

---

# 10 Bayesian Logic Programming: Theory and Tool

*Kristian Kersting and Luc De Raedt*

Bayesian networks provide an elegant formalism for representing and reasoning about uncertainty. They are a probabilistic extension of propositional logic and, hence, inherit some of the limitations of propositional logic, such as the difficulties with representing objects and relations. In this chapter, we introduce Bayesian logic programs, which are an extension of Bayesian networks to overcome these limitations. Bayesian logic programs tightly integrate definite logic programs with Bayesian networks. The key idea underlying 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 dependency relation. In doing so, Bayesian logic programs combine the advantages of both definite clause logic and Bayesian networks: notions of objects and relations, a separation of quantitative and qualitative aspects of the world, and a graphical representation.

---

## 10.1 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 [5] for an overview). Muggleton [30] and Cussens [4] have upgraded stochastic grammars toward *stochastic logic programs*, Sato and Kameya [42] have introduced *probabilistic distributional semantics* for logic programs, and Domingos and Richardson [9] have upgraded Markov networks toward *Markov logic networks*. Another research stream including Poole's *independent choice logic* [38], Ngo and Haddawy's *Probabilistic-Logic Programs* [34], Jaeger's *relational Bayesian networks* [17], and Pfeffer's *probabilistic relational models* [37] concentrates on first-order logical and relational extensions of Bayesian networks.

Bayesian networks [36] 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 [16]. 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 modeling the localization of genes/proteins as was the task at the KDD Cup 2001 [3]. 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

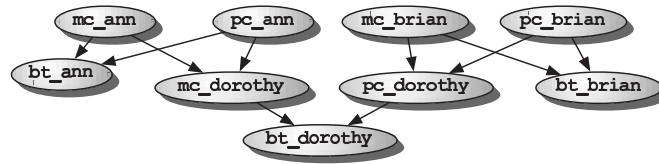
*the localization L of gene G is influenced by the localization L' of another gene G' that interacts with G.*

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 toward 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 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;
- 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.



**Figure 10.1** The graphical structure of a Bayesian network modeling the inheritance of blood types within a particular family.

The chapter is structured as follows. We begin by briefly reviewing Bayesian networks and logic programs in section 10.2. In section 10.3, we define Bayesian logic programs as well as their semantics. Afterward, in section 10.4, 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 the effective treatment of logic atoms and of aggregate functions. In section 10.5, 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.

---

## 10.2 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 then show how these are combined in Bayesian logic programs. For a full and detailed treatment of each of these topics, we refer to [28] for logic programming or Prolog and to [18] for Bayesian networks.

We introduce Bayesian logic programs using an example from genetics which is inspired by [10]:

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

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

### 10.2.1 Bayesian Networks

A *Bayesian network* [36] is an augmented, directed acyclic graph, where each node corresponds to a random variable  $x_i$  and each edge indicates a *direct in-*

*fluence* among the random variables. It represents the joint probability distribution  $\mathbf{P}(\mathbf{x}_1, \dots, \mathbf{x}_n)$  over a fixed, finite set  $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$  of random variables. Each random variable  $\mathbf{x}_i$  possesses a finite set  $\mathbf{S}(\mathbf{x}_i)$  of mutually exclusive states. Figure 10.1 shows the graph of a Bayesian network modeling our blood type example for a particular family. The familial relationship, which is taken from Jensen's *stud farm* example [19], 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  $\mathbf{bt}(\mathbf{dorothy})$  is  $\mathbf{S}(\mathbf{bt}(\mathbf{dorothy})) = \{a, b, ab, 0\}$ ; the set of possible states of  $\mathbf{pc}(\mathbf{dorothy})$  and  $\mathbf{mc}(\mathbf{dorothy})$  are  $\mathbf{S}(\mathbf{pc}(\mathbf{dorothy})) = \mathbf{S}(\mathbf{mc}(\mathbf{dorothy})) = \{a, b, 0\}$ . The same holds for **ann** and **brian**. The direct predecessors of a node  $\mathbf{x}$ , the parents of  $\mathbf{x}$ , are denoted by  $\mathbf{Pa}(\mathbf{x})$ . For instance,  $\mathbf{Pa}(\mathbf{bt}(\mathbf{ann})) = \{\mathbf{pc}(\mathbf{ann}), \mathbf{mc}(\mathbf{ann})\}$ .

A Bayesian network stipulates the following conditional independence assumption.

**Proposition 10.1 Independence Assumption of Bayesian Networks**

Each node  $\mathbf{x}_i$  in the graph is conditionally independent of any subset  $\mathbf{A}$  of nodes that are not descendants of  $\mathbf{x}_i$  given a joint state of  $\mathbf{Pa}(\mathbf{x}_i)$ , i.e.,

$$\mathbf{P}(\mathbf{x}_i \mid \mathbf{A}, \mathbf{Pa}(\mathbf{x}_i)) = \mathbf{P}(\mathbf{x}_i \mid \mathbf{Pa}(\mathbf{x}_i)) .$$

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

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

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

$$\mathbf{P}(\mathbf{x}_1, \dots, \mathbf{x}_n) = \prod_{i=1}^n \mathbf{P}(\mathbf{x}_i \mid \mathbf{Pa}(\mathbf{x}_i))$$

by applying the independence assumption 10.1 to the chain rule expression of the joint probability distribution. Thereby, we associate with each node  $\mathbf{x}_i$  of the graph the conditional probability distribution  $\mathbf{P}(\mathbf{x}_i \mid \mathbf{Pa}(\mathbf{x}_i))$ , denoted as  $\mathbf{cpd}(\mathbf{x}_i)$ . The conditional probability distributions in our blood type domain are:

$\mathbf{mc}(\mathbf{dorothy})$	$\mathbf{pc}(\mathbf{dorothy})$	$\mathbf{P}(\mathbf{bt}(\mathbf{dorothy}))$
<i>a</i>	<i>a</i>	(0.97, 0.01, 0.01, 0.01)
<i>b</i>	<i>a</i>	(0.01, 0.01, 0.97, 0.01)
...	...	...
0	0	(0.01, 0.01, 0.01, 0.97)

(similarly for **ann** and **brian**) and

mc(ann)	pc(ann)	P(mc(dorothy))
<i>a</i>	<i>a</i>	(0.98, 0.01, 0.01)
<i>b</i>	<i>a</i>	(0.01, 0.98, 0.01)
...	...	...
0	0	(01, 0.01, 0.98)

