (see e.g. (Russell & Norvig 1995)).

Bayesian logic programs

Bayesian logic programs consist of two components. The first component is the logical one. It consists of a set of Bayesian clauses (cf. below) which captures the qualitative structure of the domain and is based on"pure" Prolog. The second component is the quantitative one. It encodes the quantitative information about the domain and employs – as in Bayesian nets – the notions of a conditional probability table (CPT) and a combining rule.

A *Bayesian predicate* is a predicate r to which a finite domain D_r is associated. We define a *Bayesian definite clause* as an expression of the form $A \mid A_1, \ldots, A_n$ where the $A, A_1, \ldots, A_n$ are atoms and all variables are (implicitly) universally quantified. When writing down Bayesian definite clauses, we will closely follow Prolog notation (with the exception that Prolog's `:-` is replaced by `|`). So, variables start with a capital, constant and functor symbols start with a lowercase character. The main difference between Bayesian and classical clauses is that Bayesian atoms represent classes of similar random variables. More precisely, each ground atom in a Bayesian logic program represents a random variable. Each random variable can take on various possible values from the (finite) domain D_r of the corresponding Bayesian predicate r. In any state of the world, a random variable takes exactly one value. E.g., we paraphrase that James' house is not burglarized with $burglary(james) = false$. Therefore, a logical predicate r is a special case of a Bayesian one with $D_r = \{true, false\}$. An example of a Bayesian definite clause inspired on (Ngo & Haddawy 1997) is `burglary(X) | neighborhood(X).` where the domains are $D_{burglary} = \{true, false\}$ and $D_{neighbourhood} = \{bad, average, good\}$. Roughly speaking, a Bayesian definite clause specifies that for each substitution θ that grounds the clause the random variable $A\theta$ depends on $A_1\theta, \ldots, A_n\theta$. For instance, let $\theta = \{X \leftarrow james\}$, then the random variable *burglary(james)* depends on *neighbourhood(james)*.

As for Bayesian nets there is a table of conditional probabilities associated to each Bayesian definite clause[1].

neighbourhood(X)	*burglary(X)* *true*	*burglary(X)* *false*
bad	0.6	0.4
average	0.4	0.6
good	0.3	0.7

The CPT specifies our knowledge about the conditional probability distribution[2] $\mathbf{P}(A\theta \mid A_1\theta, \ldots, A_n\theta)$ for every ground instance θ of the clause. We assume total CPTs, i.e. for each tuple of values $\mathbf{u} \in D_{A_1} \times \ldots \times D_{A_n}$ the CPT specifies a distribution $P(D_A \mid \mathbf{u})$. For this reason we write $\mathbf{P}(A \mid A_1, \ldots, A_n)$ to denote the CPT associated to the Bayesian clause $A \mid A_1, \ldots, A_n$. For instance, the above Bayesian definite clause and CPT together imply that $P(burglary(james) = true \mid neighbourhood(james) = bad) = 0.6$. Each Bayesian predicate is defined by a set of definite Bayesian clauses, e.g.

```
alarm(X) | burglary(X).
alarm(X) | tornado(X).
```

If a ground atom A is influenced directly (see below) only by the ground atoms of the body of one ground instance of one clause, then the associated CPT specified a conditional probability distribution over A given the atoms of the body. But, if there are more than one different ground instances of rules which all have A as head, we have multiple conditional probability distribution over A – in particular this is the normal situation if a Bayesian atom is defined by several clauses. E.g., given the clauses for *alarm*, the random variable *alarm(james)* depends on both *burglary(james)* and *tornado(james)*. However, the CPT for *alarm* do not specify $\mathbf{P}(alarm(james) \mid burglary(james), tornado(james))$. The standard solution to obtain the desired probability distribution from the given CPTs is to use a so called combining rule. Theoretically speaking, a combination rule is any algorithm which maps every finite set of CPTs $\{\mathbf{P}(A \mid A_{i1}, \ldots, A_{in_i}) \mid 1 \leq i \leq m,\ n_i \geq 0\}$ over ground atoms onto one CPT, called combined CPT, $\mathbf{P}(A \mid B_1, \ldots, B_n)$ with $\{B_1, \ldots, B_n\} \subseteq \bigcup_{i=1}^{m} A_{i1}, \ldots, A_{in_i}$. The output is empty iff the input is empty. Our definition of a combining rule is basically a reformulation of the definition given in (Ngo & Haddawy 1997)[3]. As an example we consider the combining rule *max*. The functional formulation is

