1 Bayesian Logic Programming: Theory and Tool

Kristian Kersting
Institute for Computer Science, Machine Learning Lab
Albert-Ludwigs-Universität, Georges-Köhler-Allee, Gebäude 079
D-79085 Freiburg i. Brg., Germany
kersting@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/∼kersting

Luc De Raedt
Institute for Computer Science, Machine Learning Lab
Albert-Ludwigs-Universität, Georges-Köhler-Allee, Gebäude 079
D-79085 Freiburg i. Brg., Germany
deraedt@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/∼deraedt

1.1 Introduction

In recent years, there has been a significant interest in integrating probability theory with first order logic and relational representations [see De Raedt and Kersting, 2003, for an overview]. Muggleton [1996] and Cussens [1999] have upgraded stochastic grammars towards *Stochastic Logic Programs*, Sato and Kameya [2001] have introduced *Probabilistic Distributional Semantics* for logic programs, and Domingos and Richardson [2004] have upgraded Markov networks towards *Markov Logic Networks*. Another research stream including Poole's *Independent Choice Logic* [1993], Ngo and Haddawy's *Probabilistic-Logic Programs* [1997], Jäger's *Relational Bayesian Networks* [1997], and Pfeffer's *Probabilistic Relational Models* [2000] concentrates on first order logical and relational extensions of Bayesian networks.

Bayesian networks [Pearl, 1991] are one of the most important, efficient and elegant frameworks for representing and reasoning with probabilistic models. They have been applied to many real-world problems in diagnosis, forecasting, automated vision, sensor fusion, and manufacturing control [Heckerman et al., 1995]. A Bayesian network specifies a joint probability distribution over a finite set of

random variables and consists of two components:

1. a *qualitative* or *logical* one that encodes the local influences among the random variables using a directed acyclic graph, and

2. a *quantitative* one that encodes the probability densities over these local influences.

Despite these interesting properties, Bayesian networks also have a major limitation, i.e., they are essentially propositional representations. Indeed, imagine to model the localization of genes/proteins as was the task at the KDD Cup 2001 [Cheng et al., 2002]. When using a Bayesian network, every gene is a single random variable. There is no way of formulating general probabilistic regularities among the localizations of the genes such as

the localization `L` *of gene* `G` *is influenced by the localization* `L'` *of another gene* `G'` *that interacts with* `G`.

The propositional nature and limitations of Bayesian networks are similar to those of traditional attribute-value learning techniques, which have motivated a lot of work on upgrading these techniques within inductive logic programming. This in turn also explains the interest in upgrading Bayesian networks towards using first order logical representations.
Bayesian logic programs unify Bayesian networks with logic programming which allows the propositional character of Bayesian networks and the purely 'logical' nature of logic programs to be overcome. From a knowledge representation point of view, Bayesian logic programs can be distinguished from alternative frameworks by having both logic programs (i.e. definite clause programs, which are sometimes called 'pure' Prolog programs) as well as Bayesian networks as an immediate special case. This is realized through the use of a small but powerful set of primitives. Indeed, the underlying idea of Bayesian logic programs is to establish a one-to-one mapping between ground atoms and random variables, and between the *immediate consequence operator* and the *direct influence* relation. Therefore, Bayesian logic programs can also handle domains involving structured terms as well as continuous random variables.
In addition to reviewing Bayesian logic programs, this chapter

- contributes a *graphical representation* for Bayesian logic programs and
- its implementation in the Bayesian logic programs *tool* BALIOS, and
- shows how *purely logical predicates* as well as *aggregate function* are employed within Bayesian logic programs.

The chapter is structured as follows. We begin by briefly reviewing Bayesian networks and logic programs in Section 1.2. In Section 1.3, we define Bayesian logic programs as well as their semantics. Afterwards, in Section 1.4, we discuss several extensions of the basic Bayesian logic programming framework. More precisely, we introduce a graphical representation for Bayesian logic programs and we discuss

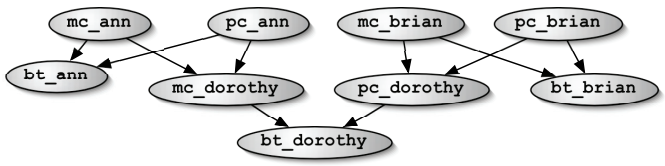


Figure 1.1 The graphical structure of a Bayesian network modelling the inheritance of blood types within a particular family.

the effective treatment of logic atoms and of aggregate functions. In Section 1.5, we sketch how to learn Bayesian logic programs from data. Before touching upon related work and concluding, we briefly present BALIOS, the engine for Bayesian logic programs.

1.2 On Bayesian Networks And Logic Programs

In this section, we first introduce the key concepts and assumptions underlying Bayesian networks and logic programs. In the next section we will then show how these are combined in Bayesian logic programs. For a full and detailed treatment of each of these topics, we refer to [Lloyd, 1989] for logic programming or Prolog and to [Jensen, 2001] for Bayesian networks.
We will introduce Bayesian logic programs using an example from genetics which is inspired by Friedman et al. [1999]:

"it is a genetic model of the inheritance of a single gene that determines a person's $\mathtt{X}$ *blood type* $\mathtt{bt(X)}$*. Each person* $\mathtt{X}$ *has two copies of the chromosome containing this gene, one,* $\mathtt{mc(Y)}$*, inherited from her mother* $\mathtt{m(Y,X)}$*, and one,* $\mathtt{pc(Z)}$*, inherited from her father* $\mathtt{f(Z,X)}$*."*

We will use the following convention: $\mathtt{x}$ denotes a (random) variable, x a state and $\mathbf{X}$ (resp. $\boldsymbol{x}$) a set of variables (resp. states). We will use $\mathbf{P}$ to denote a probability distribution, e.g. $\mathbf{P}(\mathtt{x})$, and P to denote a probability value, e.g. $P(\mathtt{x} = x)$ and $P(\mathbf{X} = \boldsymbol{x})$.

1.2.1 Bayesian Networks

A *Bayesian network* [Pearl, 1991] is an augmented, directed acyclic graph, where each node corresponds to a random variable $\mathtt{x_i}$ and each edge indicates a *direct influence* among the random variables. It represents the joint probability distribution $\mathbf{P}(\mathtt{x_1}, \ldots, \mathtt{x_n})$ over a fixed, finite set $\{\mathtt{x_1}, \ldots, \mathtt{x_n}\}$ of random variables. Each random variable $\mathtt{x_i}$ possesses a finite set $\mathbf{S}(\mathtt{x_i})$ of mutually exclusive states. Figure 1.1 shows the graph of a Bayesian network modelling our blood type example for a particular family. The familial relationship, which is taken

from Jensen's *stud farm* example [1996], forms the basis for the graph. The network encodes e.g. that Dorothy's blood type is influenced by the genetic information of her parents Ann and Brian. The set of possible states of $\mathtt{bt(dorothy)}$ is $\mathbf{S}(\mathtt{bt(dorothy)}) = \{a, b, ab, 0\}$; the set of possible states of $\mathtt{pc(dorothy)}$ and $\mathtt{mc(dorothy)}$ are $\mathbf{S}(\mathtt{pc(dorothy)}) = \mathbf{S}(\mathtt{mc(dorothy)}) = \{a, b, 0\}$. The same holds for $\mathtt{ann}$ and $\mathtt{brian}$. The direct predecessors of a node $\mathtt{x}$, the parents of $\mathtt{x}$ are denoted by $\mathbf{Pa}(\mathtt{x})$. For instance, $\mathbf{Pa}(\mathtt{bt(ann)}) = \{\mathtt{pc(ann)}, \mathtt{mc(ann)}\}$.
A Bayesian network stipulates the following conditional independence assumption.

Proposition 1.1 Independence Assumption of Bayesian Networks
Each node $\mathtt{x_i}$ in the graph is conditionally independent of any subset $\mathbf{A}$ of nodes that are not descendants of $\mathtt{x_i}$ given a joint state of $\mathbf{Pa}(\mathtt{x_i})$, i.e.

$$\mathbf{P}(\mathtt{x_i} \mid \mathbf{A}, \mathbf{Pa}(\mathtt{x_i})) = \mathbf{P}(\mathtt{x_i} \mid \mathbf{Pa}(\mathtt{x_i})) \ .$$

For example, $\mathtt{bt(dorothy)}$ is conditionally independent of $\mathtt{bt(ann)}$ given a joint state of its parents $\{\mathtt{pc(dorothy)}, \mathtt{mc(dorothy)}\}$. Any pair $(\mathtt{x_i}, \mathbf{Pa}(\mathtt{x_i}))$ is called the *family* of $\mathtt{x_i}$ denoted as $\mathbf{Fa}(\mathtt{x_i})$, e.g. $\mathtt{bt(dorothy)}$'s family is

$$(\mathtt{bt(dorothy)}, \{\mathtt{pc(dorothy)}, \mathtt{mc(dorothy)}\}) \ .$$

Because of the conditional independence assumption, we can write down the joint probability density as

$$\mathbf{P}(\mathtt{x_1}, \ldots, \mathtt{x_n}) = \prod_{i=1}^{n} \mathbf{P}(\mathtt{x_i} \mid \mathbf{Pa}(\mathtt{x_i}))$$

by applying the independence assumption 1.1 to the chain rule expression of the joint probability distribution. Thereby, we associate with each node $\mathtt{x_i}$ of the graph the conditional probability distribution $\mathbf{P}(\mathtt{x_i} \mid \mathbf{Pa}(\mathtt{x_i}))$, denoted as $\mathrm{cpd}(\mathtt{x_i})$. The conditional probability distributions in our blood type domain are:

$\mathtt{mc(dorothy)}$	$\mathtt{pc(dorothy)}$	$\mathbf{P}(\mathtt{bt(dorothy)})$
a	a	$(0.97, 0.01, 0.01, 0.01)$
b	a	$(0.01, 0.01, 0.97, 0.01)$
$\ldots$	$\ldots$	$\ldots$
0	0	$(0.01, 0.01, 0.01, 0.97)$

