Bayesian Logic Programs*

Kristian Kersting and Luc De Raedt

Institute for Computer Science, Machine Learning Lab
Albert-Ludwigs-University, Georges-Köhler-Allee, Gebäude 079,
D-79085 Freiburg i. Brg., Germany
{kersting, deraedt}@informatik.uni-freiburg.de

Abstract. Various proposals for combining first order logic with Bayesian nets exist. Many of these are based on the so-called knowledge-based model construction method, e.g. probabilistic logic programs by Ngo and Haddawy, relational Bayesian nets by Jaeger and the more recent probabilistic relational models by Koller et. al. Upon a first investigation these frameworks seem different despite the fact that they attack essentially the same problem. The relationship among these approaches has so far not been studied. The main contribution of this paper is, that we clarify the relation among the three existing frameworks. This is achieved through the introduction of Bayesian logic programs, which serve as a common kernel to these first order Bayesian net approaches. Bayesian logic programs are not really new; they are basically a simplification and reformulation of Ngo's and Haddawy's probabilistic logic programs. However, Bayesian logic programs are sufficiently powerful to represent essentially the same knowledge in a more elegant manner. The elegance of Bayesian logic programs is illustrated by the fact that they can represent both propositional Bayesian nets and definite clause programs (as in pure Prolog).

1 Introduction

A Bayesian net [Pea91] specifies a probability distribution over a fixed set of discrete random variables[1]. 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 in the UAI community have proposed various first order extensions of Bayesian nets. Many of these techniques employ the notion of *Knowledge-based model construction* (KBMC) [Had99] , where first-order rules with associated uncertainty parameters are used as a basis for generating Bayesian nets for particular queries. This is especially useful in domains

* A slightly different version of the paper appears as technical report [KDK00] of the AAAI-2000 workshop on Learning Statistical Models from Relational Data (SRL).

[1] For the benefit of simplicity we neglect approaches for continuous random variables. References to the literature can be founded e.g. in [RN95].

where the number of relevant random variables depends on the specific problem. E.g. when reasoning about the probability that a person inherits a disease, the relevant random variables will depend crucially on the available knowledge and structure of the family of that person.

This paper investigates various approaches to building first order extensions of Bayesian nets: probabilistic logic programs [NH97] (PLP), relational Bayesian nets [Jae97] (RBN) and probabilistic relational models [Kol99] (PRM). Despite the fact that these approaches essentially address the same problem, they appear to be quite different. These differences are in part due to a different background (logic programming for Ngo and Haddawy, databases for Koller et al. and model theory for Jaeger). Furthermore, each of the the various approaches incorporates features that - in a sense - go beyond first order logic (for instance aggregate functions, coping with negation as failure, or using hierarchical combination functions). This makes it hard to understand the relationship among the various frameworks and to understand the fundamental primitives of a first order Bayesian net.

We take a different approach. We tried to identify a formalism that serves as a kind of common kernel to the above mentioned approaches and that is as simple as possible. Taking this framework we clarify the relations among PLPs, PRMs and RBNs. The result is the framework of Bayesian logic programs, which we introduce below and which is essentially a reformulation and simplification of the probabilistic logic programs by Ngo and Haddawy. While introducing Bayesian logic programming we employed one key design principle. The principle states that since we wish to combine first order logic with Bayesian nets, 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. If one accepts this principle, then it is important to show how "pure" Prolog programs as well as Bayesian nets turn out to be a special case of Bayesian logic programs. Furthermore, the formalism should be straightforward and intuitive.

The framework of Bayesian logic programs not only allows us to elegantly specify first order Bayesian nets, it also allows us to specify the relationships among the alternative frameworks. Therein the article differs from [KDK00] which does not investigate these relationships but stresses an interpretation of Bayesian logic programs using a Prolog engine and states suggestions how to support learning them.

