

# Probabilistic Logic Programming Under the Distribution Semantics

Fabrizio Riguzzi<sup>1</sup> and Terrance Swift<sup>2</sup>

<sup>1</sup> Dipartimento di Matematica e Informatica, University of Ferrara Ferrara, Italy

<sup>2</sup> Coherent Knowledge Systems, Mercer Island, Washington, USA

**Abstract.** The combination of logic programming and probability has proven useful for modeling domains with complex and uncertain relationships among elements. Many probabilistic logic programming (PLP) semantics have been proposed, among these the distribution semantics has recently gained an increased attention and is adopted by many languages such as the Independent Choice Logic, PRISM, Logic Programs with Annotated Disjunctions, ProbLog and P-log.

This paper reviews the distribution semantics, beginning in the simplest case with stratified Datalog programs, and showing how the definition is extended to programs that include function symbols and non-stratified negation. The languages that adopt the distribution semantics are also discussed and compared both to one another and to Bayesian networks. We then survey existing approaches for inference in PLP languages that follow the distribution semantics. We concentrate on the PRISM, ProbLog and PITA systems. PRISM was one of the first such algorithms and can be applied when certain restrictions on the program hold. ProbLog introduced the use of Binary Decision Diagrams that provide a computational basis for removing these restrictions and so performing inferences over more general classes of logic programs. PITA speeds up inference by using tabling and answer subsumption. It supports general probabilistic programs, but can easily be optimized for simpler settings and even possibilistic uncertain reasoning. The paper also discusses the computational complexity of the various approaches together with techniques for limiting it by resorting to approximation.

## 1 Introduction

If inference is a central aspect of mathematical logic, then *uncertain inference* (a term coined by Henry Kyburg) is central to much of computational logic and to logic programming in particular. The term uncertain inference captures non-monotonic reasoning, fuzzy and possibilistic logic, as well as combinations of logic and probability. Intermixing probabilistic with logical inference is of particular interest for artificial intelligence tasks such as modeling agent behavior, diagnosing complex systems, assessing risk, and conducting structure learning. It is also useful for exploiting learned knowledge that it is itself probabilistic, such as used in medical or bio informatics, natural language parsing, marketing,

and much else. These aspects have led to a huge amount of research into probabilistic formalisms such as Bayesian networks, Markov networks, and statistical learning techniques.

These trends have been reflected within the field of logic programming by various approaches to Probabilistic Logic Programming (PLP). These approaches include the languages and frameworks Probabilistic Logic Programs (Dantsin, 1991), Probabilistic Horn Abduction (Poole, 1993a), Independent Choice Logic (Poole, 1997), PRISM (Sato, 1995), pD (Fuhr, 2000), Logic Programs (BLPs) (Kersting & Raedt, 2001), CLP(BN) (Santos Costa et al., 2003), Logic Programs with Annotated Disjunctions (Vennekens et al., 2004), P-log (Baral et al., 2009), ProbLog (De Raedt et al., 2007) and CP-logic (Vennekens et al., 2009). While such a profusion of approaches indicates a ferment of interest in PLP, the question arises that if there are so many different languages, is any of them the right one to use? And why should PLP be used at all as opposed to Bayesian networks or other more popular approaches?

Fortunately, most of these approaches have similarities that can be brought into focus using various forms of the distribution semantics (Sato, 1995)<sup>3</sup>. Under the distribution semantics, a logic program defines a probability distribution over a set, each element of which is a normal logic program (termed a *world*). When there are a finite number of such worlds, as is the case with Datalog programs, the probability of an atom  $A$  is directly based on the proportion of the worlds whose model contains  $A$  as true. It can immediately be seen that distribution semantics are types of frequency semantics (cf. e.g., Halpern (2003)). By replacing worlds with sets of worlds, the same idea can be used to construct probabilities for atoms in programs that have function symbols and so may have an infinite number of worlds. From another perspective, various forms of the distribution semantics differ on how a model is associated with a world: a model may be a (minimal) stratified model, a stable model, or even a well-founded model. Finally, the semantics of Bayesian networks can be shown to be equivalent to a restricted class of probabilistic logic program under the distribution semantics, indicating that approaches to PLP are at least as powerful as Bayesian networks.