$$\mathbf{P}(A \mid \cup_{i=1}^{n} A_{i1}, \ldots, A_{in_i}) = \max_{i=1}^{n}\{\mathbf{P}(A \mid A_{i1}, \ldots, A_{in_i})\}$$

It is remarkable that a combining rule has full knowledge about the input, i.e., it knows all the appearing ground atoms or rather random variables and the associated domains of the random variables.

We assume that for each Bayesian predicate there is a corresponding combining rule and that the combined CPT still specifies a conditional probability distribution. From a practical perspective, the combining rules used in Bayesian logic programs will be those commonly employed in Bayesian nets, such as e.g. noisy-or, max.

Semantics of Bayesian logic programs

Following the principles of KBMC, each Bayesian logic program essentially specifies a propositional Bayesian net that can be queried using usual Bayesian net inference engines. This view implicitly assumes that all knowledge about the domain of discourse is encoded in the Bayesian logic program (e.g. the persons belonging to a family). If the domain of discourse changes (e.g. the family under consideration), then part of the Bayesian logic program has to be changed. Usually, these modifications will only concern ground facts (e.g. the Bayesian predicates "person", "parent" and "sex").

The structure of the corresponding Bayesian net follows from the semantics of the logic program, whereas the quantitative aspects are encoded in the CPTs and combining rules.

The set of random variables specified by a Bayesian logic program is the least Herbrand model of the program[4]. The least Herbrand model $LH(L)$ of a definite clause program L contains the set of all ground atoms that are logically entailed by the program[5], it represents the intended meaning of the program. By varying the evidence (some of the ground facts) one also modifies the set of random variables. Inference for logic programs has been well-studied (see e.g. (Lloyd 1989)) and various methods exist to answer queries or to compute the least Herbrand model. All of these methods can essentially be adapted to our context. Here, we assume that the computation of $LH(L)$ relies on the use

[1]In the examples, we use a naive representation as a table, because it is the simplest representation. We stress, however, that other representations are possible and known (Boutilier *et al.* 1996).

[2]We denote a single probability with P and a distribution with $\mathbf{P}$.

[3]It differs mainly in the restriction of the input set to be finite. We make this assumption in order to keep things simple.

[4]Formally it is the least Herbrand model of the logical program L', which one gets from L by omitting the associated CPTs and combination rules as well as interpreting all predicates as classical, logical predicates. For the benefit of greater readability, in the sequel we do not distinguish between L and L'.

[5]If we ignore termination issues, these atoms can – in principle – be computed by a theorem prover, such as e.g. Prolog.

of the well-known T_L (cf. (Lloyd 1989)) operator[6]. Let L be a Bayesian logic program.

$$T_L(\mathcal{I}) = \{A\theta \mid \text{there is a substitution } \theta \text{ and a clause } A \mid A_1, \ldots, A_n \text{ in } L \text{ such that } A\theta \mid A_1\theta, \ldots, A_n\theta \text{ is ground and all } A_i\theta \in \mathcal{I}\}$$

The least Herbrand model $LH(L)$ of L is then the least fixed point of $T_L(\emptyset)$. It specifies the set of relevant random variables. For instance, if one takes as Bayesian logic program the union of all Bayesian clauses written above together with *neighbourhood(james)* then $LH(L)$ consists of *neighbourhood(james)*, *burglary(james)* and *alarm(james)*. Notice that the least Herbrand model can be infinite when the logic program contains structured terms. This is not necessarily problematic as we will see later.