The paper is laid out as follows. Sections 2-4 present the authors' solution: Bayesian logic programs. Section 5 shows that Bayesian logic programs satisfy the stated design principle. In section 6 we show how probabilistic logic programs, probabilistic relational models and relational Bayesian nets can be represented as Bayesian logic programs. Based on this discussion Section 7 concludes the main part of the paper with a short comparison of these approaches. The last section relates Bayesian logic programs to other frameworks for combining first-order logic with probability theory and states future work.

2 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 conditional probability table (CPT) and combining rule (cf. below). We assume some familiarity with Prolog or logic programming (see e.g. [SS86]) as well as with Bayesian nets (see e.g. [RN95]).

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 logical 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. The main difference between Bayesian and classical clauses is that Bayesian atoms represent classes or rather sets 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 predicate r. In any state of the world, a random variable takes exactly one value. 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 [NH97] is `burglary(X) | neighborhood(X).` where the domains are $D_{burglary} = \{yes, no\}$ and $D_{neighbourhood} = \{bad, average, good\}$. Roughly speaking, a Bayesian definite clause specifies that for each substitution θ (cf. [Llo89]) 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[2].

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[3] $\mathbf{P}(A\theta \mid A_1\theta, \ldots, A_n\theta)$ 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 $\mathbf{P}(D_A \mid \mathbf{u})$. For this reason we write $\mathbf{P}(A \mid A_1, \ldots, A_n)$ to denote

[2] 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 [BFGK96].

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

the CPT associated to the Bayesian clause $A \mid A_1, \ldots, A_n$. For instance, the above Bayesian definite clause and conditional probability table together imply that $P(\mathit{burglary}(\mathit{james}) = \mathit{true} \mid \mathit{neighbourhood}(\mathit{james}) = \mathit{bad}) = 0.6$.

Each Bayesian predicate is defined by a set of definite Bayesian clauses, e.g.

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

As noted, an associated CPT should specify a conditional probability distribution. But, if there is more than one different ground instance of a rule with the same ground atom as head, we have multiple conditional probability distributions over this atom – 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 conditional probability tables for *alarm* do not specify $\mathbf{P}(\mathit{alarm}(\mathit{james}) \mid \mathit{burglary}(\mathit{james}), \mathit{tornado}(\mathit{james})$. The standard solution to obtain this probability distribution from the conditional probability tables is to use a so called combining rule (we closely follow [NH97]). Theoretically speaking, a combination rule is any algorithm which maps every finite set of CPTs $\{\mathbf{P}(A \mid A_{i1}, \ldots, A_{in_i}) \mid 1 \leq i \leq m,\ n_i \geq 0\}$, $m \geq 1$, 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 [NH97][4].

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 combined CPTs still specify a conditional probability distribution and that for each Bayesian predicate there is exactly one corresponding combining rule. 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*.

3 Semantics of Bayesian logic programs

Following the principles of KBMC each Bayesian logic program actually specifies a propositional Bayesian net that can be queried using the usual Bayesian net inference engines. This view implicitly assumes that all knowledge about the

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

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 conditional probability tables and combining rules.

3.1 Structure of the Net

The role of the logical component is to specify the set of random variables. More precisely, the set of random variables specified by a Bayesian logic program is the least Herbrand model of the program[5]. The least Herbrand model $LH(L)$ of a definite clause program contains the set of all ground atoms that are logically entailed by the program[6], 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. [Llo89]) 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 merely sketch the computation of the least Herbrand model through the use of the well-known T_L operator[7]. Let L be a Bayesian logic program and $\mathcal{I}$ a model (cf. [Llo89]) of L.

$$\begin{aligned} T_L(\mathcal{I}) = \{A\theta \mid & \text{there is a substitution } \theta \text{ and a} \\ & \text{clause } A \mid A_1, \ldots, A_n \text{ in } L \text{ such that} \\ & A\theta \mid A_1\theta, \ldots, A_n\theta \text{ is ground and} \\ & \text{for all } i \in \{1, \ldots, n\}\text{: } A_i\theta \in \mathcal{I}\} \end{aligned}$$

The least Herbrand model $LH(L)$ of L is then the least fixpoint of $\mathcal{I} = \emptyset$. It specifies the set of random variables. For instance, if one takes as a

[5] 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'.