(similarly for  $\text{pc(dorothy)}$ ). Further conditional probability tables are associated with the a priori nodes, i.e., the nodes having no parents:

P(mc(ann))	P(mc(ann))	P(mc(ann))	P(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)

### 10.2.2 Logic Programs

To introduce logic programs, consider figure 10.2, 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., **parent(jef,paul)**, **nat(s(X))**, **parent(X,Z)**, etc. We are now able to define the key concept of a (definite) *clause*. Clauses are formulae of the form  $A :- B_1, \dots, B_m$  where **A** and the **B<sub>i</sub>** are logical atoms where all variables are understood to be universally quantified. For example, the clause **grandparent(X,Y) :- parent(X,Z), parent(Z,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 **grandparent(X,Y)** the head(*c*) of this clause, and **parent(X,Z), parent(Z,Y)** the body(*c*). Clauses with an empty body, such as **parent(jef,paul)**, are called *facts*. A (definite) clause program (or *logic program* for short) consists of a set of clauses. In figure 10.2, there are thus two logic programs, one defining **grandparent/2** and one defining **nat/1**.

```

parent(jef,paul).
parent(paul,ann).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

nat(0).
nat(s(X)) :- nat(X).

```

**Figure 10.2** Two logic programs, *grandparent* and *nat*.

The set of variables in a term, atom, or clause *E* is denoted as  $\text{Var}(E)$ , e.g.,  $\text{Var}(c) = \{X, Y, Z\}$ . A term, atom, or clause *E* is called *ground* when there is no variable occurring in *E*, i.e.,  $\text{Var}(E) = \emptyset$ . A *substitution*  $\theta = \{V_1/t_1, \dots, V_n/t_n\}$ , e.g.,  $\{X/\text{ann}\}$ , is an assignment of terms  $t_i$  to variables  $V_i$ . Applying a substitution  $\theta$  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  $\text{grandparent}(\text{ann}, Y) :- \text{parent}(\text{ann}, Z), \text{parent}(Z, Y)$ .

The *Herbrand base* of a logic program  $T$ , denoted as  $\text{HB}(T)$ , is the set of all ground atoms constructed with the predicate, constant, and function symbols in the alphabet of  $T$ . For example,  $\text{HB}(\text{nat}) = \{\text{nat}(0), \text{nat}(\text{s}(0)), \text{nat}(\text{s}(\text{s}(0))), \dots\}$  and

$$\begin{aligned} \text{HB}(\text{grandparent}) = \\ \{ & \text{parent}(\text{ann}, \text{ann}), \text{parent}(\text{jef}, \text{jef}), \\ & \text{parent}(\text{paul}, \text{paul}), \text{parent}(\text{ann}, \text{jef}), \text{parent}(\text{jef}, \text{ann}), \dots, \\ & \text{grandparent}(\text{ann}, \text{ann}), \text{grandparent}(\text{jef}, \text{jef}), \dots \}. \end{aligned}$$

A *Herbrand interpretation* for a logic program  $T$  is a subset of  $\text{HB}(T)$ . The *least Herbrand model*  $\text{LH}(T)$  (which constitutes the semantics of the logic program) consists of all facts  $f \in \text{HB}(T)$  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

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

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

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

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

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

---

### 10.3 Bayesian Logic Programs

The logical component of Bayesian networks essentially corresponds to a propositional logic program.<sup>1</sup> Consider, for example, the program in figure 10.3. It encodes

---

1. Haddawy [14] and Langley [27] 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 10.3** A propositional clause program encoding the structure of the blood type Bayesian network in figure 10.1.

the structure of the blood type Bayesian network in figure 10.1. 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 probability 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.

### 10.3.1 Representation Language

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

**Definition 10.2 Bayesian Clause**

A *Bayesian (definite) clause*  $c$  is an expression of the form  $A \mid A_1, \dots, A_n$  where  $n \geq 0$ , the  $A, A_1, \dots, A_n$  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  $A$ .

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

1. the atoms  $p(t_1, \dots, t_1)$  and predicates  $p/1$  arising are Bayesian, which means that they have an associated (finite<sup>2</sup>) set  $S(p/1)$  of possible states, and
2. we use “ $\mid$ ” instead of “ $:-$ ” to highlight the conditional probability distribution.

---

2. For the sake of simplicity we consider finite random variables, i.e., random variables having a finite set  $S$  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).

For instance, consider the Bayesian clause  $c \text{ bt}(\mathbf{X}) | \text{mc}(\mathbf{X}), \text{pc}(\mathbf{X})$  where  $\mathbf{S}(\text{bt}/1) = \{a, b, ab, 0\}$  and  $\mathbf{S}(\text{mc}/1) = \mathbf{S}(\text{pc}/1) = \{a, b, 0\}$ . Intuitively, a Bayesian predicate  $\text{p}/l$  generically represents a set of random variables. More precisely, each Bayesian ground atom  $g$  over  $\text{p}/l$  represents a random variable over the states  $\mathbf{S}(g) := \mathbf{S}(\text{p}/l)$ . For example,  $\text{bt}(\text{ann})$  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.,  $\text{Var}(\text{head}(c)) \subseteq \text{Var}(\text{body}(c))$ . Range restriction is often imposed in the database literature; it allows one to avoid the derivation of nonground true facts (cf. section 10.2.2). As already indicated while discussing figure 10.3, 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  $\text{cpd}(c)$  encoding  $\mathbf{P}(\text{head}(c) \mid \text{body}(c))$ ; cf. figure 10.4. To keep the exposition simple, we will assume that  $\text{cpd}(c)$  is represented as a table. More elaborate representations such as decision trees or rules would be possible too. The distribution  $\text{cpd}(c)$  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  $c_2$

$$\begin{array}{l} \text{bt}(\mathbf{X}) \mid \text{mc}(\mathbf{X}). \\ \text{bt}(\mathbf{X}) \mid \text{pc}(\mathbf{X}). \end{array} \setminus ,$$

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

### Definition 10.3 Combining Rule

A combining rule is a function that maps finite sets of conditional probability distributions  $\{\mathbf{P}(\mathbf{A} \mid \mathbf{A}_{i1}, \dots, \mathbf{A}_{in_i}) \mid i = 1, \dots, m\}$  onto one (*combined*) conditional probability distribution  $\mathbf{P}(\mathbf{A} \mid \mathbf{B}_1, \dots, \mathbf{B}_k)$  with  $\{\mathbf{B}_1, \dots, \mathbf{B}_k\} \subseteq \bigcup_{i=1}^m \{\mathbf{A}_{i1}, \dots, \mathbf{A}_{in_i}\}$ .