Given two ground atoms A and $B \in LH(L)$, we write that A *is directly influenced by* B if and only if there is a clause $A' \mid B_1, \ldots, B_n$ in L and a substitution θ that grounds the clause such that $A = A'\theta$ and $B = B_i\theta$ for some i and all $B_i\theta \in LH(L)$. The relation *influences* is then the recursive closure of the relation *directly influences*. Roughly speaking, a ground atom A influences B whenever there exists a proof for B that employs A. For instance, *alarm(james)* is influenced by *neighbourhood(james)* and directly influenced by *burglary(james)*. Using the *influenced by* relation we can now state a conditional independency assumptions: let $A_1, \ldots, A_n$ be the set of all random variables that directly influence a variable A. Then each other random variable B not influenced by A, is conditionally independent of A given $A_1, \ldots, A_n$, i.e. $\mathbf{P}(A \mid A_1, \ldots, A_n, B) = \mathbf{P}(A \mid A_1, \ldots, A_n)$. E.g. given the propositional Bayesian logic program (the famous example due to Pearl)

```
earthquake.
burglary.
alarm | earthquake, burglary.
johncalls | alarm.
marycalls | alarm.
```

the random variable `johncalls` is conditionally independent of `marycalls` given `alarm`.

Obviously, the relation *influenced by* should be acyclic in order to obtain a well-defined Bayesian net. The network can only be cyclic when there exists an atom A that influences itself. In this case executing the query ?- A (using e.g. Prolog) is also problematic – the SLD-tree (see below) of the query will be infinite and the query may not terminate. Thus the logical component of the Bayesian logic program is itself problematic. Additional simple considerations lead to the following proposition:

Proposition 1. *Let B be a Bayesian logic program and $LH(B)$ the least Herbrand model of B. If B fulfills the following conditions:*

1. *the influenced by relation over $LH(B)$ is acyclic and*
2. *each random variable in $LH(B)$ is only influenced by a finite set of random variables,*

then it specifies a distribution P over $LH(B)$ which is unique in the sense that for each finite subset $S \subset LH(B)$ the induced distribution $\mathbf{P}(S)$ is unique.

A proof can be found in (Kersting 2000). The conditions still allow infinite least Herbrand models but account for Bayesian nets: they are acyclic graphs and each node has a finite set of predecessors. Let us have a look at a program which violates the conditions, more exactly said, the properties of the random variable $r(a)$ together with the *directly influenced by* relation violates them:

```
r(a). s(a,b).
r(X) | r(X).
r(X) | s(X,f(Y)).
s(X,f(Y)) | s(X,Y).
```

Given this Program the random variable $r(a)$ is directly influenced by itself and by $s(a, f(b)), s(a, f(f((b))), \ldots$.

```
s(a).
r(X)    | r(f(X)).
r(f(X)) | s(f(X)).
s(f(X)) | s(X).
```

Given this Program the random variable $r(a)$ is influenced (not directly) by $r(f(a)), r(f(f((a))), \ldots$ though it has a finite proof. In this paper, we assume that the Bayesian logic program is unproblematic in this respect[7].

To summarize, the least Herbrand model of a Bayesian logic program specifies the random variables in the domain of discourse. These random variables can then in principle[8] be represented in a Bayesian net where the parents of a random variable v are all facts directly influencing v. Any algorithm solving the inference problem for Bayesian nets can now be applied. At this point it should be clear how Bayesian nets are represented as Bayesian logic programs. We only encode the dependency structure as a propositional Bayesian logic program. Everything else remains the same.