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

[7] 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.

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 for inference as we will show 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 the 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. $\mathbf{P}(A \mid A_1, \ldots, A_n, B) = \mathbf{P}(A \mid A_1, \ldots, A_n)$.

So far, we have ignored one important requirement: the relation *influenced by* should be acyclic in order to obtain a well-defined Bayesian net. The net can only be cyclic when there exists an atom that influences itself. However, when there exists such an atom A, executing the query ?-A (using Prolog) is also problematic (the SLD tree (cf. [Llo89]) of the query will be infinite and the query may not terminate). Thus in such cases 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 $\mathbf{P}$ 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 [Ker00]. 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[8].

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[9] 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.

4 Query-answering procedure

In this section, we show how to answer queries with Bayesian logic programs. We first consider the case where no evidence is given, and then show how to extend this in the presence of evidence.

4.1 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).
lives_in(stefan,freiburg).
neighbourhood(james).
tornado(yorkshire).
burglary(X) | neighbourhood(X).
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. These random variables 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. [Llo89, SS86]). For our purposes it is only important to realize that

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

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

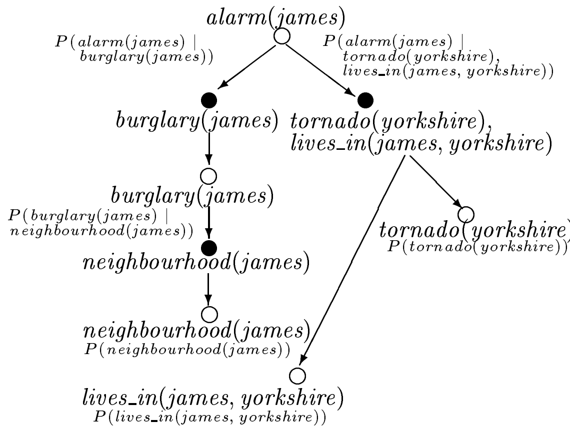

Fig. 1. The pruned and-or tree (with associated CPTs) of the query `?- alarm(james).`

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 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, this actually implies that the subtrees at both occurrences of A will be the same. It is then necessary to turn the pruned and-or tree into an and-or graph by merging the two nodes for A[10].

The resulting 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)*. Further-

[10] 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.

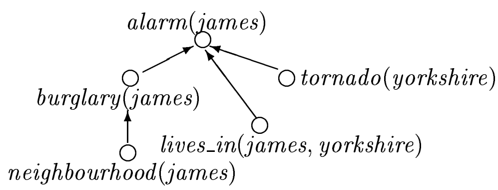

Fig. 2. The dependency structure of the resulting Bayesian net of the query `?-alarm(james)`.

more, 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* nodes n and redirecting each subnode of n to the parent of n (as shown in Figure 2), and (2) by using the combined CPT at each *or* node.

4.2 Querying with evidence

So far, we have neglected the evidence. It takes the form of a set of ground atoms or rather 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 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 2. Normally the resulting Bayesian nets are not optimal and can be pruned.

Having regard to proposition 1 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.

A_1 ... A_n	A *true*	A *false*
true ... *true*	1.0	0.0
... ...	0.0	1.0
false ... *false*	0.0	1.0

Table 1. CPT which is associated to a "logical" clause.

5 Special cases of Bayesian logic programs

In this section, we illustrate the representational power and elegance of Bayesian logic programs by demonstrating that both Bayesian nets and definite clause programs (as in "pure" Prolog) can directly and straightforwardly be encoded as Bayesian logic programs. Furthermore, we also give examples of the use of Bayesian logic programs that involve structured terms (and have an infinite least Herbrand model).