We assume that for each Bayesian predicate  $\text{p}/l$  there is a corresponding combining rule  $\text{cr}(\text{p}/l)$ , such as *noisy-or* (see, e.g., [18]) or *average*. The latter assumes  $n_1 = \dots = n_m$  and  $\mathbf{S}(\mathbf{A}_{ij}) = \mathbf{S}(\mathbf{A}_{kj})$ , and computes the average of the distributions over  $\mathbf{S}(\mathbf{A})$  for each joint state over  $\bigotimes_j \mathbf{S}(\mathbf{A}_{ij})$ ; see also section 10.3.2.

By now, we are able to formally define Bayesian logic programs.



	mc(X)	pc(X)	P(bt(X))	
	<i>a</i>	<i>a</i>	(0.97, 0.01, 0.01, 0.01)	
m(ann, dorothy).	<i>b</i>	<i>a</i>	(0.01, 0.01, 0.97, 0.01)	
f(brian, dorothy).	...	...	...	
pc(ann).	0	0	(0.01, 0.01, 0.01, 0.97)	
pc(brian).				
mc(ann).				
mc(brian).				
	m(Y, X)	mc(Y)	pc(Y)	P(mc(X))
mc(X) m(Y, X), mc(Y), pc(Y).	<i>true</i>	<i>a</i>	<i>a</i>	(0.98, 0.01, 0.01)
pc(X) f(Y, X), mc(Y), pc(Y).	<i>true</i>	<i>b</i>	<i>a</i>	(0.01, 0.98, 0.01)
bt(X) mc(X), pc(X).	...	...	...	...
	<i>false</i>	<i>a</i>	<i>a</i>	(0.33, 0.33, 0.33)
	...	...	...	...

**Figure 10.4** 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  $\text{bt}(\mathbf{X})|\text{mc}(\mathbf{X}), \text{pc}(\mathbf{X})$  and  $\text{mc}(\mathbf{X})|\text{m}(\mathbf{Y}, \mathbf{X}), \text{mc}(\mathbf{Y}), \text{pc}(\mathbf{Y})$  are represented as tables. The other distributions are correspondingly defined. The Bayesian predicates  $\text{m}/2$  and  $\text{f}/2$  have as possible states  $\{\text{true}, \text{false}\}$ .

#### Definition 10.4 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  $\text{cpd}(c)$ , and for each Bayesian predicate  $\text{p}/l$  there is exactly one combining rule  $\text{cr}(\text{p}/l)$ .

A Bayesian logic program encoding our blood type domain is shown in figure 10.4.

#### 10.3.2 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  $\text{LH}(B)$ . It encodes the *direct influence* relation over the random variables in  $\text{LH}(B)$ : *there is an edge from a node  $\mathbf{x}$  to a node  $\mathbf{y}$  if and only if there exists a clause  $c \in B$  and a substitution  $\theta$ , s.t.  $\mathbf{y} = \text{head}(c\theta)$ ,  $\mathbf{x} \in \text{body}(c\theta)$  and for all ground atoms  $\mathbf{z}$  in  $c\theta$  :  $\mathbf{z} \in \text{LH}(B)$* . Figures 10.5 and 10.6 show the dependency graph for our *blood type* program. Here,  $\text{mc}(\text{dorothy})$  *directly* influences  $\text{bt}(\text{dorothy})$ . Furthermore, defining the *influence* relation as the transitive closure of the *direct influence* relation,  $\text{mc}(\text{ann})$  influences  $\text{bt}(\text{dorothy})$ .

The Herbrand base  $\text{HB}(B)$  constitute the set of all random variables we can talk about. However, only those atoms that are in the least Herbrand model  $\text{LH}(B) \subseteq \text{HB}(B)$  will appear in the dependency graph. These are the atoms that are true in

```

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).

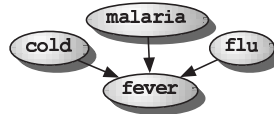
```

**Figure 10.5** The grounded version of the *blood type* Bayesian logic program of figure 10.4 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 10.6. The structure of the Bayesian network coincides with the dependency graph of the *blood type* Bayesian logic program.

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 not belonging to the least Herbrand model are irrelevant. Now, to each node  $\mathbf{x}$  in  $DG(B)$  we associate the combined conditional probability distribution which is the result of applying the combining rule  $\text{cr}(\mathbf{p}/n)$  of the corresponding Bayesian predicate  $\mathbf{p}/n$  to the set of  $\text{cpd}(c\theta)$ 's where  $\text{head}(c\theta) = \mathbf{x}$  and  $\{\mathbf{x}\} \cup \text{body}(c\theta) \subseteq \text{LH}(B)$ . 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*  $\{\mathbf{P}(\text{fever}|\text{flu}), \mathbf{P}(\text{fever}|\text{cold}), \mathbf{P}(\text{fever}|\text{malaria})\}$  is associated with *fever* (see [40], p. 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 10.5 Independence Assumption of Dependency Graph**

Each node  $\mathbf{x}$  is independent of its nondescendants given a joint state of its parents  $\text{Pa}(\mathbf{x})$  in the dependency graph.

For instance, the dependency graph of the *blood type* program as shown in Figures 10.5 and 10.6 encodes that the random variable  $\mathbf{bt}(\mathbf{dorothy})$  is independent of  $\mathbf{pc}(\mathbf{ann})$  given a joint state of  $\mathbf{pc}(\mathbf{dorothy}), \mathbf{mc}(\mathbf{dorothy})$ . Using this assumption the following proposition (taken from [21]) holds:

**Proposition 10.6 Semantics**

Let  $B$  be a Bayesian logic program. If

1.  $\mathbf{LH}(B) \neq \emptyset$ ,
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  $\mathbf{LH}(B)$ .