For these reasons, this paper uses distribution semantics as an organizing principle to present an introduction to PLP. Section 2 starts with examples of some PLP languages. Section 3 then formally presents the distribution semantics in stages. The simplest case — of Datalog programs with a single stratified model — is presented first in Section 3.1; using this basis the languages of Section 2 are shown to be expressively equivalent for stratified Datalog. Next, Section 3.3 extends the distribution semantics for programs with function symbols, by associating each *explanation* of a query (a set of probabilistic facts needed to prove the query) with a set of worlds, and constructing probability distributions on these sets. As a final extension, the assumption of a single stratified model for each world is lifted (Section 3.4). Section 4 then discusses PLP languages that

---

<sup>3</sup> In this paper the term *distribution semantics* is used in different contexts to refer both to a particular semantics and to a family of related semantics.

are closely related to Bayesian networks and shows how Bayesian networks are equivalent to special cases of PLPs.

The distribution semantics is essentially model-theoretic; Section 5 discusses how inferences can be made from probabilistic logic programs. First, the relevant complexity results are recalled in Section 5.1: given that probabilistic logic programs are as expressive as Bayesian networks query answering in probabilistic logic programs is easily seen to be NP-hard, and in fact is  $\#P$ -complete. Current exact inferencing techniques for general probabilistic logic programs, such as the use of Binary Decision Diagrams as pioneered in the ProbLog system (Kimmig et al., 2011) are discussed in Section 5.2. Section 5.3 discusses special cases of probabilistic logic programs for which inferencing is tractable and that have been exploited especially by the PRISM system (Sato et al., 2010).

### 1.1 Other Semantics for Probabilistic Logics

Before turning to the main topic of the paper, we briefly discuss some frameworks that are outside of the distribution semantics.

**Nilsson’s probabilistic logic** Nilsson’s probabilistic logic (Nilsson, 1986) takes a different approach than the distribution semantics for combining logic and probability: while the first considers sets of distributions, the latter computes a single distribution over possible worlds. In Nilsson’s logic, a probabilistic interpretation  $Pr$  defines a probability distribution over the set of interpretations  $Int$ . The probability of a logical formula  $F$  according to  $Pr$ , denoted  $Pr(F)$ , is the sum of all  $Pr(I)$  such that  $I \in Int$  and  $I \models F$ . A probabilistic knowledge base  $\mathcal{W}$  is a set of probabilistic formulas of the form  $F \geq p$ . A probabilistic interpretation  $Pr$  satisfies  $F \geq p$  iff  $Pr(F) \geq p$ .  $Pr$  satisfies  $\mathcal{W}$ , or  $Pr$  is a model of  $\mathcal{W}$ , iff  $Pr$  satisfies all  $F \geq p \in \mathcal{W}$ .  $Pr(F) \geq p$  is a tight logical consequence of  $\mathcal{W}$  iff  $p$  is the infimum of  $Pr(F)$  in the set of all models  $Pr$  of  $\mathcal{W}$ . Computing tight logical consequences from probabilistic knowledge bases can be done by solving a linear optimization problem.

Nilsson’s logic allows different consequences to be drawn from logical formulas than the distribution semantics. Consider a ProbLog program (cf. Section 2) composed of the facts  $0.4 :: c(a)$ . and  $0.5 :: c(b)$ .; and a probabilistic knowledge base composed of  $c(a) \geq 0.4$  and  $c(b) \geq 0.5$ . For the distribution semantics  $P(c(a) \vee c(b)) = 0.7$ , while with Nilsson’s logic the lowest  $p$  such that  $Pr(c(a) \vee c(b)) \geq p$  holds is 0.5. This difference is due to the fact that, while in Nilsson’s logic no assumption about the independence of the statements is made, in the distribution semantics the probabilistic axioms are considered as independent. While independencies can be encoded in Nilsson’s logic by carefully choosing the values of the parameters, reading off the independencies from the theories becomes more difficult.

The assumption of independence of probabilistic axioms, does not restrict expressiveness as one can specify any joint probability distribution over the logical

ground atoms, possibly introducing new atoms if needed. This claim is substantiated by the fact that Bayesian networks can be encoded in probabilistic logic programs under the distribution semantics, as discussed in Section 4.3.

**Markov Logic Networks** A Markov Logic Network (MLN) is a first order logical theory in which each sentence has a real-valued weight. A MLN is a template for generating Markov networks, graphical models where the edges among variables are undirected. Given sets of constants defining the domains of the logical variables, an MLN defines a Markov network that has a node for each ground atom and edges connecting the atoms appearing together in a grounding of a formula. MLNs follow the so-called Knowledge Base Model Construction approach for defining a probabilistic model (Wellman et al., 1992; Bacchus, 1993) in which the probabilistic-logic theory is a template for generating an underlying probabilistic graphical model (Bayesian or Markov networks). The probability distribution encoded by an MLN is

$$P(\mathbf{x}) = \frac{1}{Z} \exp\left(\sum_{f_i \in M} w_i n_i(\mathbf{x})\right)$$

where  $\mathbf{x}$  is a joint assignment of truth value to all atoms in the Herbrand base,  $M$  is the model,  $f_i$  is the  $i$ -th formula in  $M$ ,  $w_i$  is its weight,  $n_i(\mathbf{x})$  is the number of times formula  $f_i$  is satisfied in  $\mathbf{x}$  and  $Z$  is a normalization constant.