5.1 Bayesian nets

Any Bayesian net (over finite domains) directly translates to a Bayesian logic program in the following manner. Any node n in the Bayesian net will be defined by a single Bayesian clause of the form $n \mid p_1, \ldots, p_n$, where $\{p_1, \ldots, p_n\}$ is the set of all parents of n. The CPT associated to the node in the net becomes the CPT of the predicate n. As an illustration, consider the famous standard example of [Pea91] about burglary alarms at home. It translates to the following program:

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

where the associated CPTs are identical to the CPTs of the Bayesian net. It is easy to see that any program which get using is the described translation fullfils the conditions of proposition 1.

5.2 Pure Prolog programs and structured terms

Another interesting subclass of Bayesian logic programs are "pure" Prolog programs. The Bayesian logic program

```
father(jef,paul).
mother(an,paul).
parent(X,Y)       | father(X,Y).
parent(X,Y)       | mother(X,Y).
```

defines the parent predicate in terms of father and mother. Let us now define the domains of all predicates as $\{true, false\}$ and associate to each clause $A \mid A_1, \ldots, A_n$ the CPT of Table 1 which assigns a probability 1.0 to $A = true$ only when all of the A_i are $true$, otherwise $A = false$ with probability 1.0. If one then uses a combining rule like max or *noisy-or* one obtains the same semantics and behaviour as for "pure" Prolog. This also motivates us to use Prolog's notation `:-` instead of `|` (and the CPTs) as a short hand for these assumptions. Consider the following Bayesian logic program:

```
even(0).
even(s(X)) | odd(X).
odd(s(X))  | even(X).
```

It is easy to see that we can compute the probability distribution of any ground atom for the predicates *even* and *odd* with or without evidence. First, consider the case where there is no evidence. Then the SLD tree for the query will be finite and as a consequence also the and-or graph will be. This should allows us to compute the desired answer. Second, consider the case where there is evidence, e.g. $even(s(s(0))) = true$ and we want to compute the probability of $odd(s(0))$. Then we need to compute the Bayesian net, which is $even(0) \rightarrow odd(s(0)) \rightarrow even(s(s(0)))$, feed it into a propositional Bayesian net engine and compute the result. One can easily see that despite the presence of structured terms and an infinite number of random variables, the required computations are finite. This will - of course - only be the case when the corresponding Prolog program behaves well w.r.t. the query and evidence. In other words it should fullfil proposition 1.

6 Relationship to other frameworks

In this section, we investigate the relationship of Bayesian logic programs to three other first-order extensions of Bayesian nets: probabilistic logic programs (PLP), probabilistic relational models (PRM) and relational Bayesian nets (RBN).

6.1 Probabilistic logic programs

PLPs [NH95, NH97] follow the KBMC technique and also use a least Herbrand model to specify the relevant random variables. An example probabilistic logic program (inspired by [NH97]) is

$$\begin{aligned}
&P(neighbourhood(X, average)) = 0.4\\
&P(neighbourhood(X, good)) = 0.3\\
&P(burglary(X, yes) \mid neighbourhood(X, average)) = 0.4\\
&P(burglary(X, no) \mid neighbourhood(X, good)) = 0.7
\end{aligned}$$

It consists of four so-called probabilistic sentences. Each such a sentence states a probabilistic dependency between random variables, e.g. in the above example the a priori probability of the neighbourhood of any person X to be good is 0.3 and the a posteriori probability of a burglary given that the neighbourhood is good is 0.7. The example shows five main differences between PLPs and Bayesian logic programs, apart from which both approaches result to be equal:

First, instead of writing $neighbourhood(james) = average$ as in Bayesian logic programs, PLPs use $neighbourhood(james, average)$. This means that ground atoms in PLPs not only specify random variables but also the states of the variables. One side-effect of the first difference is that PLPs employ a more complicated and less efficient inference procedure. Indeed, to query `?-burglary(james)` the PLPs inference engine would construct one proof tree for all possible values of all possible random variables influencing $burglary(james)$.