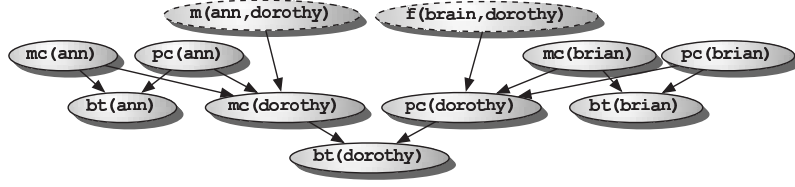
To see this, note that the least Herbrand  $\mathbf{LH}(B)$  always exists, is unique, and countable. Thus,  $DG(B)$  exists and is unique, and due to condition (3) the combined probability distribution for each node of  $DG(B)$  is computable. Furthermore, because of condition (1) a total order  $\pi$  on  $DG(B)$  exists, so that one can see  $B$  together with  $\pi$  as a stochastic process over  $\mathbf{LH}(B)$ . An induction argument over  $\pi$  together with condition (2) allows one to conclude that the family of finite-dimensional distributions of the process is projective (cf. [2]), i.e., the joint probability distribution over each finite subset  $\mathbf{S} \subseteq \mathbf{LH}(B)$  is uniquely defined and  $\sum_y \mathbf{P}(\mathbf{S}, \mathbf{x} = y) = \mathbf{P}(\mathbf{S})$ . Thus, the preconditions of *Kolmogorov's theorem* [[2], p. 307] hold, and it follows that  $B$  given  $\pi$  specifies a probability distribution  $\mathbf{P}$  over  $\mathbf{LH}(B)$ . This proves the proposition because the total order  $\pi$  used for the induction is arbitrary.

A program  $B$  satisfying the conditions (1), (2), and (3) of proposition 10.6 is called *well-defined*. A well-defined Bayesian logic program  $B$  specifies a joint distribution over the random variables in the least Herbrand model  $\mathbf{LH}(B)$ . As with Bayesian networks, the joint distribution over these random variables can be factored to

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

where the *parent* relation  $\mathbf{Pa}$  is according to the dependency graph.

The *blood type* Bayesian logic program in figure 10.4 is an example of a well-defined Bayesian logic program. Its grounded version is shown in figure 10.5. It essentially encodes the original blood type Bayesian network of Figures 10.1 and 10.3. The only differences are the two predicates  $\mathbf{m}/2$  and  $\mathbf{f}/2$  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) encodes 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).



**Figure 10.6** The structure of the Bayesian network represented by the grounded *blood type* Bayesian logic program in figure 10.5. The structure of the Bayesian network coincides with the dependency graph. Omitting the dashed nodes yields the original Bayesian network of figure 10.1.

### 10.3.3 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 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 10.7 Probabilistic Query**

A probabilistic query to a Bayesian logic program  $B$  is an expression of the form

$$?- q_1, \dots, q_n \mid e_1 = e_1, \dots, e_m = e_m$$

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

$$P(q_1, \dots, q_n \mid e_1 = e_1, \dots, e_m = e_m)$$

of the query variables  $q_1, \dots, q_n$  where  $\{q_1, \dots, q_n, e_1, \dots, e_m\} \subseteq \text{HB}(B)$ .

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.

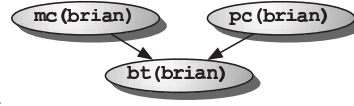
**Definition 10.8 Support Network**

The *support network*  $N$  of a random variable  $x \in \text{LH}(B)$  is defined as the induced subnetwork of

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

The support network of a finite set  $\{x_1, \dots, x_k\} \subseteq \text{LH}(B)$  is the union of the networks of each single  $x_i$ .

For instance, the support network for  $\text{bt}(\text{dorothy})$  is the Bayesian network shown in figure 10.6. The support network for  $\text{bt}(\text{brian})$  is the subnetwork with root  $\text{bt}(\text{brian})$ , i.e.,



That the support network of a finite set  $\mathbf{X} \subseteq \text{LH}(B)$  is sufficient to compute  $\mathbf{P}(\mathbf{X})$  follows from the following theorem (taken from [21]):

**Theorem 10.9 Support Network**

Let  $N$  be a possibly infinite Bayesian network, let  $\mathbf{Q}$  be nodes of  $N$ , and  $\mathbf{E} = \mathbf{e}$ ,  $\mathbf{E} \subset N$ , be some evidence. The computation of  $\mathbf{P}(\mathbf{Q} \mid \mathbf{E} = \mathbf{e})$  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  $N(\{\mathbf{q}\})$  of a single variable  $\mathbf{q}$  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  $:-\mathbf{G}_1, \mathbf{G}_2 \dots, \mathbf{G}_n$  and a clause  $\mathbf{G}:-\mathbf{L}_1, \dots, \mathbf{L}_m$  such that  $\mathbf{G}_1\theta = \mathbf{G}\theta$ , applying SLD resolution yields the new goal  $:-\mathbf{L}_1\theta, \dots, \mathbf{L}_m\theta, \mathbf{G}_2\theta \dots, \mathbf{G}_n\theta$ . 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,  $\text{bt}(\text{dorothy})$  is true, because of the following refutation:

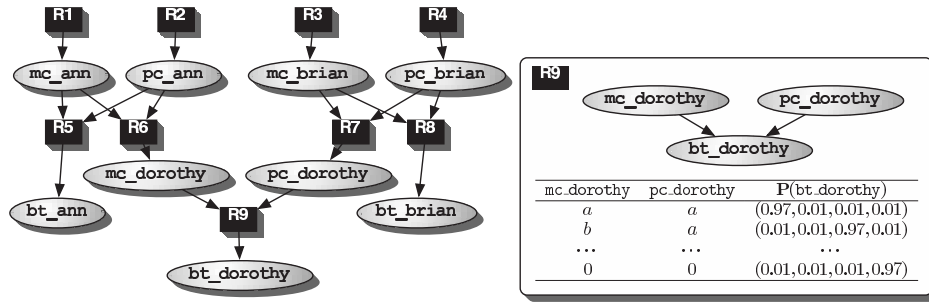
```

:-bt(dorothy)
:-mc(dorothy),pc(dorothy)
:-m(ann,dorothy),mc(ann),pc(ann),pc(dorothy)
:-mc(annn),pc(ann),pc(dorothy)
:-pc(ann),pc(dorothy)
:-pc(dorothy)
:-f(brian,dorothy),mc(brian),pc(brian)
:-mc(brian),pc(brian)
:-pc(brian)
:-

```

Resolution is employed by many theorem provers (such as Prolog). Indeed, when given the goal  $\text{bt}(\text{dorothy})$ , Prolog would compute the above successful resolution refutation and answer that the goal is true.