(similarly for $\mathtt{ann}$ and $\mathtt{brian}$) and

$\mathtt{mc(ann)}$	$\mathtt{pc(ann)}$	$\mathbf{P}(\mathtt{mc(dorothy)})$
a	a	$(0.98, 0.01, 0.01)$
b	a	$(0.01, 0.98, 0.01)$
$\ldots$	$\ldots$	$\ldots$
0	0	$(01, 0.01, 0.98)$

```
parent(jef,paul).                                    nat(0).
parent(paul,ann).                                    nat(s(X)) :- nat(X).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

Figure 1.2 Two logic programs, *grandparent* and *nat*.

(similarly for pc(dorothy)). Further conditional probability tables are associated with the apriori nodes, i.e., the nodes having no parents:

$\mathbf{P}(\texttt{mc(ann)})$	$\mathbf{P}(\texttt{mc(ann)})$	$\mathbf{P}(\texttt{mc(ann)})$	$\mathbf{P}(\texttt{mc(ann)})$
$(0.38, 0.12, 0.50)$	$(0.38, 0.12, 0.50)$	$(0.38, 0.12, 0.50)$	$(0.38, 0.12, 0.50)$

1.2.2 Logic Programs

To introduce logic programs, consider Figure 1.2, containing two programs, *grandparent* and *nat*. Formally speaking, we have that grandparent/2, parent/2 and nat/1 are *predicates* (with their *arity* i.e., number of arguments listed explicitly). Furthermore, jef, paul and ann are *constants* and X, Y and Z are *variables*. All constants and variables are also *terms*. In addition, there exist structured terms, such as s(X), which contains the *functor* s/1 of arity 1 and the term X. Constants are often considered as functors of arity 0. *Atoms* are predicate symbols followed by the necessary number of terms, e.g., $\texttt{parent}(\texttt{jef}, \texttt{paul})$, $\texttt{nat}(\texttt{s}(\texttt{X}))$, $\texttt{parent}(\texttt{X}, \texttt{Z})$, etc. We are now able to define the key concept of a (definite) *clause*. Clauses are formulas of the form $\texttt{A} :- \texttt{B}_1, \ldots, \texttt{B}_\texttt{m}$ where A and the $\texttt{B}_\texttt{i}$ are logical atoms where all variables are understood to be universally quantified. E.g., the clause $\texttt{grandparent}(\texttt{X}, \texttt{Y}) :- \texttt{parent}(\texttt{X}, \texttt{Z}), \texttt{parent}(\texttt{Z}, \texttt{Y})$ can be read as X is the grandparent of Y if X is a parent of Z and Z is a parent of Y. Let us call this clause c. We call $\texttt{grandparent}(\texttt{X}, \texttt{Y})$ the head(c) of this clause, and $\texttt{parent}(\texttt{X}, \texttt{Z}), \texttt{parent}(\texttt{Z}, \texttt{Y})$ the body(c). Clauses with an empty body, such as $\texttt{parent}(\texttt{jef}, \texttt{paul})$ are called *facts*. A (definite) clause program (or *logic program* for short) consists of a set of clauses. In Figure 1.2, there are thus two logic programs, one defining grandparent/2 and one defining nat/1.

The set of variables in a term, atom or clause E, is denoted as $\mathrm{Var}(E)$, e.g., $\mathrm{Var}(c) = \{\texttt{X}, \texttt{Y}, \texttt{Z}\}$. A term, atom or clause E is called *ground* when there is no variable occurring in E, i.e. $\mathrm{Var}(E) = \emptyset$. A *substitution* $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$, e.g. $\{\texttt{X}/\texttt{ann}\}$, is an assignment of terms t_i to variables V_i. Applying a substitution θ to a term, atom or clause e yields the instantiated term, atom, or clause $e\theta$ where all occurrences of the variables V_i are simultaneously replaced by the term t_i, e.g. $c\theta$ is $\texttt{grandparent}(\texttt{ann}, \texttt{Y}) :- \texttt{parent}(\texttt{ann}, \texttt{Z}), \texttt{parent}(\texttt{Z}, \texttt{Y})$.

The *Herbrand base* of a logic program T, denoted as $\mathrm{HB}(T)$, is the set of all ground atoms constructed with the predicate, constant and function symbols in

the alphabet of T. E.g., $\mathrm{HB}(nat) = \{\mathtt{nat(0)}, \mathtt{nat(s(0))}, \mathtt{nat(s(s(0)))}, ...\}$ and

$$\begin{aligned}&\mathrm{HB}(grandparent) = \\ &\quad\{\mathtt{parent(ann,ann)}, \mathtt{parent(jef,jef)},\\ &\quad\ \mathtt{parent(paul,paul)}, \mathtt{parent(ann,jef)}, \mathtt{parent(jef,ann)}, ...,\\ &\quad\ \mathtt{grandparent(ann,ann)}, \mathtt{grandparent(jef,jef)}, ...\}.\end{aligned}$$

A *Herbrand interpretation* for a logic program T is a subset of $\mathrm{HB}(T)$. The *least Herbrand model* $\mathrm{LH}(T)$ (which constitutes the semantics of the logic program) consists of all facts $f \in \mathrm{HB}(T)$ such that T logically entails f, i.e. $T \models f$. Various methods exist to compute the least Herbrand model. We merely sketch its computation through the use of the well-known *immediate consequence* operator T_B. The operator T_B is the function on the set of all Herbrand interpretations of B such that for any such interpretation $\mathcal{I}$ we have

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

Now, for range restricted clauses, the least Herbrand model can be obtained using the following procedure:

1: Initialize LH := $\emptyset$
2: **repeat**
3: LH := T_B(LH)
4: **until** LH does not change anymore

At this point the reader may want to verify that $\mathrm{LH}(nat) = \mathrm{HB}(nat)$ and

$$\begin{aligned}&\mathrm{LH}(grandparent) = \\ &\quad\{\mathtt{parent(jef,paul)}, \mathtt{parent(paul,ann)}, \mathtt{grandparent(jef,ann)}\}.\end{aligned}$$

1.3 Bayesian Logic Programs

The logical component of Bayesian networks essentially corresponds to a propositional logic program [1]. Consider for example the program in Figure 1.3. It encodes the structure of the blood type Bayesian network in Figure 1.1. Observe that the random variables in the Bayesian network correspond to logical atoms. Furthermore, the *direct influence* relation corresponds to the immediate consequence operator. Now, imagine another totally separated family, which could be described by a similar Bayesian network. The graphical structure and associated conditional prob-

1. Haddawy [1994] and Langley [1995] have a similar view on Bayesian networks. For instance, Langley does not represent Bayesian networks graphically but rather uses the notation of propositional definite clause programs.

```
pc(ann).
pc(brian).
mc(ann).
mc(brian).
mc(dorothy) :- mc(ann), pc(ann).
pc(dorothy) :- mc(brian), pc(brian).
bt(ann) :- mc(ann), pc(ann).
bt(brian) :- mc(brian), pc(brian).
bt(dorothy) :- mc(dorothy), pc(dorothy).
```

Figure 1.3 A propositional clause program encoding the structure of the blood type Bayesian network in Figure 1.1.

ability distribution for the two families are controlled by the same intensional regularities. But these overall regularities cannot be captured by a traditional Bayesian network. So, we need a way to represent these overall regularities.
Because this problem is akin to that with propositional logic and the structure of Bayesian networks can be represented using propositional clauses, the approach taken in Bayesian logic programs is to upgrade these propositional clauses encoding the structure of the Bayesian network to proper first order clauses.

1.3.1 Representation Language

Applying the above mentioned idea leads to the central notion of a Bayesian clause.

Definition 1.2 Bayesian Clause
A *Bayesian (definite) clause* c *is an expression of the form* $\mathtt{A} \mid \mathtt{A}_1, \ldots, \mathtt{A}_n$ *where* $n \geq 0$*, the* $\mathtt{A}, \mathtt{A}_1, \ldots, \mathtt{A}_n$ *are Bayesian atoms (see below) and all Bayesian atoms are (implicitly) universally quantified. When* $n = 0$*,* c *is called a Bayesian fact and expressed as* $\mathtt{A}$*.*

So, the differences between a *Bayesian clause* and a *logical clause* are:

1. the atoms $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_l)$ and predicates $\mathtt{p}/\mathtt{l}$ arising are Bayesian, which means that they have an associated (finite[2]) set $\mathbf{S}(\mathtt{p}/\mathtt{l})$ of possible states, and
2. we use '|' instead of ':−' to highlight the conditional probability distribution.

For instance, consider the Bayesian clause c $\mathtt{bt}(\mathtt{X})|\mathtt{mc}(\mathtt{X}), \mathtt{pc}(\mathtt{X})$ where $\mathbf{S}(\mathtt{bt}/\mathtt{1}) = \{a, b, ab, 0\}$ and $\mathbf{S}(\mathtt{mc}/\mathtt{1}) = \mathbf{S}(\mathtt{pc}/\mathtt{1}) = \{a, b, 0\}$. Intuitively, a Bayesian predicate $\mathtt{p}/\mathtt{l}$ generically represents a set of random variables. More precisely, each Bayesian ground atom g over $\mathtt{p}/\mathtt{l}$ represents a random variable over the states $\mathbf{S}(g) :=$

2. For the sake of simplicity we consider finite random variables, i.e. random variables having a finite set $\mathbf{S}$ of states. However, because the semantics rely on Bayesian networks, the ideas easily generalize to discrete and continuous random variables (modulo the restrictions well-known for Bayesian networks).

$\mathbf{S}(\mathtt{p/l})$. For example, $\mathtt{bt(ann)}$ represents the blood type of a person named Ann as a random variable over the states $\{a, b, ab, 0\}$. Apart from that, most *logical* notions carry over to Bayesian logic programs. So, we will speak of Bayesian predicates, terms, constants, substitutions, propositions, ground Bayesian clauses, Bayesian Herbrand interpretations etc. For the sake of simplicity we will sometimes omit the term *Bayesian* as long as no ambiguities arise. We will assume that all Bayesian clauses c are range-restricted, i.e., $\mathrm{Var}(\mathrm{head}(c)) \subseteq \mathrm{Var}(\mathrm{body}(c))$. Range restriction is often imposed in the database literature; it allows one to avoid the derivation of non-ground true facts (cf. Section 1.2.2). As already indicated while discussing Figure 1.3, a set of Bayesian clauses encodes the qualitative or structural component of the Bayesian logic programs. More precisely, ground atoms correspond to random variables, and the set of random variables encoded by a particular Bayesian logic program corresponds to its least Herbrand domain. In addition, the *direct influence* relation corresponds to the immediate consequence.
In order to represent a probabilistic model we also associate with each Bayesian clause c a conditional probability distribution $\mathrm{cpd}(c)$ encoding $\mathbf{P}(\mathrm{head}(c) \mid \mathrm{body}(c))$, cf. Figure 1.4. To keep the exposition simple, we will assume that $\mathrm{cpd}(c)$ is represented as a table. More elaborate representations such as decision trees or rules would be possible too. The distribution $\mathrm{cpd}(c)$ generically represents the conditional probability distributions associated with each ground instance $c\theta$ of the clause c.
In general, one may have many clauses. Consider clauses c_1 and c_2