Second, to guarantee that each random variable yields at most one value, PLPs need so called exclusivity constraints, such as $\leftarrow neighbourhood(X, average), neighbourhood(X, bad)$. This is not needed in Bayesian logic programs, which also use a more compact notation.

Third, in PLPs the qualitative information (the logical component) is mixed with the quantitative information (about the CPT), whereas in Bayesian logic programs this information is - as in Bayesian nets - nicely separated. This is due to the fact that a probabilistic sentence specifies an entry in a CPT. But this separation is considered an important advantage of Bayesian nets by [RN95], who write that Bayesian nets "are a natural way to represent conditional independence information. The links between nodes represent qualitative aspects of the domain, and the conditional probability tables represent the quantitative aspects."

Fourth, in PLPs it is possible to employ partially defined CPTs (as in the example above), i.e. some entries in a CPTs are undefined. Though Bayesian logic programs could - in principle - be extended to also allow for such partially defined CPTs[11], we prefer not to do so. Reasons for this are that it complicates the notation and also that it is unclear whether there are any advantages of partially defined CPTs[12].

Fifth, probabilistic sentences are essentially typed definite horn clauses (cf. [Llo89]) augmented with probability values. As long as the types are finite (and [NH97] give only for this case an implementable query-answering procedure) any typed definite horn clause can be transformed into an untyped, range-restricted horn clause as is shown in [Fla94, p. 112].

A further extention of PLPs is the use of context information. Context information is used to filter away clauses that do not apply to the current query from the knowledge base. Consider e.g. the following PLP (inspired by [NH97])

---

[11] Alternatively, we could handle partially defined CPTs using the combining rules.

[12] There is a discussion on the question whether there is need for partially CPTs (see [NH97, Jae98]). From the point of view of Bayesian nets we don't have to consider partial CPTs.

$$P(neighbourhood(X, bad)) = 0.2 \leftarrow live\_in(X, yorkshire)$$
$$P(neighbourhood(X, bad)) = 0.4 \leftarrow live\_in(X, vienna)$$

This PLP states that we have different CPTs depending on whether $live\_in(x, yorkshire)$ or $live\_in(x, vienna)$ is true or false, given some external logic program L[13]. Whereas context information may be important for efficiency reasons, we believe it is more natural to view this information as deterministic knowledge that can be specified using a "pure" Prolog program (a subset of Bayesian logic programs) although the filter process used in [NH97] can be easily incorporated into our framework. Using this approach, the last PLP would be written as the following Bayesian logic program

```
neighbourhood(X) | live_in(X,yorkshire)
neighbourhood(X) | live_in(X,vienna)
```

where we assume the existence of some "pure" Prolog clauses defining `live_in`. Though most of the above mentioned differences are merely syntactic or cosmetic ones as they do not really affect the expressive power of the formalisms, we believe these differences are important. One reason is the above mentioned design principle. From this perspective, it is a lot easier to write Bayesian logic programs and to understand their behaviour than with PLPs. This is easily seen by trying to write down and execute a "pure" Prolog program or a propositional Bayesian net as a PLP.

6.2 Probabilistic relational models

Koller et. al. [FGKP99, Kol99] define Probabilistic relational models, based on the well-known entity/relationship model. Figure 3 shows an example given in [FGKP99]: "it is a genetic model of the inheritance of a single gene that determines a person's blood type. Each person has two copies of the chromosome containing this gene, one inherited from her mother, and one inherited from her father. There is also a possibly contaminated test that attempts to recognize the person's blood type." In PRMs, the random variables are the attributes. The relations between entities are deterministic, i.e. they are only true or false. To represent this within Bayesian logic programs we use the following normal form: each attribute a of an entity type E is a Bayesian predicate $a(E)$ and each n-ary relation r is a n-ary logical predicate n. PRMs consist of a qualitative dependency structure over the attributes and their associated quantitative parameters (the CPTs). [FGKP99] distinguish among two types of parents. First, an attribute $a(X)$ of X can depend on another attribute $b(X)$ of X, e.g. the

[13] This external logic programming may employ negation, its clauses are so called normal logic programs. We cannot deal with negation as failure or completion semantics but we could use Bayesian logic programs to define negated predicates explicitly.