The set of all proofs of  $:-\text{bt}(\text{dorothy})$  captures all information needed to compute  $N(\{\text{bt}(\text{dorothy})\})$ . More exactly, the set of all ground clauses employed to prove  $\text{bt}(\text{dorothy})$  constitutes the families of the support network  $N(\{\text{bt}(\text{dorothy})\})$ . For  $:-\text{bt}(\text{dorothy})$ , they are the ground clauses shown in figure 10.5. To build the



**Figure 10.7** 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.

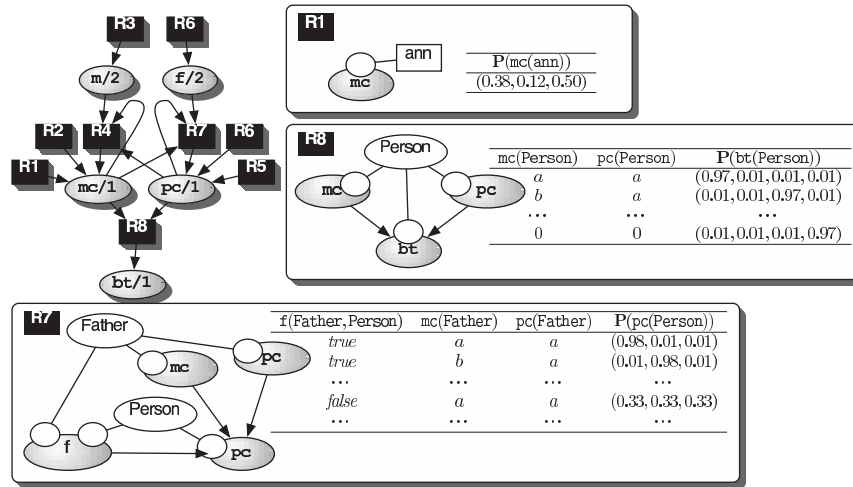
support network, we only have to gather all ground clauses used to prove the query variable and have to combine multiple copies of ground clauses with the same head using corresponding combining rules. To summarize, the support network  $N(\{q\})$  can be computed as follows:

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

Applying this to  $\text{:-bt(dorothy)}$  yields the support network as shown in figure 10.6. Furthermore, the method can easily be extended to compute the support network for  $P(Q \mid E = e)$ . We simply compute all proofs of  $\text{:-}q$ ,  $q \in Q$ , and  $\text{:-}e$ ,  $e \in E$ . 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 [43].

## 10.4 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 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 (HMMs) [39] and probabilistic grammars [29].



**Figure 10.8** The graphical representation of the *blood type* Bayesian logic program. On the righthand side, some local probability models associated with Bayesian clause nodes are shown, e.g., the Bayesian clause  $R7$   $pc(Person)|f(Father, Person), mc(Father), pc(Father)$  with associated conditional probability distribution. For the sake of simplicity, not all Bayesian clauses are shown.

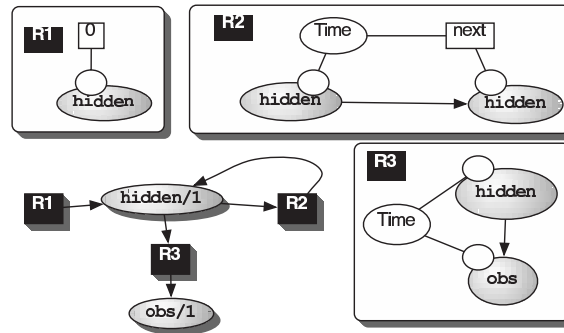
#### 10.4.1 Graphical Representation

Bayesian logic programs have so far been introduced using an adaptation of a logic programming syntax. Bayesian networks are, however, also graphical models and owe at least part of their popularity to their intuitively appealing graphical notation [20]. Inspired by Bayesian networks, we develop in this section 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 10.7. 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.

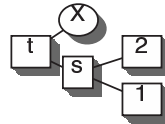
There is a box for each family  $\mathbf{Fa}(x_i)$  in the Bayesian network. The incoming edges refer to the parents  $\mathbf{Pa}(x_i)$ ; the single outgoing edge points to  $X_i$ . Each box is augmented with a Bayesian network fragment specifying the conditional probability distribution  $P(x_i|\mathbf{Pa}(x_i))$ . For instance, in figure 10.7, the fragment associated with  $R9$  specifies the conditional probability distribution of  $P(bt(dorothy)|mc(dorothy), pc(dorothy))$ . Interpreting this as a propositional



**Figure 10.9** A dynamic Bayesian logic program modeling a hidden Markov model. The functor `next/1` is used to encode the discrete time.

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  $R_i \equiv h_i | b_{i_1}, \dots, b_{i_m}$ , there are edges from  $R_i$  to  $h_i$  and from  $b_{i_j}$  to  $R_i$ . Furthermore, to each Bayesian clause node, we associate the corresponding Bayesian clause as a Bayesian network fragment. Indeed, the graphical model in figure 10.7 represents the propositional Bayesian logic program of figure 10.5.

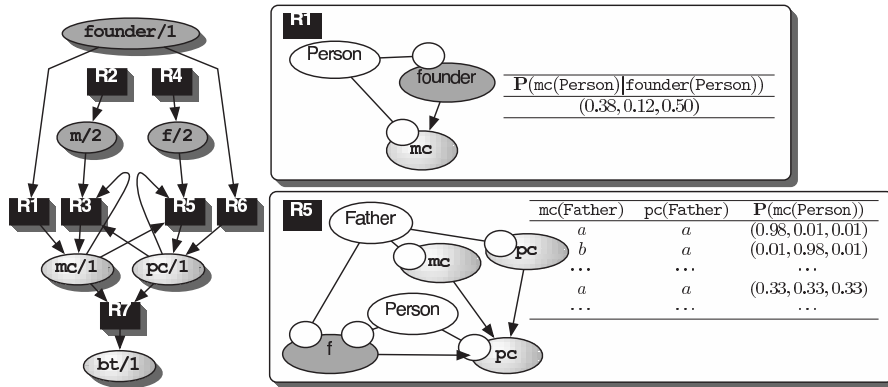
In order to represent first-order Bayesian logic programs graphically, we have to encode Bayesian atoms and their variable bindings in the associated local probability models. Indeed, logical terms can naturally be represented graphically. They form trees. For instance, the term  $t(s(1, 2), X)$  corresponds 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 gray 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 10.8. 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*.

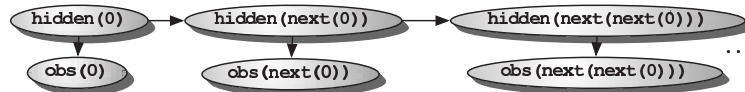




**Figure 10.10** The *blood type* Bayesian logic program distinguishing between *Bayesian* (gradient gray ovals) and *logical* atoms (solid gray ovals).