```
bt(X) | mc(X).
bt(X) | pc(X).
```

and assume corresponding substitutions θ_i that ground the clauses c_i such that $\mathrm{head}(c_1\theta_1) = \mathrm{head}(c_2\theta_2)$. In contrast to $\mathtt{bt(X)|mc(X), pc(X)}$, they specify $\mathrm{cpd}(c_1\theta_1)$ and $\mathrm{cpd}(c_2\theta_2)$, but not the desired distribution $\mathbf{P}(\mathrm{head}(c_1\theta_1) \mid \mathrm{body}(c_1) \cup \mathrm{body}(c_2))$. The standard solution to obtain the distribution required are so called *combining rules*.

Definition 1.3 Combining Rule
A combining rule is a function that maps finite sets of conditional probability distributions $\{\mathbf{P}(\mathtt{A} \mid \mathtt{A_{i1}}, \ldots, \mathtt{A_{in_i}}) \mid i = 1, \ldots, m\}$ onto one (*combined*) conditional probability distribution $\mathbf{P}(\mathtt{A} \mid \mathtt{B_1}, \ldots, \mathtt{B_k})$ with $\{\mathtt{B_1}, \ldots, \mathtt{B_k}\} \subseteq \bigcup_{i=1}^{m} \{\mathtt{A_{i1}}, \ldots, \mathtt{A_{in_i}}\}$.

We assume that for each Bayesian predicate $\mathtt{p}/l$ there is a corresponding combining rule $\mathrm{cr}(\mathtt{p}/l)$, such as *noisy_or* [see e.g., Jensen, 2001] or *average*. The latter assumes $n_1 = \ldots = n_m$ and $\mathbf{S}(\mathtt{A_{ij}}) = \mathbf{S}(\mathtt{A_{kj}})$, and computes the average of the distributions over $\mathbf{S}(\mathtt{A})$ for each joint state over $\bigotimes_j \mathbf{S}(\mathtt{A_{ij}})$, see also the next Section 1.3.2.
By now, we are able to formally define Bayesian logic programs.

Definition 1.4 Bayesian Logic Program
A *Bayesian logic program* B consists of a (finite) set of Bayesian clauses. For each Bayesian clause c there is exactly one conditional probability distribution $\mathrm{cpd}(c)$,

m(ann, dorothy).
f(brian, dorothy).
pc(ann).
pc(brian).
mc(ann).
mc(brian).

mc(X)|m(Y, X), mc(Y), pc(Y).
pc(X)|f(Y, X), mc(Y), pc(Y).
bt(X)|mc(X), pc(X).

mc(X)	pc(X)	**P**(bt(X))
a	a	$(0.97, 0.01, 0.01, 0.01)$
b	a	$(0.01, 0.01, 0.97, 0.01)$
...	...	...
0	0	$(0.01, 0.01, 0.01, 0.97)$

m(Y, X)	mc(Y)	pc(Y)	**P**(mc(X))
true	a	a	$(0.98, 0.01, 0.01)$
true	b	a	$(0.01, 0.98, 0.01)$
...	...	...	...
false	a	a	$(0.33, 0.33, 0.33)$
...	...	...	...

Figure 1.4 The Bayesian logic program *blood type* encoding our genetic domain. For each Bayesian predicate, the identity is the combining rule. The conditional probability distributions associated with the Bayesian clauses bt(X)|mc(X), pc(X) and mc(X)|m(Y, X), mc(X), pc(Y) are represented as tables. The other distributions are correspondingly defined. The Bayesian predicates m/2 and f/2 have as possible states $\{true, false\}$.

and for each Bayesian predicate p/l there is exactly one combining rule cr(p/l).

A Bayesian logic program encoding our blood type domain is shown in Figure 1.4.

1.3.2 Declarative Semantics

Intuitively, each Bayesian logic program represents a (possibly infinite) Bayesian network, where the nodes are the atoms in the least Herbrand model of the Bayesian logic program. These declarative semantics can be formalized using the annotated *dependency graph*. The *dependency graph* $DG(B)$ is that directed graph whose nodes correspond to the ground atoms in the least Herbrand model LH(B). It encodes the *direct influence* relation over the random variables in LH(B): *there is an edge from a node* x *to a node* y *if and only if there exists a clause* $c \in B$ *and a substitution* θ*, s.t.* y = *head*$(c\theta)$*,* x $\in$ *body*$(c\theta)$ *and for all ground atoms* z *in* $c\theta$ *:* z $\in$ *LH*(B). Figures 1.5 and 1.6 show the dependency graph for our *blood type* program. Here, mc(dorothy) *directly* influences bt(dorothy). Furthermore, defining the *influence* relation as the transitive closure of the *direct influence* relation, mc(ann) influences bt(dorothy) .

The Herbrand base HB(B) constitute the set of all random variables we can talk about. However, only those atoms that are in the least Herbrand model LH(B) $\subseteq$ HB(B) will appear in the dependency graph. These are the atoms that are true in the logical sense, i.e., if the Bayesian logic program B is interpreted as a logical program. They are the so-called *relevant* random variables, the random variables over which a probability distribution is well-defined by B, as we will see. The atoms

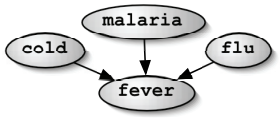
```
m(ann,dorothy).
f(brian,dorothy).
pc(ann).
pc(brian).
mc(ann).
mc(brian).
mc(dorothy) | m(ann, dorothy),mc(ann),pc(ann).
pc(dorothy) | f(brian, dorothy),mc(brian),pc(brian).
bt(ann) | mc(ann), pc(ann).
bt(brian) | mc(brian), pc(brian).
bt(dorothy) | mc(dorothy),pc(dorothy).
```

Figure 1.5 The grounded version of the *blood type* Bayesian logic program of Figure 1.4 where only clauses c with $\text{head}(c) \in \text{LH}(B)$ and $\text{body}(c) \subset \text{LH}(B)$ are retained. It (directly) encodes the Bayesian network as shown in Figure 1.6. The structure of the Bayesian network coincides with the dependency graph of the *blood type* Bayesian logic program.

not belonging to the least Herbrand model are irrelevant. Now, to each node `x` in $DG(B)$ we associate the combined conditional probability distribution which is the result of applying the combining rule $\text{cr}(\mathtt{p}/n)$ of the corresponding Bayesian predicate $\mathtt{p}/n$ to the set of $\text{cpd}(c\theta)$'s where $\text{head}(c\theta) = \mathtt{x}$ and $\{\mathtt{x}\} \cup \text{body}(c\theta) \subseteq \text{LH}(B)$. Consider