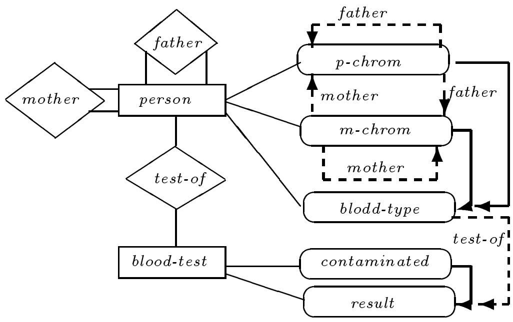

Fig. 3. PRM of the genetic model. We use the standard graphical notation of entity/relationship models: ovals represent attributes and boxes entities. Dashed lines indicate aggregations as parents, solid ones indicate attributes as parents.

blood type of a person depends on the chromosome inheritated from the father (*p-chrom*). This is equivalent to the Bayesian clause $a(X) \mid b(X)$. Second, an attribute $a(X)$ of X possibly depends on an attribute $b(Y)$ of a related entity Y, e.g. the chromosomes of a person depends on the chromosomes inherited from the mother (*m-chrom*). The relation between X and Y is described by a slot $s(X, Y)$ which is either a projection of a relation, i.e. $s(X, Y)$:- $r(X_1, \ldots, X_n)$, or a composition of slots, i.e. $s(X, Y)$:- $s_1(X, X_1), s_2(X_1, X_2), \ldots, s_m(X_{m-1}, Y)$. Given these clauses in a Bayesian logic program, the original dependency is represented by $a(X) \mid b(X), s(X, Y)$. Thus the example in Figure 3 can be represented by

```
m_chrom(X)      | mother(X,Y), p_chrom(Y), m_chrom(Y).
p_chrom(X)      | father(X,Y), p_chrom(Y), m_chrom(Y).
blood_type(X)   | m_chrom(X), p_chrom(X).
contaminated(X)| blood_test(X).
result(X)       | test_of(X,Y), contaminated(X), blood_type(Y).
```

One original feature of PRMs concerns the way they deal with multiple instantiations of a single clause. To this purpose Bayesian logic programs employ combining rules. PRMs however use aggregate functions (as in database languages) to map multiple values of a relevant attribute onto a single value. E.g. consider the above clause for *m_chrom* and assume that there are multiple possible values for Z. Then the PRM would apply an aggregate function to these values and use this derived attributes in the corresponding CPT. Let us first note that aggregate functions as well as combining rules do not belong to first order logic, they are typically considered second order. But in contrast to combining rules they first aggregate the states of some random variables and then specify a probability. If one would desire to use aggregate functions within Bayesian logic programs one has to simulate this calculation by moving the aggregate function inside the combining rules. We gather all relevant random variables thus a set of ground clauses, compute the aggregate of each joint state of them and finally specify the probabilities according to the original aggregate function. Indeed, the CPTs of a Bayesian logic program not only specify the distribution but also list the involved random variables as well as their values. Because the combining function combines the relevant CPTs it could implicitly employ the aggregate functions to realize the same effects. At this point it should be clear that PRMs employ a more restricted logical component than the other frameworks but a powerful mechanism to specify combined CPTs. Is should also be noted that PRMs are the only framework for which learning algorithms are known (see [FGKP99]).