As another example, consider figure 10.9 which shows the use of functors to represent dynamic probabilistic models. More precisely, it shows an HMM [39]. HMMs are extremely popular for analyzing sequential data. Application areas include computational biology, user modeling, speech 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:



#### 10.4.2 Logical Atoms

Reconsider the *blood type* Bayesian logic program in figure 10.8. 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 represent them in the conditional probability distributions. Consider the *blood type* Bayesian logic program in figure 10.10. Here, `mother/2` and `father/2` are declared to be *logical*. Consequently, the conditional probability distribution asso-

ciated 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 `f(Father, 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 10.10, the logical background knowledge defines the `founder/1` relation as

```
founder(Person):-\+(mother(_,Person);father(_,Person)).
```

Here, `\+` denotes *negation*, the symbol `_` represents an anonymous variable which is treated as a 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 10.4. Instead of explicitly listing `pc(ann),mc(ann),pc(brian),mc(brian)` in the extensional part we have `pc(P)|founder(P)` and `mc(P)|founder(P)` in the intensional part.

The full power of Prolog is also useful to elegantly encode dynamic probabilistic models. Figure 10.11 (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:

```
next(X,Y):-integer(Y),Y > 0,X is Y - 1.
```

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

```
?- obs(next(next(...(next(0))...))).
```

Whereas HMMs define probability distributions over regular languages, *probabilistic context-free grammars* (s) [29] 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

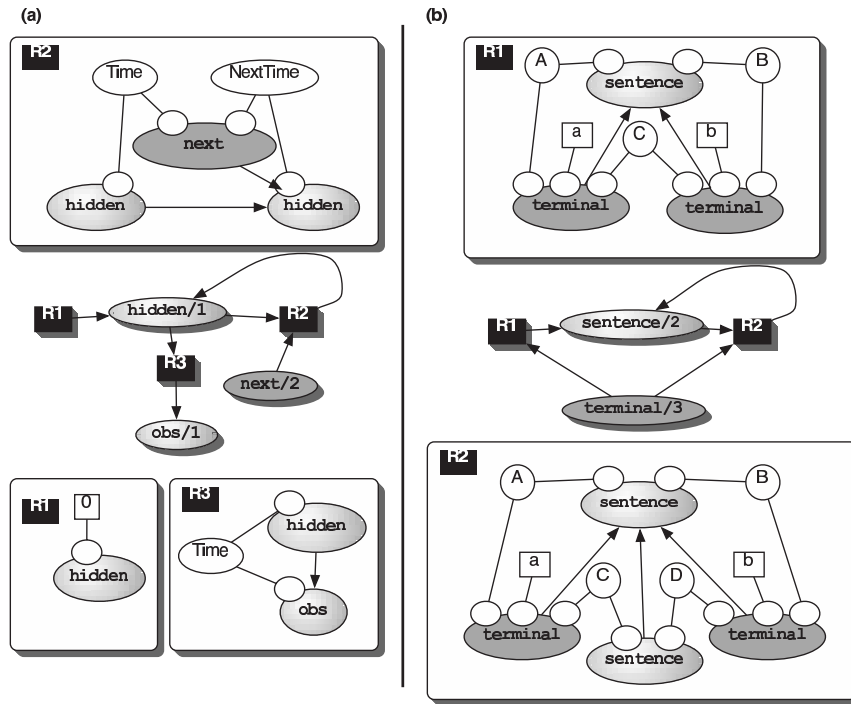
```
terminal([A|B],A,B).
```

```
0.3 : sentence(A,B):-terminal(A,a,C),terminal(C,b,B).
```

```
0.7 : sentence(A,B):-terminal(A,a,C),sentence(C,D),terminal(D,b,B).
```

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 sec-

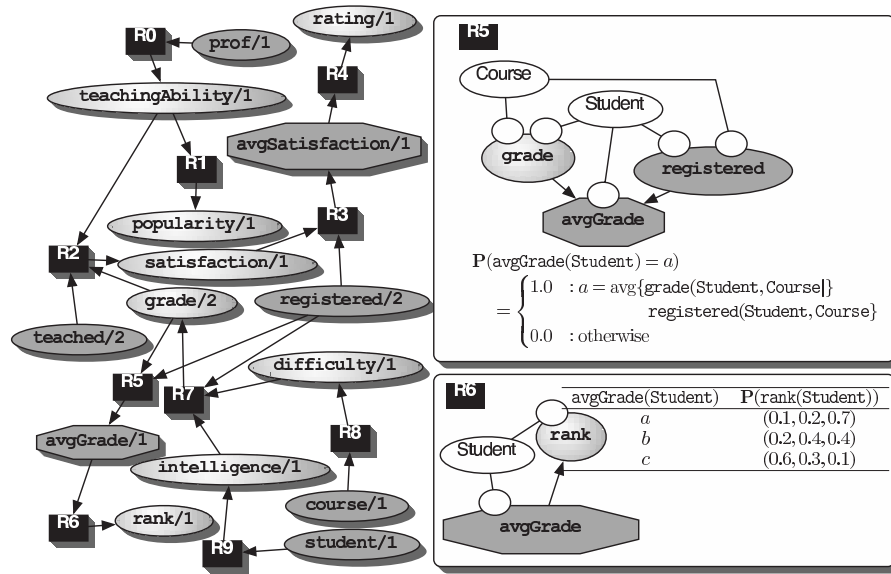


**Figure 10.11** Two dynamic Bayesian logic programs. (a) The generic structure of a hidden Markov model more elegantly represented as in figure 10.9 using `next(X,Y) :- integer(Y), Y > 0, X is Y - 1..` (b) A probabilistic context-free grammar over  $\{a^n b^n\}$ . The logical background theory defines `terminal/3` as `terminal([A|B], A, B)`.

ond 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* [30] which extend the PCFGs to definite clause logic.

Figure 10.11 (b) shows the  $\{a^n b^n\}$  PCFG represented as a 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 10.12** The Bayesian logic program for the *university* domain. Octagonal nodes denote aggregate predicates and atoms.

### 10.4.3 Aggregate Functions

An alternative to combining rules are *aggregate functions*. Consider the *university* domain due to [12]. 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 10.12. 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* [37]. As we will show in the related work section, using aggregates in Bayesian logic programs, it is easy to model probabilistic relational models.

## 10.5 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  $D = \{D_1, \dots, D_m\}$  of data cases, which can be used for learning.