[6]For simplicity, we will assume that all clauses in a Bayesian logic program are range-restricted. This means that all variables appearing in the conclusion part of a clause also appear in the condition part. This is a common restriction in computational logic. When working with range-restricted clauses, all facts entailed by the program are ground. Also, the pruned and-or trees and graphs (cf. below) will only contain ground facts. This in turn guarantees that the constructed Bayesian net for any query contains only proper random variables.

[7]This is a reasonable assumption if the Bayesian logic program has been written by anyone familiar with Prolog.

[8]We neglect the finiteness of Bayesian nets for the moment.

Query-answering procedure

In this section, we show how to answer queries with Bayesian logic programs. A probabilistic query or shortly a query is an expression of the form

```
?- Q | E1=1,...,EN=eN
```

and asks for the conditional probability distribution $\mathbf{P}(Q \mid E_1 = e_1, \ldots, E_n = e_n)$. We first consider the case where no evidence is given, and then show how to extend this in the presence of evidence.

Querying without evidence

First, we show how to compute the probability of the different possible values for a ground atom (a random variable) Q. Given a Bayesian logic program,

```
lives_in(james,yorkshire).
burglary(james).
lives_in(stefan,freiburg).
tornado(yorkshire).
alarm(X)     | burglary(X).
alarm(X)     | lives_in(X,Y), tornado(Y).
```

the query `?- alarm(james)` asks for the probabilities of *alarm(james) = true* and *alarm(james) = false*. To answer a query ?- Q we do not have to compute the complete least Herbrand model of the Bayesian logic program. Indeed, the probability of Q only depends on the random variables that influence Q, which will be called *relevant* w.r.t. Q and the given Bayesian logic program. The relevant random variables are themselves the ground atoms needed to prove that Q is true (in the logical sense).

The usual execution model of logic programs relies on the notion of SLD-trees (see e.g. (Lloyd 1989; Sterling & Shapiro 1986)). For our purposes it is only important to realize that the succeeding branches in this tree contain all the relevant random variables. Furthermore, due to the range-restriction requirement all succeeding branches contain only ground facts. Instead of using the SLD-tree to answer the query in the probabilistic sense, we will use a *pruned and-or tree*, which can be obtained from the SLD-tree. The advantage of the pruned and-or tree is that it allows us to combine the probabilistic and logical computations. An *and-or tree* represents all possible partial proofs of the query. The nodes of an and-or tree are partitioned into *and* (black) and *or* (white) nodes. An *and* node for a query ?- $Q_1, \ldots, Q_n$ is proven if all of its successors nodes ?- Q_i are proven. An *or* node ?- Q is proven if at least one of its successors nodes is proven. There is a successor node ?- $A_1\theta, \ldots, A_n\theta$ for an *or* node ?- A if there exists a substitution θ and a Bayesian definite clause $A'|A_1, \ldots, A_n$ such that $A'\theta = A\theta$. Since we are only interested in those random variables used in successful proofs of the original query, we prune all subtrees which do not evaluate to true. A *pruned* and-or tree thus represents all proofs of the query. One such tree is shown in Figure 1. It is easy to see that each ground atom (random variable) has a unique pruned and-or tree. On the other hand, for some queries and Bayesian

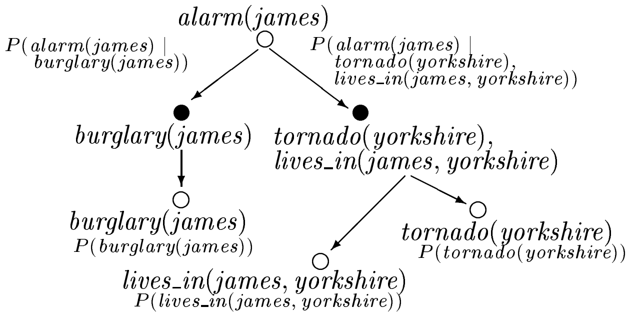

Figure 1: The pruned and-or tree (with associated CPTs) of the query `?- alarm(james)`.