```
cold.          fever | cold.
flu.           fever | flu.
malaria.       fever | malaria.
```

where all Bayesian predicates have *true*, *false* as states and *noisy_or* as combining rule. The dependency graph is



and *noisy or* $\{\mathbf{P}(\mathtt{fever}|\mathtt{flu}), \mathbf{P}(\mathtt{fever}|\mathtt{cold}), \mathbf{P}(\mathtt{fever}|\mathtt{malaria})\}$ is associated with `fever` [see Russell and Norvig, 1995, pq. 444]. Thus, if $DG(B)$ is acyclic and not empty, and every node in $DG(B)$ has a finite indegree then $DG(B)$ encodes a (possibly infinite) Bayesian network, because the least Herbrand model always exists and is unique. Consequently, the following independence assumption holds:

Proposition 1.5 Independence Assumption of Dependency Graph
Each node `x` is independent of its non-descendants given a joint state of its parents $\mathbf{Pa}(\mathtt{x})$ in the dependency graph.

For instance the dependency graph of the *blood type* program as shown in Figures 1.5 and 1.6 encodes that the random variable $\mathtt{bt}(\mathtt{dorothy})$ is independent from $\mathtt{pc}(\mathtt{ann})$ given a joint state of $\mathtt{pc}(\mathtt{dorothy}), \mathtt{mc}(\mathtt{dorothy})$. Using this assumption the following proposition [taken from Kersting and De Raedt, 2001b] holds:

Proposition 1.6 Semantics
Let B be a Bayesian logic program. If

1. $\mathrm{LH}(B) \neq \emptyset$,
2. $DG(B)$ is acyclic, and
3. each node in $DG(B)$ is influenced by a finite set of random variables

then B specifies a unique probability distribution $\mathbf{P}_B$ over $\mathrm{LH}(B)$.

To see this, note that the least Herbrand $\mathrm{LH}(B)$ always exists, is unique and countable. Thus, $DG(B)$ exists and is unique, and due to condition (3) the combined probability distribution for each node of $DG(B)$ is computable. Furthermore, because of condition (1) a total order π on $DG(B)$ exists, so that one can see B together with π as a stochastic process over $\mathrm{LH}(B)$. An induction argument over π together with condition 2 allow one to conclude that the family of finite-dimensional distributions of the process is projective (cf. Bauer [1991]), i.e. , the joint probability distribution over each finite subset $\mathbf{S} \subseteq \mathrm{LH}(B)$ is uniquely defined and $\sum_y \mathbf{P}(\mathbf{S}, \mathtt{x} = y) = \mathbf{P}(\mathbf{S})$. Thus, the preconditions of *Kolmogorov's theorem* [Bauer, 1991, page 307] hold, and it follows that B given π specifies a probability distribution $\mathbf{P}$ over $\mathrm{LH}(B)$. This proves the proposition because the total order π used for the induction is arbitrary.
A program B satisfying the conditions (1), (2) and (3) of Proposition 1.6 is called *well-defined*. A well-defined Bayesian logic program B specifies a joint distribution over the random variables in the least Herbrand model $\mathrm{LH}(B)$. As with Bayesian networks, the joint distribution over these random variables can be factored to

$$\mathbf{P}(\mathrm{LH}(B)) = \prod_{\mathtt{x} \in \mathrm{LH}(B)} \mathbf{P}(\mathtt{x} | \mathbf{Pa}(\mathtt{x}))$$

where the *parent* relation **Pa** is according to the dependency graph.
The *blood type* Bayesian logic program in Figure 1.4 is an example of a well-defined Bayesian logic program. Its grounded version is shown in Figure 1.5. It essentially encodes the original blood type Bayesian network of Figures 1.1 and 1.3. The only differences are the two predicates `m/2` and `f/2` which can be in one of the logical set of states *true* and *false*. Using these predicates and an appropriate set of Bayesian facts (the 'extension') one can encode the Bayesian network for any family. This situation is akin to that in deductive databases, where the 'intension' (the clauses) encode the overall regularities and the 'extension' (the facts) the specific context of interest. By interchanging the extension, one can swap contexts (in our case, families).

1.3.3 Procedural Semantics

Clearly, any (conditional) probability distribution over random variables of the Bayesian network corresponding to the least Herbrand model can — in principle — be computed. As the least Herbrand model (and therefore the corresponding

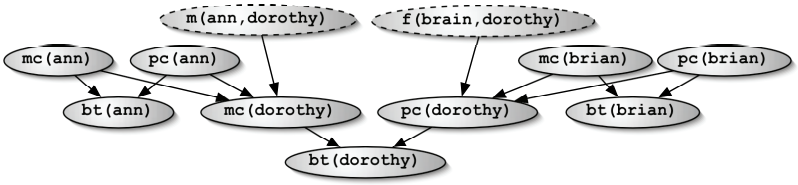


Figure 1.6 The structure of the Bayesian network represented by the grounded *blood type* Bayesian logic program in Figure 1.5. The structure of the Bayesian network coincides with the dependency graph. Omitting the dashed nodes yields the original Bayesian network of Figure 1.1.

Bayesian network) can become (even infinitely) large, the question arises as to whether one needs to construct the full least Herbrand model (and Bayesian network) to be able to perform inferences. Here, inference means the process of answering probabilistic queries.

Definition 1.7 Probabilistic Query
A probabilistic query to a Bayesian logic program B is an expression of the form

$$\texttt{?-}\ \mathtt{q}_1, \ldots, \mathtt{q}_n \mid \mathtt{e}_1 = e_1, \ldots, \mathtt{e}_m = e_m$$

where $n > 0$, $m \geq 0$. It asks for the conditional probability distribution

$$\mathbf{P}(\mathtt{q}_1, \ldots, \mathtt{q}_n \mid \mathtt{e}_1 = e_1, \ldots, \mathtt{e}_m = e_m)$$

of the query variables $\mathtt{q}_1, \ldots, \mathtt{q}_n$ where $\{\mathtt{q}_1, \ldots, \mathtt{q}_n, \mathtt{e}_1, \ldots, \mathtt{e}_m\} \subseteq \mathrm{HB}(B)$.

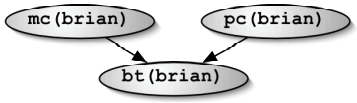
To answer a probabilistic query, one fortunately does not have to compute the complete least Herbrand model. It suffices to consider the so-called support network.

Definition 1.8 Support Network
The *support network* N of a random variable $\mathtt{x} \in \mathrm{LH}(B)$ is defined as the induced subnetwork of

$$\{\mathtt{x}\} \cup \{\mathtt{y} \mid \mathtt{y} \in \mathrm{LH}(B) \text{ and } \mathtt{y} \text{ influences } \mathtt{x}\}\ .$$

The support network of a finite set $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\} \subseteq \mathrm{LH}(B)$ is the union of the networks of each single $\mathtt{x}_i$.

For instance, the support network for `bt(dorothy)` is the Bayesian network shown in Figure 1.6. The support network for `bt(brian)` is the subnetwork with root `bt(brian)`, i.e.



That the support network of a finite set $\mathbf{X} \subseteq \mathrm{LH}(B)$ is sufficient to compute $\mathbf{P}(\mathbf{X})$ follows from the following Theorem [taken from Kersting and De Raedt, 2001b]:

Theorem 1.9 Support Network
Let N be a possibly infinite Bayesian network, let $\mathbf{Q}$ be nodes of N and $\mathbf{E} = \boldsymbol{e}$, $\mathbf{E} \subset N$, be some evidence. The computation of $\mathbf{P}(\mathbf{Q} \mid \mathbf{E} = \boldsymbol{e})$ does not depend on any node $\mathbf{x}$ of N which is not a member of the support network $N(\mathbf{Q} \cup \mathbf{E})$.

To compute the support network $N(\{\mathtt{q}\})$ of a single variable $\mathtt{q}$ efficiently, let us look at logic programs from a proof theoretic perspective. From this perspective, a logic program can be used to prove that certain atoms or goals (see below) are logically entailed by the program. Provable ground atoms are members of the least Herbrand model.
Proofs are typically constructed using the SLD-resolution procedure which we will now briefly introduce. Given a goal $\mathtt{:-G_1, G_2 \ldots, G_n}$ and a clause $\mathtt{G:-L_1, \ldots, L_m}$ such that $\mathtt{G_1}\theta = \mathtt{G}\theta$, applying SLD resolution yields the new goal $\mathtt{:-L_1}\theta, \ldots, \mathtt{L_m}\theta, \mathtt{G_2}\theta \ldots, \mathtt{G_n}\theta$. A *successful* refutation, i.e., a proof of a goal is then a sequence of resolution steps yielding the empty goal, i.e. :- . *Failed* proofs do not end in the empty goal. For instance in our running example, $\mathtt{bt(dorothy)}$ is true, because of the following refutation:

$\mathtt{:-bt(dorothy)}$
$\mathtt{:-mc(dorothy), pc(dorothy)}$
$\mathtt{:-m(ann, dorothy), mc(ann), pc(ann), pc(dorothy)}$
$\mathtt{:-mc(annn), pc(ann), pc(dorothy)}$
$\mathtt{:-pc(ann), pc(dorothy)}$
$\mathtt{:-pc(dorothy)}$
$\mathtt{:-f(brian, dorothy), mc(brian), pc(brian)}$
$\mathtt{:-mc(brian), pc(brian)}$
$\mathtt{:-pc(brian)}$
:-

Resolution is employed by many theorem provers (such as Prolog). Indeed, when given the goal $\mathtt{bt(dorothy)}$, Prolog would compute the above successful resolution refutation and answer that the goal is true.
The set of all proofs of $\mathtt{:-bt(dorothy)}$ captures all information needed to compute $N(\{\mathtt{bt(dorothy)}\})$. More exactly, the set of all ground clauses employed to prove $\mathtt{bt(dorothy)}$ constitutes the families of the support network $N(\{\mathtt{bt(dorothy)}\})$. For $\mathtt{:-bt(dorothy)}$, they are the ground clauses shown in Figure 1.5. To build the support network, we only have to gather all ground clauses used to prove the query variable and have to combine multiple copies of ground clauses with the same head using corresponding combining rules. To summarize, the support network $N(\{\mathtt{q}\})$ can be computed as follows:

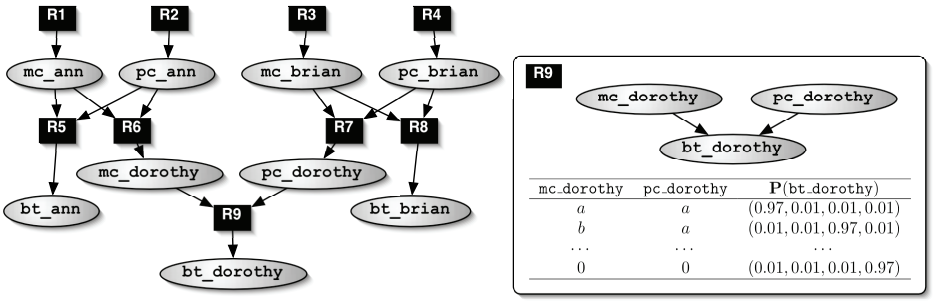


Figure 1.7 The rule graph for the *blood type* Bayesian network. On the right-hand side the local probability model associated with node R9 is shown, i.e., the Bayesian clause `bt_dorothy`|`mc_dorothy`, `pc_dorothy` with associated conditional probability table.

1: Compute all proofs for :-`q`.
2: Extract the set S of ground clauses used to prove :-`q`.
3: Combine multiple copies of ground clauses `h`|`b` $\in S$ with the same head `h` using combining rules.

Applying this to :-`bt`(`dorothy`) yields the support network as shown in Figure 1.6. Furthermore, the method can easily be extended to compute the support network for $\mathbf{P}(\mathbf{Q} \mid \mathbf{E} = \boldsymbol{e})$. We simply compute all proofs of :-`q`, `q` $\in \mathbf{Q}$, and :-`e`, `e` $\in \mathbf{E}$. The resulting support network can be fed into any (exact or approximative) Bayesian network engine to compute the resulting (conditional) probability distribution of the query. To minimize the size of the support network, one might also apply Schachter's Bayes' Ball algorithm [1998].