### 10.5.1 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 constitutes a Herbrand interpretation for the *blood type* Bayesian logic program.

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

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 [31] 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 that is a partial assignment of states to the random variables in  $D_i$ . As an example consider the following data case:

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

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 expectation maximization (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  $score_D$ .

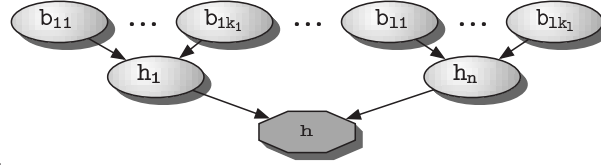
**Find** a candidate  $H^* \in \mathcal{H}$  which is acyclic on the data cases such that the data cases  $D_i \in D$  are models of  $H^*$  (in the logical sense) and  $H^*$  matches the data  $D$  best according to  $score_D$ .

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.

### 10.5.2 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 [22] for more details) that the likelihood of a Bayesian logic program coincides with the likelihood of the support network induced over  $D$ . Thus, learning 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 10.13** Decomposable combining rules can be expressed within support networks. The nodes  $h_i$  have the domain of  $h$  and  $\text{cpd}(c)$  associated. The node  $h$  becomes a deterministic node, i.e., its parameters are fixed. For example, for *noisy\_or*, logical *or* is associated as function with  $h$ . Note that the  $h_i$ 's are never observed; only  $h$  might be observed.

### 10.5.2.1 Parameter Estimation

The parameters of nonground Bayesian clauses have to be estimated. In order to adapt techniques traditionally used for parameter estimation of Bayesian networks such as the EM algorithm [7], combining rules are assumed to be *decomposable*<sup>3</sup> [15]. Decomposable combining rules can be completely expressed by adding extra nodes to the induced support network; cf. figure 10.13. 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  $\text{cpd}(c)$ . Therefore, the EM estimates the improved parameters as the following ratio:

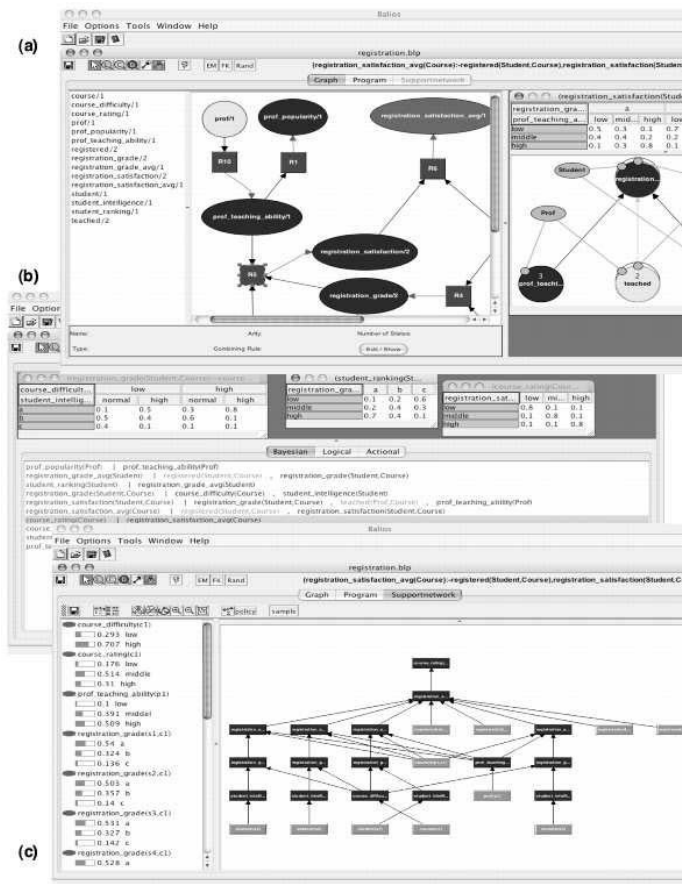
$$\frac{\sum_{l=1}^m \sum_{\theta} \mathbf{P}(\text{head}(c\theta), \text{body}(c\theta) \mid D_l)}{\sum_{l=1}^m \sum_{\theta} \mathbf{P}(\text{body}(c\theta) \mid D_l)},$$

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

### 10.5.2.2 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 too; cf. [35].

3. Most combining rules commonly employed in Bayesian networks such as *noisy\_or* are decomposable.



**Figure 10.14** 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.

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 a 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 “neighbors” 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.



---

## 10.6 Balios – The Engine for Basic Logic Programs

An engine for Bayesian logic programs featuring a graphical representation, logical atoms, and aggregate functions has been implemented in the BALIOS system [23], which is freely available for academic use at

<http://www.informatik.uni-freiburg.de/~kersting/profile/>. BALIOS is written in JAVA. It calls SICSTUS Prolog to perform logical inference and a Bayesian network inference engine to perform probabilistic inference. BALIOS features a GUI graphically representing Bayesian logic programs (figure 10.14), computing the most likely configuration, approximative inference methods (rejection, likelihood, and Gibbs sampling), and parameter estimation methods (hard EM, EM, and conjugate gradient).

---

## 10.7 Related Work

In the last ten years, there has been a lot of work done at the intersection of probability theory, logic programming, and machine learning [38, 14, 41, 30, 34, 17, 26, 1, 24, 9]; see [5] for an overview. Instead of giving a probabilistic characterization of logic programming such as [32], this research highlights the machine learning aspect and is known under the names of *statistical relational learning* (SRL) [11, 8], probabilistic logic learning (PLL) [5], or probabilistic inductive logic programming (PILP) [6]. Bayesian logic programs belong to the SRL line of research which extends Bayesian networks. They are motivated and inspired by the formalisms discussed in [38, 14, 34, 17, 10, 25]. We will now investigate these relationships in more detail.