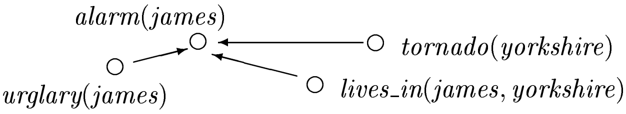

Figure 2: The dependency structure of the resulting Bayesian net of the query `?- alarm(james)`.

logic programs it might occur that a ground fact A occurs more than once in the pruned and-or tree. Given the uniqueness of pruned and-or trees for ground facts, it is necessary to turn the pruned and-or tree into an and-or graph by merging any nodes for A. This can actually be achieved by storing all ground atoms proven so-far in a look-up table, and using this table to avoid redundant computations.

The resulting pruned and-or graph compactly represents the dependencies between the random variables entailed by the Bayesian logic program. E.g. the tree in Figure 1 says that *burglary(james)* is influenced by *neighbourhood(james)*. Furthermore, the and-or graph reflects the structure of the quantitative computations required to answer the query. To perform this computation, we store at each branch from an *or* node to an *and* node the corresponding CPT (cf. Figure 1). The combined CPT for the random variable v in the *or* node is then obtained by combining the CPTs on v's sub-branches using the combining rule for the predicate in v. It is always possible to turn the and-or graph into a Bayesian net. This is realized by (1) deleting each *and* node n and redirecting each successor of n to the parent of n (as shown in Figure 2), and (2) by using the combined CPT at each *or* node.

Querying with evidence

So far, we have neglected the evidence. It takes the form of a set of ground random variables $\{E_1, \ldots, E_n\}$ and their corresponding values $\{e_1, \ldots, e_n\}$. The Bayesian net needed to compute the probability of a random variable Q given the evidence consists of the union of all pruned and-or graphs for the facts in $\{Q, E_1, \ldots, E_n\}$. This Bayesian net can be computed incrementally, starting by computing the graph (and the look-up table as described above) for Q and then using this graph and look-up table when answering the logical query for E_1 in order to guarantee that each random variable occurs only once in the resulting graph. The resulting graph is then the starting point for E_2 and so on. Given the corresponding Bayesian net of the final and-or graph, one can then answer the original query using any Bayesian net inference engine to compute

$$\mathbf{P}(Q \mid E_1 = e_1, \ldots, E_n = e_n).$$

The qualitative dependency structure of the resulting Bayesian net for the query `?- alarm(james)` is shown in Figure 1. Normally the resulting Bayesian net is not optimal and can be pruned.

Retrospectively we can say that a probabilistic query `?- Q | E1=e1,..., EN=eN` is legal if the union of all and-or graphs of `Q, E1,..., EN` is finite. In other words, the SLD trees of `Q, E1,..., EN` must be finite.

Implementation

The following Prolog code enables one to compute the structure of the pruned and-or graph of a random variable as a set of ground facts of the predicate *imply*, assuming that the logical structure of a Bayesian logic program is given as a Prolog program. The and-or graph is represented as a set of ground atoms of `imply(or:X,and:Y)` and `imply(and:X,or:Y)`. The use of the Prolog's own query procedure proves for two reasons as efficient: (1) it implements the desired search and (2) it is efficient and uses an efficient hash table. We do not present the entire source code, because the remaining program parts follow directly from the previous discussion.

```
build_AOG(Goal) :-
  clause(Goal, Body),imply(or:Goal,and:Body),!.
build_AOG(Goal) :-
  clause(Goal, Body), build_AOG_Body(Goal,Body),
  assert(imply(or:Goal,and:Body)).
build_AOG_Body(_,true):- !.
build_AOG_Body(_,(Body,Bodies)) :- !,
  build_AOG(Body),
  build_AOG_Conj((Body,Bodies),Bodies),
  assert(imply(and:(Body,Bodies),or:Body)).
build_AOG_Body(_,(Body)) :-
  build_AOG(Body), assert(imply(and:Body,or:Body)).
build_AOG_Conj((Goal,Goals),(Body,Bodies)) :- !,
  build_AOG(Body),
  build_AOG_Conj((Goal,Goals),Bodies),
  assert(imply(and:(Goal,Goals),or:Body)).
build_AOG_Conj((Goal,Goals),Body) :- !,
  build_AOG(Body),
  assert(imply(and:(Goal,Goals),or:Body)).
```