6.3 Relational Bayesian nets

Jaeger [Jae97] considers Bayesian nets where the nodes are predicate symbols. The states of these random variables are then possible interpretations of the symbols over an arbitrary, finite domain (here we only consider Herbrand domains). Then random variables are set-valued. On the other hand, the inference problem addressed by Jaeger does not ask for the probability of a ground atom A belonging to any specific interpretation, but only for the probability that an interpretation contains A. Under these conditions, RBNs are viewed as Bayesian nets where the nodes are the ground atoms (over the domain) and all random variables have the logical domain, i.e. $\{\mathit{true}, \mathit{false}\}$[14]. The key difference between RBNs and the other frameworks is that CPTs are represented by so called probability formulae (PF). These PFs employ the notions of a combination function as well as that of equality constraints[15]. A combination function is any function that maps every finite multiset with elements from $[0, 1]$ into $[0, 1]$. The interesting point about PFs is that nested combination functions are allowed. [Jae97] gives the following example. Let $F_{cancer}(x)$ be

$$\mathit{noisy\text{-}or}\{\mathit{comb}_{\Gamma}\{\mathit{exposed}(x, y, z) \mid z; \mathit{true}\} \mid y; \mathit{true}\}$$

This PF states that that for any specific organ y, multiple exposures to radiation have a cumulative effect on the risk of developing cancer of y. But developing cancer at any of the various organs y can be viewed as independent causes. As shown in [Jae97] a PF not only specifies the CPT but also the dependency structure. [Ker00] shows that it is - in principle - possible to also model these hierarchical PFs as Bayesian logic programs, e.g. `cancer(X) | exposed(X,Y,Z).` with the right combining rule and CPT. We refer for a detailed discussion to [Ker00].

[14] It is possible, but complicated to model domains having more than two values.

[15] To simplify the discussion, we will further ignore these equality constraints here. For details we refer to [Ker00].

7 Conclusions

We have introduced Bayesian logic programs and compared them to well-known related frameworks. We have argued that Bayesian logic programs can actually serve as a kind of common kernel to these other approaches because Bayesian logic programs can essentially be used to represent the same knowledge. We have also listed distinctive features of the other approaches (such as aggregate functions and hierarchical probability formulae), which are not easily modeled by Bayesian logic programs and which could require extensions to the basic Bayesian logic program framework. On the other hand, by investigating the relation of Bayesian logic programs to PLPs, RBNs and PRMs, we have also - implicitly - investigated the relationships among these other frameworks. From this comparison, it is intuitively clear that there exists a positive inclusion chain for what concerns the expressive power of the various formalism. More specifically, PRMs can be represented by RBNs which in turn can be represented by PLPs if we assume finite domains for the random variables. A formal proof of this claim can be found in [Ker00]. Another point concerns our design principle. PRMs certainly satisfies this principle. However, instead of using first order logic, PRMs work with the more restricted formalism of databases more specific entity/relationship model. We believe this feature of Bayesian logic programs is important because it should allow people familiar with both Bayesian nets and "pure" Prolog to use them.

8 Related and future work

Besides the mentioned papers our framework is motivated by the formalism given in [Had94]. Furthermore by simplifying PLPs it also extends the work of [Poo93] because PLPs do so [NH97]. The idea of associating CPTs to clauses is also noticed in the work [FL98], though they see a ground atom as a random variable over $\{true, false\}$ and give a quite different query-answering procedure which e.g. does not incorporate the concept of combining rules. Instead, all clauses which have the same consequence are ranked according to a user defined value. For initial queries, the system will backward chain on the rules with the highest order value. Should the user wish to know whether there are any other chains of reasoning, alternative rules are expanded according to the order in a backtracking fashion (cf. [FL98]). The use of combining rules may be more appropriate than this proposed selection rule. In any case the selection rule can be easily incorporated into our formalism by extending the notion of combining rules.