1.4 Extensions of the Basic Framework

So far, we described the basic Bayesian logic programming framework and defined the semantics of Bayesian logic programs. Various useful extensions and modifications are possible. In this section, we will discuss a *graphical representation*, efficient treatment of *logical atoms*, and *aggregate functions*. At the same time, we will also present further examples of Bayesian logic programs such as hidden Markov models [Rabiner, 1989] and probabilistic grammars [Manning and Schütze, 1999].

1.4.1 Graphical Representation

Bayesian logic programs have so far been introduced using an adaption of a logic programming syntax. Bayesian network are, however, also graphical models and owe at least part of their popularity to their intuitively appealing graphical notation [Jordan, 1998]. Inspired by Bayesian networks, we develop in this section

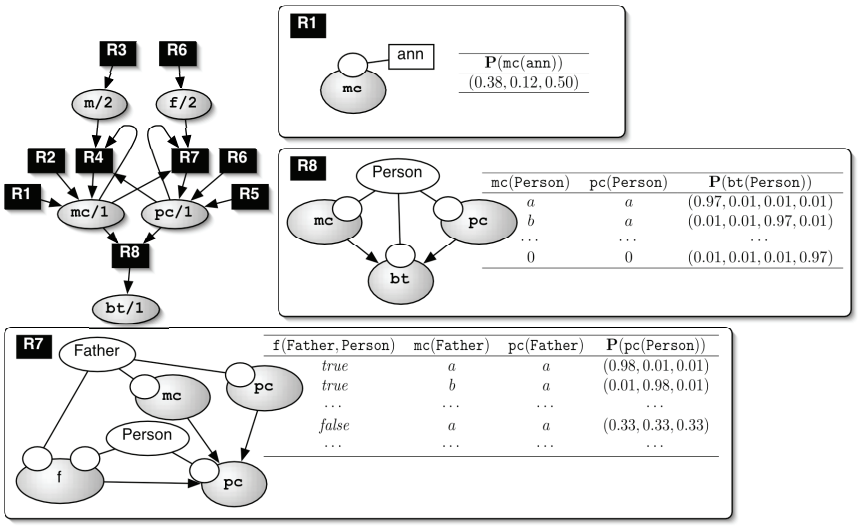


Figure 1.8 The graphical representation of the *blood type* Bayesian logic program. On the right-hand side, some local probability models associated with Bayesian clause nodes are shown, e.g. , the Bayesian clause $R7$ $\texttt{pc(Person)}|\texttt{f(Father, Person)}, \texttt{mc(Father)}, \texttt{pc(Father)}$ with associated conditional probability distribution. For the sake of simplicity, not all Bayesian clauses are shown.

a graphical notation for Bayesian logic programs.
In order to develop a graphical representation for Bayesian logic programs, let us first consider a more redundant representation for Bayesian networks: augmented bipartite (directed acyclic) graphs as shown in Figure 1.7. In a bipartite graph, the set of nodes is composed of two disjoint sets such that no two nodes within the same set are adjacent. There are two types of nodes, namely

1. *gradient gray ovals* denoting random variables, and
2. *black boxes* denoting local probability models.

There is a box for each family $\mathbf{Fa}(\mathtt{x_i})$ in the Bayesian network. The incoming edges refer to the parents $\mathbf{Pa}(\mathtt{x_i})$; the single outgoing edge points to X_i. Each box is augmented with a Bayesian network fragment specifying the conditional probability distribution $\mathbf{P}(\mathtt{x_i}|\mathbf{Pa}(\mathtt{x_i}))$. For instance in Figure 1.7, the fragment associated with $R9$ specifies the conditional probability distribution of $\mathbf{P}(\mathtt{bt(dorothy)}|\mathtt{mc(dorothy)}, \mathtt{pc(dorothy)})$. Interpreting this as a propositional Bayesian logic program, the graph can be viewed as a *rule graph* as known from database theory. Ovals represent Bayesian predicates, and boxes denote Bayesian clauses. More precisely, given a (propositional) Bayesian logic program B with Bayesian clauses $R_i \equiv \mathtt{h_i}|\mathtt{b_{i_1}}, \ldots, \mathtt{b_{i_m}}$, there are edges from from R_i to $\mathtt{h_i}$ and from $\mathtt{b_{i_j}}$ to R_i . Furthermore, to each Bayesian clause node, we associate the cor-

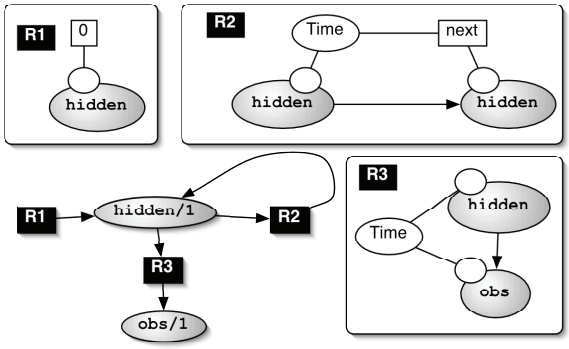
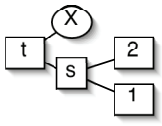


Figure 1.9 A dynamic Bayesian logic program modeling a hidden Markov model. The functor `next/1` is used to encode the discrete time.

responding Bayesian clause as a Bayesian network fragment. Indeed, the graphical model in Figure 1.7 represents the propositional Bayesian logic program of Figure 1.5.

In order to represent first order Bayesian logic programs graphically, we have to encode Bayesian atoms and their variable bindings in the associated local probability models. Indeed, logical terms can naturally be represented graphically. They form trees. For instance, the term $\texttt{t}(\texttt{s}(1,2),\texttt{X})$ corresponds to the tree



Logical variables such as `X` are encoded as white ovals. Constants and functors such as `1`, `2`, `s`, and `t` are represented as white boxes. Bayesian atoms are represented as gradient grays ovals containing the predicate name such as `pc`. Arguments of atoms are treated as placeholders for terms. They are represented as white circles on the boundary of the ovals (ordered from left to right). The term appearing in the argument is represented by an undirected edge between the white oval representing the argument and the 'root' of the tree encoding the term (we start in the argument and follow the tree until reaching variables).

As an example, consider the Bayesian logic program in Figure 1.8. It models the *blood type* domain. The graphical representation indeed conveys the meaning of the Bayesian clause $R7$: *the paternal genetic information* `pc(Person)` *of a person is influenced by the maternal* `mc(M)` *and the paternal* `pc(M)` *genetic information of the person's* `Father`.

As another example, consider Figure 1.9 which shows the use of functors to represent dynamic probabilistic models. More precisely, it shows a hidden Markov model (HMM) [Rabiner, 1989]. HMMs are extremely popular for analyzing sequential data. Application areas include computational biology, user modeling, speech

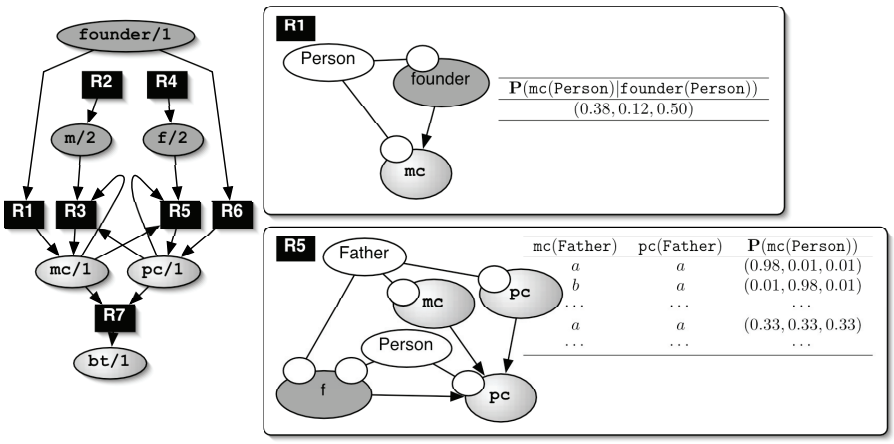
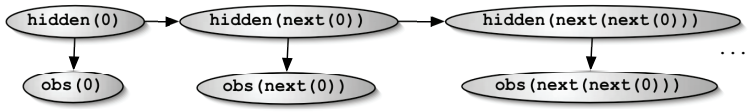


Figure 1.10 The *blood type* Bayesian logic program distinguishing between *Bayesian* (gradient gray ovals) and *logical* atoms (solid gray ovals).

recognition, empirical natural language processing, and robotics.
At each `Time`, the system is in a state `hidden(Time)`. The time-independent probability of being in some state at the next time `next(Time)` given that the system was in a state at `TimePoint` is captured in the Bayesian clause $R2$. Here, the next time point is represented as functor `next/1` . In HMMs, however, we do not have direct access to the states `hidden(Time)`. Instead, we measure some properties `obs(Time)` of the states. The measurement is quantified in Bayesian clause $R3$. The dependency graph of the Bayesian logic program directly encodes the well-known Bayesian network structure of HMMs:



1.4.2 Logical Atoms

Reconsider the *blood type* Bayesian logic program in Figure 1.8. The `mother/2` and `father/2` relations are not really *random* variables but *logical* ones because they are always in the same state, namely *true*, with probability 1 and can depend only on other logical atoms. These predicates form a kind of logical background theory. Therefore, when predicates are declared to be *logical*, one need not to represent them in the conditional probability distributions. Consider the *blood type* Bayesian logic program in Figure 1.10. Here, `mother/2` and `father/2` are declared to be *logical*. Consequently, the conditional probability distribution associated with the definition of e.g. `pc/1` takes only `pc(Father)` and `mc(Father)` into account but not `f(Father,Person)`. It applies only to those substitutions for which

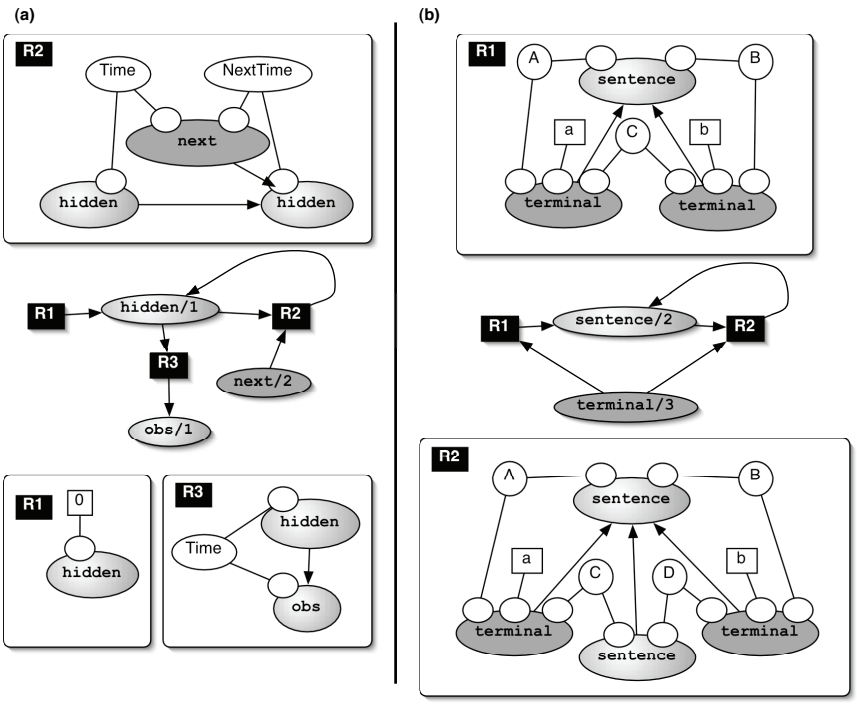


Figure 1.11 Two dynamic Bayesian logic programs. **(a)** The generic structure of a hidden Markov model more elegantly represented as in Figure 1.9 using $\texttt{next}(\texttt{X},\texttt{Y}) : -\texttt{integer}(\texttt{Y}), \texttt{Y} > 0, \texttt{X is Y} - 1.$. **(b)** A probabilistic context-free grammar over $\{a^n b^n\}$. The logical background theory defines `terminal/3` as $\texttt{terminal}([\texttt{A}|\texttt{B}], \texttt{A}, \texttt{B})$.

$\texttt{f}(\texttt{Father}, \texttt{Person})$ is true, i.e., in the least Herbrand model. This can efficiently be checked using any Prolog engine. Furthermore, one may omit these logical atoms from the induced support network. More importantly, logical predicates provide the user with the full power of Prolog. In the *blood type* Bayesian logic program of Figure 1.10, the logical background knowledge defines the `founder/1` relation as

$$\texttt{founder}(\texttt{Person}) \texttt{:-} \backslash + (\texttt{mother}(\_, \texttt{Person}); \texttt{father}(\_, \texttt{Person})).$$

Here, $\backslash +$ denotes *negation*, the symbol _ represents an anonymous variable which is treated as new, distinct variable each time it is encountered, and the semicolon denotes a *disjunction*. The rest of the Bayesian logic program is essentially as in Figure 1.4. Instead of explicitly listing $\texttt{pc}(\texttt{ann}), \texttt{mc}(\texttt{ann}), \texttt{pc}(\texttt{brian}), \texttt{mc}(\texttt{brian})$ in the extensional part we have $\texttt{pc}(\texttt{P})|\texttt{founder}(\texttt{P})$ and $\texttt{mc}(\texttt{P})|\texttt{founder}(\texttt{P})$ in the intensional part.

The full power of Prolog is also useful to elegantly encode dynamic probabilistic models. Figure 1.11 **(a)** shows the generic structure of an HMM where the discrete

time is now encoded as `next/2` in the logical background theory using standard Prolog predicates:

$$\texttt{next(X,Y)}\texttt{:-integer(Y)}, \texttt{Y} > 0, \texttt{X is Y} - 1.$$

Prolog's predefined predicates (such as `integer/1`) avoid a cumbersome representation of the dynamics via the successor functor $0, \texttt{next}(0), \texttt{next}(\texttt{next}(0)), \ldots$ Imagine querying `?- obs(100)` using the successor functor,

$$\texttt{?- obs(next(next(}\ldots\texttt{(next(0))}\ldots\texttt{)))} \; .$$

Whereas HMMs define probability distributions over regular languages, *probabilistic context-free grammars* (PCFGs) [Manning and Schütze, 1999] define probability distributions over context-free languages. Application areas of PCFGs include e.g. natural language processing and computational biology. For instance, mRNA sequences constitute context-free languages. Consider e.g. the following PCFG

$$\begin{aligned} & \texttt{terminal([A|B],A,B).} \\ 0.3 \; : \; & \texttt{sentence(A,B)}\texttt{:-terminal(A,a,C),terminal(C,b,B).} \\ 0.7 \; : \; & \texttt{sentence(A,B)}\texttt{:-terminal(A,a,C),sentence(C,D),terminal(D,b,B).} \end{aligned}$$

defining a distribution over $\{a^n b^n\}$. The grammar is represented as probabilistic *definite clause grammar* where the terminal symbols are encoded in the logical background theory via the first rule `terminal([A|B],A,B)` .