The pruned and-or graph is the component containing the root node as the following example clarifies. On the query `?- alarm(stefan).` the code asserts

```
imply(or:lives_in(stefan,freiburg), and:true).
```

The reason for that is that the and-or graph of a ground atom g, which comes in a body of a rule before a ground atom g', is asserted without proving the truth of g'. To extract the right component one can use the following code:

```
extract_pruned_AOG([]).
extract_pruned_AOG([_:true|Rest]) :-
  extract_pruned_AOG(Rest).
extract_pruned_AOG([Goal|Rest]) :-
  findall(Body,(imply(Goal,Body),
                not marked(Goal,Body),
                assert(marked(Goal,Body))),
                Successors),
  append(Successors,Rest,NewRest),
  extract_pruned_AOG(NewRest).
```

Calling `extract_pruned_AOG([or: alarm(stefan)])` it marks all nodes of the component containing the root node. After marking we can use

```
findall((X,Y),(imply(X,Y),
               not marked(X,Y),
               retract(imply(X,Y))),_).
```

to delete all irrelevant nodes and arcs. Furthermore, the code typifies the reason why "pure" Prolog programs as well as structured terms can be elegantly handled with Bayesian logic programs: it describes essentially an usual Prolog meta-interpreter. Moreover it should make the definitions of legal queries clearer.

Learning

So far we have merely introduced a framework that combines Bayesian nets with first order logic. In this section, we provide some initial ideas on how Bayesian logic programs might be learned.

The inputs to a system learning Bayesian logic programs should consist of a set of cases, where each case describes a set of random variables as well as their states. One complication that often arises while learning Bayesian nets and that is also relevant here is that some random variables or the corresponding states may not be fully observable.

In the literature on learning Bayesian nets (Heckerman 1995; Buntine 1996) one typically distinguishes between:

1. learning the structure of a net (model selection) and/or
2. learning the associated CPTs.

This distinction also applies to Bayesian logic programs, where one can separate the clausal structure from the CPTs. In addition, the combining rules could be learned[9].

Let us address each of these in turn.

For what concerns learning the underlying logic program of a Bayesian logic program, it is clear that techniques from the field of inductive logic programming (Muggleton & De Raedt 1994) could be helpful.

[9] For our suggestions we assume that the rules are determined by a user because learning the rules results in an explosion of complexity.

To give an idea of how this might work, we merely outline one possibility for learning the structure of the Bayesian logic program from a set of cases in which the relevant random variables are specified (though their values need not be known). This means that for each case we know the least Herbrand model. One technique for inducing clauses from models (or interpretations) is the clausal discovery technique by De Raedt and Dehaspe (De Raedt & Deshape 1997). Basically, this technique starts from a set of interpretations (which in our case corresponds to the Herbrand models of the cases) and will induce all clauses (within a given language bias) for which the interpretations are models. E.g. given the single interpretation $\{female(soetkin), male(maarten), human(maarten), human(soetkin)\}$ and an appropriate language bias the clausal discovery engine would induce $human(X) \leftarrow male(X)$ and $human(X) \leftarrow female(X)$. The Claudien algorithm essentially performs an exhaustive search through the space of clauses which is defined by a language $\mathcal{L}$. Roughly speaking, Claudien keeps track of a list of candidate clauses Q, which is initialized to the maximally general clause in $\mathcal{L}$. It repeatedly deletes a clause c from Q, and test whether all given interpretations are a model for c. If they are, c is added to the final hypothesis, otherwise all maximally general specializations of c in $\mathcal{L}$ are computed (using a so-called *refinement operator* (Muggleton & De Raedt 1994)) and added back to Q. This process continues until Q is empty and all relevant parts of the search-space have been considered. A declarative bias hand-written by the user determines the type of regularity searched for and reduces the size of the space in this way. The pure clausal discovery process as described by De Raedt and Dehaspe may induce cyclic logic programs. However, extensions as described in (Blockeel & De Raedt 1998) can avoid these problems.