**Probabilistic logic programs** [33, 34] 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,

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

states that the a posteriori probability of a burglary in **Person**'s house given that **Person** has an average neighborhood 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) Exclusivity constraints such as

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

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 a 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** [38], which is “a pragmatically-motivated simple logic formulation that includes definite clauses and probabilities over hypotheses” [38]. 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. [10, 25] 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  $\mathbf{a}(E)$  and each  $n$ -ary relation  $r$  is an  $n$ -ary logical Bayesian predicate  $\mathbf{r}/\mathbf{n}$ . 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  $\mathbf{a}(X)$  can depend on another attribute  $\mathbf{b}(X)$ , e.g., the professor’s popularity depends on the professor’s teaching ability in the *university* domain. This is equivalent to the Bayesian clause  $\mathbf{a}(X) \mid \mathbf{b}(X)$ . Second, an attribute  $\mathbf{a}(X)$  possibly depends on an attribute  $\mathbf{b}(Y)$  of an entity  $Y$  related to  $X$ , e.g., a student’s grade in a course depends on the difficulty of the course. The relation between  $X$  and  $Y$  is described by a slot or logical relation  $\mathbf{s}(X, Y)$ . Given these logical relations, the original dependency is represented by  $\mathbf{a}(X) \mid \mathbf{s}(X, Y), \mathbf{b}(Y)$ . 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 *existential uncertainty*, *referential uncertainty*, and *domain uncertainty* exist. 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.

Jaeger [17] 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 Jaeger 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  $\{true, false\}$ .<sup>4</sup> The key difference between relational Bayesian networks and Bayesian logic programs is that the quantitative information is specified by so-called probability formulae. These formulae 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.<sup>5</sup> Let  $F(cancer)(x)$  be  $noisy\_or\{comb_{\Gamma}\{exposed(x, y, z) \mid z; true\} \mid y; true\}$ . This formula states 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. For example, the clause  $cancer(X) \mid exposed(X, Y, Z)$  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 [41] 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 [30, 4], 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 [9] 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 formulae. The weights specify a bias for ground instances to be true in a logical model.

Finally, Bayesian logic programs are related — to some extent — to the **BUGS** language [13] which aims at carrying out Bayesian inference using Gibbs sampling.

---

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.

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.

---

## 10.8 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 HMMs 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/profile/>.

and the authors invite the reader to employ it.

---

## Acknowledgments

The authors 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 **P**robabilistic **I**nductive **L**ogic Programming).

---

## References

- [1] C. R. Anderson, P. Domingos, and D. S. Weld. Relational Markov models and their application to adaptive web navigation. In *International Conference on Knowledge Discovery and Data Mining*, 2002.
- [2] Heinz Bauer. *Wahrscheinlichkeitstheorie, 4th edition*. Walter de Gruyter, Berlin, 1991.
- [3] J. Cheng, C. Hatzis, M.-A. Krogel, S. Morishita, D. Page, and J. Sese. KDD Cup 2001 report. *SIGKDD Explorations*, 3(2):47 – 64, 2002.
- [4] J. Cussens. Loglinear models for first-order probabilistic reasoning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1999.
- [5] L. De Raedt and K. Kersting. Probabilistic logic learning. *ACM-SIGKDD Explorations: Special Issue on Multi-Relational Data Mining*, 5(1):31–48, 2003.
- [6] L. De Raedt and K. Kersting. Probabilistic inductive logic programming. In *Proceedings of the International Conference on Algorithmic Learning Theory*, pages 19–36, 2004.
- [7] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, B* 39:1–39, 1977.
- [8] 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.
- [9] 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.
- [10] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1999.
- [11] L. Getoor and D. Jensen, editors. *Working Notes of the IJCAI-2003 Workshop on Learning Statistical Models from Relational Data (SRL-03)*, 2003.
- [12] L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 307–335. Kluwer, 2001.
- [13] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex bayesian modelling. *The Statistician*, 43, 1994.
- [14] P. Haddawy. Generating Bayesian networks from probabilistic logic knowledge bases. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1994.

- [15] D. Heckerman and J. Breese. Causal independence for probability assessment and inference using bayesian networks. Technical Report MSR-TR-94-08, Microsoft Research, Seattle, WA, 1994.
- [16] D. Heckerman, A. Mamdani, and M. P. Wellman. Real-world applications of Bayesian networks. *Communications of the ACM*, 38(3):24–26, March 1995.
- [17] M. Jaeger. Relational Bayesian networks. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1997.
- [18] F. V. Jensen. *Bayesian Networks and Decision Graphs*. Springer-Verlag, 2001.
- [19] F. V. Jensen. *An Introduction to Bayesian Networks*. UCL Press Limited, 1996. Reprinted 1998.
- [20] M. I. Jordan, editor. *Learning in Graphical Models*. MIT Press, Cambridge, MA, 1998.
- [21] K. Kersting and L. De Raedt. Bayesian logic programs. Technical Report 151, University of Freiburg, Institute for Computer Science, Freiburg, Germany, April 2001.
- [22] K. Kersting and L. De Raedt. Adaptive Bayesian Logic Programs. In *Proceedings of the International Conference on Inductive Logic Programming*, 2001.
- [23] K. Kersting and U. Dick. Balios - The engine for Bayesian logic programs. In *Proceedings of the European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2004.
- [24] K. Kersting, T. Raiko, S. Kramer, and L. De Raedt. Towards discovering structural signatures of protein folds based on logical hidden Markov models. In *Proceedings of the Pacific Symposium on Biocomputing*, 2003.
- [25] D. Koller. Probabilistic relational models. In *Proceedings of the International Conference on Inductive Logic Programming*, 1999.
- [26] D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *Proceedings of the National Conference on Artificial Intelligence*, 1998.
- [27] P. Langley. *Elements of Machine Learning*. Morgan Kaufmann, San Fransisco, 1995.
- [28] J. W. Lloyd. *Foundations of Logic Programming, 2nd edition*. Springer-Verlag, Berlin, 1989.
- [29] C. H. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.
- [30] S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, Amsterdam, 1996. IOS Press.
- [31] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19(20):629–679, 1994.
- [32] R. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

- [33] L. Ngo and P. Haddawy. Probabilistic logic programming and Bayesian networks. In *Algorithms, Concurrency and Knowledge: Proceedings of the Asian Computing Science Conference*, 1995.
- [34] L. Ngo and P. Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171:147–177, 1997.
- [35] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, 1997.
- [36] J. Pearl. *Reasoning in Intelligent Systems: Networks of Plausible Inference, 2nd edition*. Morgan Kaufmann, San Fransisco, 1991.
- [37] A. J. Pfeffer. *Probabilistic Reasoning for Complex Systems*. PhD thesis, Stanford University, 2000.
- [38] D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.
- [39] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
- [40] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Upper Saddle River, NJ, 1995.
- [41] T. Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the International Conference on Inductive Logic Programming*, 1995.
- [42] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.
- [43] R. D. Schachter. Bayes-Ball: The rational pasttime (for determining irrelevance and requisite information in belief networks and influence diagrams). In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1998.