A PCFG defines a stochastic process with leftmost rewriting, i.e., refutation steps as transitions. Words, say *aabb*, are parsed by querying `?- sentence([a,a,b,b],[])`. The third rule yields `?- terminal([a,a,b,b],a,C),sentence(C,D),terminal(D,b,[])`. Applying the first rule yields `?- sentence([a,b,b],D),terminal(D,b,[])` and the second rule `?- terminal([a,b,b],a,C),terminal(C,b,D),terminal(D,b,[])`. Applying the first rule three times yields a successful refutation. The probability of a refutation is the product of the probability values associated with clauses used in the refutation; in our case $0.7 \cdot 0.3$. The probability of *aabb* then is the sum of the probabilities of all successful refutations. This is also the basic idea underlying Muggleton's *stochastic logic programs* 1996 which extend the PCFGs to definite clause logic.
Figure 1.11 **(b)** shows the $\{a^n b^n\}$ PCFG represented as Bayesian logic program. The Bayesian clauses are the clauses of the corresponding *definite clause grammar*. In contrast to PCFGs, however, we associate a complete conditional probability distribution, namely $(0.3, 0.7)$ and $(0.7, 0.3; 0.0, 1.0)$ to the Bayesian clauses. For the query `?- sentence([a,a,b,b],[])`, the following Markov chain is induced (omitting logical atoms):

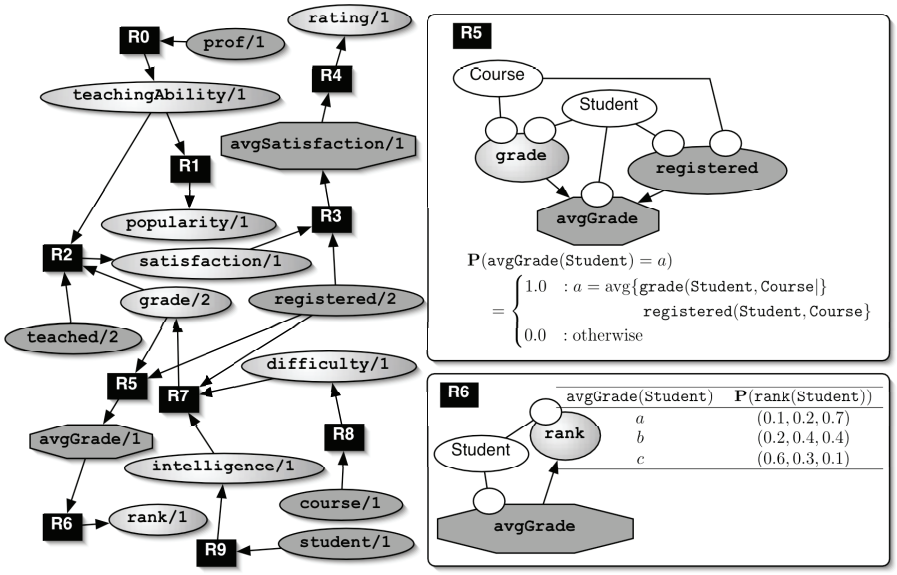


Figure 1.12 The Bayesian logic program for the *university* domain. Octagonal nodes denote aggregate predicates and atoms.

1.4.3 Aggregate Functions

An alternative to combining rules are *aggregate functions*. Consider the *university* domain due to Getoor et al. [2001]. The domain is that of a university, and contains professors, students, courses, and course registrations. Objects in this domain have several descriptive attributes such as `intelligence/1` and `rank/1` of a `student/1`. A student will typically be registered in several courses; the student's rank depends on the grades she receives in all of them. So, we have to specify a probabilistic dependence of the student's rank on a multiset of course grades of size 1, 2, and so on.

In this situation, the notion of aggregation is more appropriate than that of a combining rule. Using combining rules, the Bayesian clauses would describe the dependence for a single course only. All information of how the rank probabilistically depends on the multiset of course grades would be 'hidden' in the combining rule. In contrast, when using an aggregate function, the dependence is interpreted as a probabilistic dependence of `rank` on some *deterministically* computed aggregate property of the multiset of course grades. The probabilistic dependence is moved out of the combining rule.

To model this, we introduce *aggregate* predicates. They represent deterministic random variables, i.e., the state of an aggregate atom is a function of the joint state of its parents. As an example, consider the *university* Bayesian logic program as shown in Figure 1.12. Here, `avgGrade/1` is an aggregate predicate, denoted

as an octagonal node. As combining rule, the average of the parents' states is deterministically computed, cf. Bayesian clause $R5$. In turn, the student's `rank/1` probabilistically depends on her averaged rank, cf. $R6$.
The use of aggregate functions is inspired by *probabilistic relational models* [Pfeffer, 2000]. As we will show in the related work section, using aggregates in Bayesian logic programs, it is easy to model probabilistic relational models.

1.5 Learning Bayesian Logic Programs

When designing Bayesian logic programs, the expert has to determine the structure of the Bayesian logic program by specifying the extensional and intensional predicates, and by providing definitions for each of the intensional predicates. Given this logical structure, the Bayesian logic program induces a Bayesian network whose nodes are the *relevant* random variables. It is well-known that determining the structure of a Bayesian network, and therefore also of a Bayesian logic program, can be difficult and expensive. On the other hand, it is often easier to obtain a set $D = \{D_1, \ldots, D_m\}$ of data cases, which can be used for learning.

1.5.1 The Learning Setting

For Bayesian logic programs, a data case $D_i \in D$ has two parts, a logical and a probabilistic part. The logical part of a data case is a Herbrand interpretation. For instance, the following set of atoms constitute a Herbrand interpretation for the *blood type* Bayesian logic program.

$$\{\mathtt{m(ann,dorothy)}, \mathtt{f(brian,dorothy)}, \mathtt{pc(ann)}, \mathtt{mc(ann)}, \mathtt{bt(ann)},$$
$$\mathtt{pc(brian)}, \mathtt{mc(brian)}, \mathtt{bt(brian)}, \mathtt{pc(dorothy)}, \mathtt{mc(dorothy)}, \mathtt{bt(dorothy)}\}$$

This (logical) interpretation can be seen as the least Herbrand model of an unknown Bayesian logic program. In general, data cases specify different sets of *relevant* random variables, depending on the given "extensional context". If we accept that the genetic laws are the same for different families, then a learning algorithm should transform such extensionally defined predicates into intensionally defined ones, thus compressing the interpretations. This is precisely what *inductive logic programming* techniques [Muggleton and De Raedt, 1994] do. The key assumption underlying any inductive technique is that the rules that are valid in one interpretation are likely to hold for other interpretations. It thus seems clear that techniques for *learning from interpretations* can be adapted for learning the logical structure of Bayesian logic programs.
So far, we have specified the logical part of the learning problem: we are looking for a set H of Bayesian clauses given a set D of data cases such that all data cases are a model of H. The hypotheses H in the space $\mathcal{H}$ of hypotheses are sets of Bayesian clauses. However, we have to be more careful. A candidate set $H \in \mathcal{H}$ has to be

acyclic on the data which implies that for each data case the induced Bayesian network has to be acyclic.
Let us now focus on the quantitative components. The quantitative component of a Bayesian logic program is given by the associated conditional probability distributions and combining rules. For the sake of simplicity, we assume that the combining rules are fixed. Each data case $D_i \in D$ has a probabilistic part which is a partial assignment of states to the random variables in D_i. As an example consider the following data case:

$$\{\mathtt{m}(\mathtt{ann},\mathtt{dorothy}) = true, \mathtt{f}(\mathtt{brian},\mathtt{dorothy}) = true, \mathtt{pc}(\mathtt{ann}) = a, \mathtt{mc}(\mathtt{ann}) = a,$$
$$\mathtt{bt}(\mathtt{ann}) = a, \mathtt{pc}(\mathtt{brian}) = a, \mathtt{mc}(\mathtt{brian}) = b, \mathtt{bt}(\mathtt{brian}) = ab,$$
$$\mathtt{pc}(\mathtt{dorothy}) = b, \mathtt{mc}(\mathtt{dorothy}) = ?, \mathtt{bt}(\mathtt{dorothy}) = ab\}$$

where ? denotes an unknown state of a random variable. The partial assignments induce a joint distribution over the random variables. A candidate $H \in \mathcal{H}$ should reflect this distribution. In Bayesian networks the conditional probability distributions are typically learned using gradient descent or EM for a fixed structure of the Bayesian network. A scoring function $score_D(H)$ that evaluates how well a given structure $H \in \mathcal{H}$ matches the data is maximized.
To summarize, the learning problem is a probabilistic extension of the *learning from interpretations* setting from inductive logic programming and can be formulated as follows:

Given a set D of data cases, a set $\mathcal{H}$ of Bayesian logic programs and a scoring function $score_D$.

Find a candidate $H^* \in \mathcal{H}$ which is acyclic on the data cases such that the data cases $D_i \in D$ are models of H^* (in the logical sense) and H^* matches the data D best according to $score_D$.

Here, the best match refers to those parameters of the associated conditional probability distributions which maximize the scoring function.
The learning setting provides an interesting link between inductive logic programming and Bayesian network learning as we will show in the next section.

1.5.2 Maximum Likelihood Learning

Consider the task of performing *maximum likelihood* learning, i.e., $score_D(H) = \mathbf{P}(D|H)$. As in many cases, it is more convenient to work with the logarithm of this function, i.e., $score_D(H) = LL(D,H) := \log \mathbf{P}(D|H)$. It can be shown [see Kersting and De Raedt, 2001a, for more details] that the likelihood of a Bayesian logic program coincides with the likelihood of the support network induced over D. Thus, learning of Bayesian logic programs basically reduces to learning Bayesian networks. The main differences are the ways to estimate the parameters and to traverse the hypotheses space.

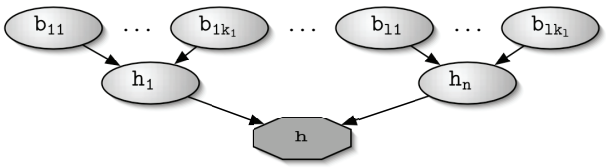


Figure 1.13 Decomposable combining rules can be expressed within support networks. The nodes $\mathtt{h_i}$ have the domain of $\mathtt{h}$ and cpd(c) associated. The node $\mathtt{h}$ becomes a deterministic node, i.e., its parameters are fixed. For example for *noisy_or*, logical *or* is associated as function with $\mathtt{h}$. Note that the $\mathtt{h_i}$'s are never observed; only $\mathtt{h}$ might be observed.

1.5.2.1 Parameter Estimation

The parameters of non-ground Bayesian clauses have to be estimated. In order to adapt techniques traditionally used for parameter estimation of Bayesian networks such as the EM algorithm [Dempster et al., 1977], combining rules are assumed to be *decomposable* [3] [Heckerman and Breese, 1994]. Decomposable combining rules can be completely expressed by adding extra nodes to the induced support network, cf. Figure 1.13. These extra nodes are copies of the (ground) head atom which becomes a deterministic node. Now, each node in the support network is "produced" by exactly one Bayesian clause c, and each node derived from c can be seen as a separate "experiment" for the conditional probability distribution cpd(c). Therefore, the EM estimates the improved parameters as the following ratio:

$$\frac{\sum_{l=1}^{m} \sum_{\theta} \mathbf{P}(\mathrm{head}(c\theta), \mathrm{body}(c\theta) \mid D_l)}{\sum_{l=1}^{m} \sum_{\theta} \mathbf{P}(\mathrm{body}(c\theta) \mid D_l)}$$

where θ denotes substitutions such that D_l is a model of $c\theta$.

1.5.2.2 Traversing the Hypotheses Space

Instead of adding, deleting, or flipping single edges in the support network, we employ refinement operators traditionally used in inductive logic programming to add, delete, or flip several edges in the support network at the same time. More specifically, according to some language bias — say, we consider only functor-free and constants-free clauses — we use the two refinement operators $\rho_s : 2^{\mathcal{H}} \mapsto \mathcal{H}$ and $\rho_g : 2^{\mathcal{H}} \mapsto \mathcal{H}$. The operator $\rho_s(H)$ adds constant-free atoms to the body of a single clause $c \in H$, and $\rho_g(H)$ deletes constant-free atoms from the body of a single clause $c \in H$. Other refinement operators such as deleting and adding logically valid clauses, instantiating variables, and unifying variables are possible,

3. Most combining rules commonly employed in Bayesian networks such as *noisy_or* are decomposable.

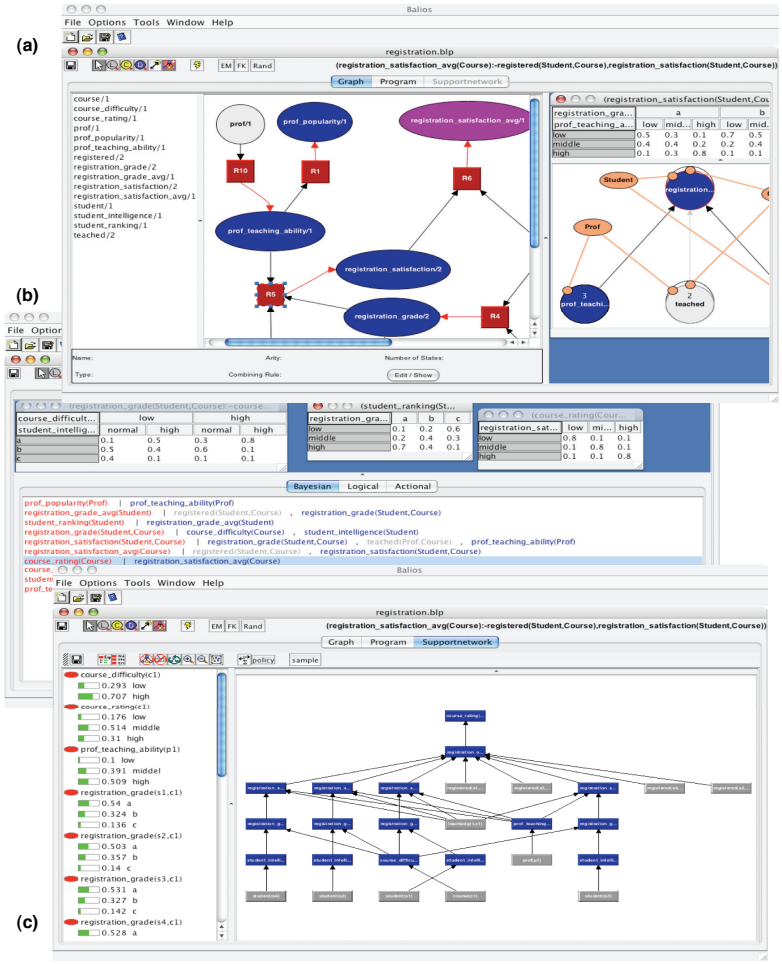


Figure 1.14 BALIOS - the engine for Bayesian logic programs. **(a)** Graphical representation of the *university* Bayesian logic program. **(b)** Textual representation of Bayesian clauses with associated conditional probability distributions. **(c)** Computed support network and probabilities for a probabilistic query.

too, cf. [Nienhuys-Cheng and de Wolf, 1997].
Combining these ideas, a basic greedy hill-climbing algorithm for learning Bayesian logic programs can be sketched as follows. Assuming some data cases D, we take some H_0 as starting point (for example computed using some standard inductive logic programming system) and compute the parameters maximizing $LL(D, H)$. Then, we use $\rho_s(H)$ and $\rho_g(H)$ to compute the legal 'neighbours' of H in $\mathcal{H}$ and score them. If $LL(D, H) < LL(D, H')$, then we take H' as new hypothesis. The process is continued until no further improvements in score are obtained.

1.6 Balios – The Engine for BLPs

An engine for Bayesian logic programs featuring a graphical representation, logical atoms, and aggregate functions has been implemented in the BALIOS system Kersting and Dick [2004], which is freely available for academic use at `http://www.informatik.uni-freiburg.de/~kersting/achilleus/`. BALIOS is written in JAVA. It calls SICSTUS Prolog to perform logical inference and a BN inference engine to perform probabilistic inference. BALIOS features a GUI graphically representing BLPs, cf. Figure 1.14, computing the most likely configuration, approximative inference methods (rejection, likelihood and Gibbs sampling), and parameter estimation methods (hard EM, EM, and conjugate gradient).

1.7 Related Work

In the last ten years, there has been a lot of work lying at the intersection of probability theory, logic programming and machine learning [Poole, 1993, Haddawy, 1994, Sato, 1995, Muggleton, 1996, Ngo and Haddawy, 1997, Jaeger, 1997, Koller and Pfeffer, 1998, Anderson et al., 2002, Kersting et al., 2003, Domingos and Richardson, 2004], see [De Raedt and Kersting, 2003] for an overview. Instead of giving a probabilistic characterization of logic programming such as Ng and Subrahmanian [1992], this research highlights the machine learning aspect and is known under the names of *statistical relational learning* (SRL) [Getoor and Jensen, 2003, Dietterich et al., 2004], probabilistic logic learning (PLL) [De Raedt and Kersting, 2003], or probabilistic inductive logic programming (PILP) [De Raedt and Kersting, 2004]. Bayesian logic programs belong to the SRL line of research which extends Bayesian networks. They are motivated and inspired by the formalisms discussed in [Poole, 1993, Haddawy, 1994, Ngo and Haddawy, 1997, Jaeger, 1997, Friedman et al., 1999, Koller, 1999]. We will now investigate these relationships in more details.

Probabilistic logic programs [Ngo and Haddawy, 1995, 1997] also adapt a logic program syntax, the concept of the least Herbrand model to specify the relevant random variables, and SLD resolution to develop a query-answering procedure. Whereas Bayesian logic programs view atoms as random variables, probabilistic-logic programs view them as states of random variables. For instance,

$$P(\texttt{burglary}(\texttt{Person}, \texttt{yes}) \mid \texttt{neighbourhood}(\texttt{Person}, \texttt{average})) = 0.4$$

states that the aposteriori probability of a burglary in `Person`'s house given that `Person` has an average neighbourhood is 0.4. Thus, instead of conditional probability distributions, conditional probability values are associated with clauses. Treating atoms as states of random variables has several consequences: (1) Exclu-

sivity constraints such as

$$\texttt{false} \leftarrow \texttt{neighbourhood(X, average)}, \texttt{neighbourhood(X, bad)}$$

have to be specified in order to guarantee that random variables are always in exactly one state. (2) The inference procedure is exponentially slower in time for building the support network than that for Bayesian logic programs because there is a proof for each configuration of random variable. (3) It is more difficult — if not impossible — to represent continuous random variables. (4) Qualitative, i.e., the logical component, and quantitative information, i.e., the probability values, are mixed. Just this separation of both information made the graphical representation for Bayesian logic programs possible.

Probabilistic and Bayesian logic programs are also related to Poole's framework of **probabilistic Horn abduction** [1993], which is "*a pragmatically-motivated simple logic formulation that includes definite clauses and probabilities over hypotheses*" [Poole, 1993]. Poole's framework provides a link to abduction and assumption-based reasoning. However, as Ngo and Haddawy point out, probabilistic and therefore also Bayesian logic programs have not as many constraints on the representation language, represent probabilistic dependencies directly rather than indirectly, have a richer representational power, and their independence assumption reflects the causality of the domain.

Koller et. al. [Friedman et al., 1999, Koller, 1999] define **probabilistic relational models**, which are based on the well-known entity/relationship model. In probabilistic relational models, the random variables are the attributes. The relations between entities are deterministic, i.e. they are only true or false. Probabilistic relational models can be described as Bayesian logic programs.

Indeed, each attribute a of an entity type E is a Bayesian predicate $\texttt{a(E)}$ and each n-ary relation r is an n-ary logical Bayesian predicate $\texttt{r/n}$. Probabilistic relational models consist of a qualitative dependency structure over the attributes and their associated quantitative parameters (the conditional probability densities). Koller et. al. distinguish between two types of parents of an attribute. First, an attribute $\texttt{a(X)}$ can depend on another attribute $\texttt{b(X)}$, e.g. the professor's popularity depends on the professor's teaching ability in the *university* domain. This is equivalent to the Bayesian clause $\texttt{a(X)} \mid \texttt{b(X)}$. Second, an attribute $\texttt{a(X)}$ possibly depends on an attribute $\texttt{b(Y)}$ of an entity $\texttt{Y}$ related to $\texttt{X}$, e.g. a student's grade in a course depends on the difficulty of the course. The relation between $\texttt{X}$ and $\texttt{Y}$ is described by a slot or logical relation $\texttt{s(X, Y)}$. Given these logical relation, the original dependency is represented by $\texttt{a(X)} \mid \texttt{s(X, Y)}, \texttt{b(Y)}$. To deal with multiple ground instantiations of a single clause (with the same head ground atom), probabilistic relational models employ aggregate functions as discussed earlier.

Clearly, probabilistic relational models employ a more restricted logical component than Bayesian logic programs do: it is a version of the commonly used entity/relationship model. Any entity/relationship model can be represented using a (range-restricted) definite clause logic. Furthermore, several extensions to treat *exis-*

tential uncertainty, *referential uncertainty*, and *domain uncertainty* exists. Bayesian logic programs have the full expressivity of definite clause logic and, therefore, of a universal Turing machine. Indeed, general definite clause logic (using functors) is undecidable. The functor-free fragment of definite clause logic, however, is decidable.

Jäger [1997] introduced **relational Bayesian networks**. They are Bayesian networks where the nodes are predicate symbols. The states of these random variables are possible interpretations of the symbols over an arbitrary, finite domain (here we only consider Herbrand domains), i.e. the random variables are set-valued. The inference problem addressed by Jäger asks for the probability that an interpretation contains a ground atom. Thus, relational Bayesian networks are viewed as Bayesian networks where the nodes are the ground atoms and have the domain $\{true, false\}$[4]. The key difference between relational Bayesian networks and Bayesian logic programs is that the quantitative information is specified by so called probability formulas. These formulas employ the notion of combination functions, functions that map every finite multiset with elements from $[0, 1]$ into $[0, 1]$, as well as that of equality constraints[5]. Let $F(cancer)(x)$ be $noisy\_or\{comb_{\Gamma}\{exposed(x, y, z) \mid z; true\} \mid y; true\}$. This formula states that that for any specific organ y, multiple exposures to radiation have a cumulative but independent effect on the risk of developing cancer of y. Thus, a probability formula not only specifies the distribution but also the dependency structure. Therefore and because of the computational power of combining rules, a probability formula is easily expressed as a set of Bayesian clauses: the head of the Bayesian clauses is the corresponding Bayesian atom and the bodies consist of all maximally generalized Bayesian atoms occurring in the probability formula. Now the combining rule can select the right ground atoms and simulate the probability formula. This is always possible because the Herbrand base is finite. E.g. the clause `cancer(X) | exposed(X,Y,Z)` together with the right combining rule and associated conditional probability distribution models the example formula.