If we assume that the logic program and the combination rules are given, we may learn the associated CPTs. Upon a first investigation, it seems that the work of (Koller & Pfeffer 1997) can be adapted towards Bayesian logic programs. They describe an EM based algorithm for learning the entries of CPTs of a probabilistic logic program in the framework of (Ngo & Haddawy 1997) which is strongly related to our framework as is shown in (Kersting 2000). The approach makes two reasonable assumptions: (1) different data cases are independent and (2) the combining rules are decomposable, i.e., they can be expressed using a set of separate nodes corresponding to the different influences, which are then combined in another node. As Koller and Pfeffer note, all commonly used combining rules meet this condition.

To summarize, it seems that ideas from inductive logic programming can be combined with those from Bayesian learning in order to induce Bayesian logic programs. Our further work intends to investigate these issues.

Conclusions

In this paper, we presented Bayesian logic programs as an intuitive and simple extension of Bayesian nets to first-order logic. Given Prolog as a basis, Bayesian logic programs can easily be interpreted using a variant of a standard meta-interpreter. We also indicated parallels to existing algorithms for learning the numeric entries in the CPTs and gave some promising suggestions for the computer-supported specification of the logical component.

Acknowledgements

We would like to thank Daphne Koller, Manfred Jaeger, Peter Flach and James Cussens for discussions and encouragement. The authors are also grateful to the anonymous reviewers.

References

Blockeel, H., and De Raedt, L. 1998. ISSID : an interactive system for database design. *Applied Artificial Intelligence* 12(5):385–420.

Boutilier, C.; Friedman, N.; Goldszmidt, M.; and Koller, D. 1996. Context-specific independence in baysian networks. In *Proc. of UAI-96.*

Buntine, W. 1996. A guide to the literature on learning probabilistic networks from data. *IEEE Trans. on Knowledge and Data Engineering* 8(2).

De Raedt, L., and Deshape, L. 1997. Clausal discovery. *Machine Learning* (26):99–146.

Haddawy, P. 1999. An overview of some recent developments on bayesian problem solving techniques. *AI Magazine - Special Issue on Uncertainty in AI.*

Heckerman, D. 1995. A tutorial on learning with bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research, Advanced Technology Division, Microsoft Corporation.

Jaeger, M. 1997. Relational bayesian networks. In *Proc. of UAI-1997.*

Kersting, K. 2000. Baye'sche-logisch Programme. Master's thesis, University of Freiburg, Germany.

Koller, D., and Pfeffer, A. 1997. Learning probabilities for noisy first-order rules. In *Proceedings of the Fifteenth Joint Conference on Artificial Intelligence.*

Koller, D. 1999. Probabilistic relational models. In *Proc. of 9th Int. Workschop on ILP.*

Lloyd, J. W. 1989. *Foundation of Logic Programming.* Berlin: Springer, 2. edition.

Muggleton, S., and De Raedt, L. 1994. Inductive logic programming: Theory and methods. *Journal of Logic Programming* 19(20):629–679.

Ngo, L., and Haddawy, P. 1997. Answering queries form context-sensitive probabilistic knowledge bases. *Theoretical Computer Science* 171:147–177.

Pearl, J. 1991. *Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 2. edition.

Russell, S. J., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc.

Sterling, L., and Shapiro, E. 1986. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press.