We focused on first-order extensions of Bayesian nets. Also other combinations of first-order logic and probability theory are known, such as the work of [NS92] who introduced a probabilistic characterization of logic programming. The article [Par96] gives a survey. Stochastic logic programs [Mug96, Cus99] are another interesting approach. They are programs of labeled and unlabeled definite clauses. In contrast to Bayesian logic programs, probabilities are defined directly on the proofs of atomic formulas. But in doing this they are also related to Bayesian logic programs because both are using SLD trees. In the notation of Halpern [Hal89] the stochastic logic programs concern probabilities on the domain where as Bayesian logic programs as well as the other discussed extensions of Bayesian nets concern probabilities on possible worlds.

In the future Bayesian logic programs should be applied to some real-world problems. In this regard learning Bayesian logic programs has to be investigated. We believe the elaborated features of Bayesian logic programs will be advantageous. Some suggestions about learning can be founded in [KDK00]. Furthermore exploring of the relations between Bayesian logic programs and stochastic logic programs is interesting because of: (1) both are using SLD trees and (2) transformations between probabilities on the domain and probabilities on possible worlds exist as Halpern [Hal89] noted.

Acknowledgements

We would like to thank Stefan Kramer for useful discussions and for his comments on a draft of this paper. We thank James Cussens, Peter Flach, Manfred Jaeger and Daphne Koller for discussions and encouragement.

References

[BFGK96] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Contect-specific independence in Bayesian networks. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI-1996)*, 1996.

[Cus99] J. Cussens. Loginear models for first-order probabilistic reasoning. In *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-1999)*, 1999.

[FGKP99] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the Sixteenth International Joint Conferences on Artificial Intelligence (IJCAI-1999)*, 1999.

[FL98] I. Fabian and D. A. Lambert. First-order Bayesian reasoning. In G. Antoniou and S. Slaney, editors, *Proceedings of 11th Australian Joint Conference on Artificial Intelligence*, number 1502 in LNAI. Springer, 1998.

[Fla94] P. Flach. *Simply logical: intelligent reasoning by example.* John Wiley and Sons Ltd., 1994.

[Had94] P. Haddawy. Generating Bayesian networks from probabilistic logic knowledge bases. In *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-1994)*, 1994.

[Had99] P. Haddawy. An overview of some recent developments on Bayesian problem solving techniques. *AI Magazine - Special Issue on Uncertainty in AI*, Summer 1999. (to appear).

[Hal89] J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1989.

[Jae97] M. Jaeger. Relational Bayesian networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-1997)*, 1997.

[Jae98] M. Jaeger. Reasoning about infinite random structures with relational Bayesian networks. In *Proceedings of KR-98*, 1998.

[KDK00] K. Kersting, L. De Raedt, and S. Kramer. Interpreting Bayesian Logic Programs. In *Working Notes of the AAAI-2000 Workshop on Learning Statistical Models from Relational Data (SRL)*, Austin, Texas, July 2000. (to appear).

[Ker00] K. Kersting. Bayes'sche-logische Programme. Master's thesis, Albert-Ludwigs-University, Freiburg, Germany, 2000.

[Kol99] D. Koller. Probabilistic relational models. In S. Dzeroski and P. Flach, editors, *Proceedings of Ninth International Workschop on Inductive Logic Programming (ILP-1999)*, number 1634 in LNAI. Springer, 1999.

[Llo89] J. W. Lloyd. *Foundation of Logic Programming.* Springer, Berlin, 2. edition, 1989.

[Mug96] S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming.* IOS Press, 1996.

[NH95] L. Ngo and P. Haddawy. Probabilistic logic programming and Bayesian networks. In *Algorithms, Concurrency and Knowledge: Proceedings of the Asian Computing Science Conference 1995*, Pathumthai, Thailand, December 1995.

[NH97] L. Ngo and P. Haddawy. Answering queries form context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171:147–177, 1997.

[NS92] R. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[Par96] S. Parsons. Current approaches to handling imperfect information in data and knowledge bases. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):353–372, June 1996.

[Pea91] J. Pearl. *Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 2. edition, 1991.

[Poo93] D. Poole. Probabilistic horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

[RN95] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Inc., 1995.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, 1986.