In addition to extensions of Bayesian networks, several other probabilistic models have been extended to the first-order or relational case: Sato [1995] introduces *distributional semantics* in which ground atoms are seen as random variables over $\{true, false\}$. Probability distributions are defined over the ground facts of a program and propagated over the Herbrand base of the program using the clauses. Stochastic logic programs [Muggleton, 1996, Cussens, 1999] introduced by Muggleton lift context-free probabilistic grammars to the first order case. Production rules are replaced by clauses labeled with probability values. Recently, Domingos and Richardson [2004] introduced Markov logic networks which upgrade Markov networks to the first order case. The features of the Markov logic network are weights attached to first-order predicate logic formulas. The weights specify a

4. It is possible, but complicated to model domains having more than two values.
5. To simplify the discussion, we will further ignore these equality constraints here.

bias for ground instances to be true in a logical model.
Finally, Bayesian logic programs are related — to some extent — to the **BUGS** language [Gilks et al., 1994] which aims at carrying out Bayesian inference using Gibbs sampling. It uses concepts of imperative programming languages such as for-loops to model regularities in probabilistic models. Therefore, the relation between Bayesian logic programs and BUGS is akin to the general relation between logical and imperative languages. This holds in particular for relational domains such as those used in this chapter. Without the notion of objects and relations among objects family trees are hard to represent: BUGS uses traditional indexing to group together random variables (e.g. X_1, X_2, X_3 ... all having the same distribution), whereas Bayesian logic programs use definite clause logic.

1.8 Conclusions

We have described Bayesian logic programs, their representation language, their semantics, and a query-answering process, and briefly touched upon learning Bayesian logic programs from data.
Bayesian logic programs combine Bayesian networks with definite clause logic. The main idea of Bayesian logic programs is to establish a one-to-one mapping between ground atoms in the least Herbrand model and random variables. The least Herbrand model of a Bayesian logic program together with its direct influence relation is viewed as a (possibly infinite) Bayesian network. Bayesian logic programs inherit the advantages of both Bayesian networks and definite clause logic, including the strict separation of qualitative and quantitative aspects. Moreover, the strict separation facilitated the introduction of a graphical representation, which stays close to the graphical representation of Bayesian networks.
Indeed, Bayesian logic programs can naturally model any type of Bayesian network (including those involving continuous variables) as well as any type of "pure" Prolog program (including those involving functors). We also demonstrated that Bayesian logic programs can model hidden Markov models and stochastic grammars, and investigated their relationship to other first order extensions of Bayesian networks.
We have also presented the Balios tool, which employs the graphical as well as the logical notations for Bayesian logic programs. It is available at

`http://www.informatik.uni-freiburg.de/~kersting/achilleus/`,

and the authors would like to invite the reader to employ it.

Acknowledgements

The authors would like to thank Uwe Dick for implementing the Balios system. This research was partly supported by the European Union IST programme under contract number IST-2001-33053 and FP6-508861, APRIL I & II (**A**pplication of **Pr**obabilistic **I**nductive **L**ogic Programming).

References

C. R. Anderson, P. Domingos, and D. S. Weld. Relational Markov Models and their Application to Adaptive Web Navigation. In D. Hand, D. Keim, O. R. Zaïne, and R. Goebel, editors, *Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining (KDD-02)*, pages 143–152, Edmonton, Canada, 2002. ACM Press.

Heinz Bauer. *Wahrscheinlichkeitstheorie*. Walter de Gruyter, Berlin, New York, 4. edition, 1991.

J. Cheng, C. Hatzis, M.–A. Krogel, S. Morishita, D. Page, and J. Sese. KDD Cup 2002 Report. *SIGKDD Explorations*, 3(2):47 – 64, 2002.

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

L. De Raedt and K. Kersting. Probabilistic Logic Learning. *ACM-SIGKDD Explorations: Special issue on Multi-Relational Data Mining*, 5(1):31–48, 2003.

L. De Raedt and K. Kersting. Probabilistic Inductive Logic Programming. In S. Ben-David, J. Case, and A. Maruoka, editors, *Proceedings of the 15th International Conference on Algorithmic Learning Theory (ALT-2004)*, pages 19–36, Padova, Italy, October 2–5 2004.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Stat. Soc.*, B 39:1–39, 1977.

T. Dietterich, L. Getoor, and K. Murphy, editors. *Working Notes of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields (SRL-04)*, 2004.

P. Domingos and M. Richardson. Markov Logic: A Unifying Framework for Statistical Relational Learning. In *Proceedings of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields*, pages 49–54, 2004.

N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conferences on Artificial Intelligence (IJCAI-99)*, pages 1300–1309, Stockholm, Sweden, 1999. Morgan Kaufmann.

L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. *Relational Data Mining*, chapter

Learning Probabilistic Relational Models. S. Džeroski and N. Lavrač, 2001. (to appear).

L. Getoor and D. Jensen, editors. *Working Notes of the IJCAI-2003 Workshop on Learning Statistical Models from Relational Data (SRL-03)*, 2003.

W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex bayesian modelling. *The Statistician*, 43, 1994.

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.

D. Heckerman and J. Breese. Causal Independence for Probability Assessment and Inference Using Bayesian Networks. Technical Report MSR-TR-94-08, Microsoft Research, 1994.

D. Heckerman, A. Mamdani, and M. P. Wellman. Real-world applications of Bayesian networks. *Communications of the ACM*, 38(3):24–26, March 1995.

M. Jaeger. Relational Bayesian networks. In D. Geiger and P. P. Shenoy, editors, *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 266–273, Providence, Rhode Island, USA, 1997. Morgan Kaufmann.

F. V. Jensen. *An introduction to Bayesian Networks*. UCL Press Limited, 1996. Reprinted 1998.

F. V. Jensen. *Bayesian networks and decision graphs*. Springer-Verlag, 2001.

M. I. Jordan, editor. *Learning in Graphical Models*. MIT Press, 1998.

K. Kersting and L. De Raedt. Adaptive Bayesian Logic Programs. In C. Rouveirol and M. Sebag, editors, *Proceedings of the Eleventh Conference on Inductive Logic Programming (ILP-01)*, volume 2157 of *LNCS*, Strasbourg, France, 2001a. Springer.

K. Kersting and L. De Raedt. Bayesian logic programs. Technical Report 151, University of Freiburg, Institute for Computer Science, April 2001b.

K. Kersting and U. Dick. Balios - The Engine for Bayesian Logic Programs. In J.-F. Boulicaut, F. Esposito anf F. Giannotti, and D. Pedreschi, editors, *Proceedings of the 8th European Conference on Principles and Practice of Knowledege Discovery in Databases (PKDD-2004)*, pages 549–551, Pisa, Italy, September 20-25 2004.

K. Kersting, T. Raiko, S. Kramer, and L. De Raedt. Towards discovering structural signatures of protein folds based on logical hidden markov models. In R. B. Altman, A. K. Dunker, L. Hunter, T. A. Jung, and T. E. Klein, editors, *Proceedings of the Pacific Symposium on Biocomputing*, pages 192 – 203, Kauai, Hawaii, USA, 2003. World Scientific.

D. Koller. Probabilistic relational models. In S. Džeroski and P. Flach, editors, *Proceedings of Ninth International Workshop on Inductive Logic Programming (ILP-99)*, volume 1634 of *LNAI*, pages 3–13, Bled, Slovenia, 1999. Springer.

D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 580–587, Madison, Wisconsin, USA, July 1998. AAAI Press.

P. Langley. *Elements of Machine Learning*. Morgan Kaufmann, 1995.

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

C. H. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

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

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

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

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.

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

S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer, 1997.

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

A. J. Pfeffer. *Probabilistic Reasoning for Complex Systems*. PhD thesis, Stanford University, 2000.

D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

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

T. Sato. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP-1995)*, pages pp. 715 – 729, 1995.

T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

R. D. Schachter. Bayes-Ball: The Rational Pasttime (for Determining Irrelevance and Requisite Information in Belief Networks and Influence Diagrams). In G. F. Cooper and S. Moral, editors, *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 480–487, Madison, Wisconsin, USA, 1998. Morgan Kaufmann.