A probabilistic logic program  $T$  under the distribution semantics differs from a MLN because  $T$  has a semantics defined directly rather than through graphical models (though there are strong relationships to graphical models, see Section 4) and because restricting the logic of choice to be logic programming rather than full first-order logic, allowing to exploiting the plethora of techniques developed in logic programming.

**Evidential Probability** Evidential probability (cf. Kyburg & Teng (2001)) is an approach to reason about probabilistic information that may be approximate, incomplete or even contradictory. Evidential probability adds statistical statements of the form

$$\% \overline{\text{vars}}(C_1, C_2, Lower, Upper)$$

where  $C_1, C_2$  are formulas, and *Lower* and *Upper* are numbers between 0 and 1.  $C_1$  is called a *target* class and  $C_2$  a *reference* class. For instance

$$\%x(\text{in\_urn}(u_1, X), \text{is\_blue}(X), 0.3, 0.4)$$

might be used to state that the proportion of balls in urn  $u_1$  that are blue is between 0.3 and 0.4.

In order to determine the likelihood of whether an individual  $o_1$  is in a class  $C$  (when  $o_1$  can not be proved for certain to be in  $C$ ) each statistical statement  $S$  is collected for which  $o_1$  is known to be an element of the reference class of  $S$

and for which  $C$  is a subset of the target class of  $S$ . An series of several rules is used to derive a single interval from these collected statements, and to weigh the evidence provided for different statements in the case of a contradiction. One such rule is the principle of specificity: a statement  $S_1$  may override statement  $S_2$  if the reference class of  $S_1$  is more specific to  $o_1$  than that of  $S_2$ . For instance, a statistical statement about an age cohort might override a statement about a general population.

Evidential probability is thus not a probabilistic logic, but a meta-logic for defeasible reasoning about statistical statements once non-probabilistic aspects of a model have been derived. It is thus less powerful than probabilistic logics based on the distribution semantics, but is applicable to situations where such logics don't apply, due to contradiction, incompleteness, or other factors.

## 2 Languages for the Distribution Semantics

The languages following distribution semantics largely differ in how they encode choices for clauses, and how the probabilities for these choices are stated. As will be shown in Section 3.2, as long as models are associated to worlds in the same manner – all have the same expressive power. This fact shows that the differences in the languages are syntactic, and also justifies speaking of *the* distribution semantics.

**Probabilistic Horn Abduction** In Probabilistic Horn Abduction (PHA) (Poole, 1993a) and Independent Choice Logic (ICL) (Poole, 1997), alternatives are expressed by facts, called *disjoint-statements*, having the form

$$\text{disjoint}([A_1 : p_1, \dots, A_n : p_n]).$$

where each  $A_i$  is a logical atom and each  $p_i$  a number in  $[0, 1]$  such that  $\sum_{i=1}^n p_i = 1$ . Such a statement can be interpreted in terms of its ground instantiations: for each substitution  $\theta$  grounding the atoms of the statement, the  $A_i\theta$ s are random alternatives and  $A_i\theta$  is true with probability  $p_i$ . Each world is obtained by selecting one atom from each grounding of each disjoint-statement in the program.

*Example 1.* The following PHA/ICL program encodes the fact that a person sneezes if he has the flu and this is the active cause of sneezing, or if he has hay fever and hay fever is the active cause for sneezing:

$$\begin{aligned} \text{sneezing}(X) &:- \text{flu}(X), \text{flu\_sneezing}(X). \\ \text{sneezing}(X) &:- \text{hay\_fever}(X), \text{hay\_fever\_sneezing}(X). \\ \text{flu}(\text{bob}). \\ \text{hay\_fever}(\text{bob}). \\ \\ \text{disjoint}([\text{flu\_sneezing}(X) : 0.7, \text{null} : 0.3]). & \quad (C_1) \\ \text{disjoint}([\text{hay\_fever\_sneezing}(X) : 0.8, \text{null} : 0.2]). & \quad (C_2) \end{aligned}$$

Here, and for the other languages based on the distribution semantics, the atom *null* does not appear in the body of any clause and is used to represent an alternative in which no atom is selected.

**PRISM** The language PRISM (Sato & Kameya, 1997) is similar to PHA/ICL but allows random facts only for the predicate *msw/2* (multi-switch) whose first argument is a *random switch*, a term representing a discrete random variable; and whose second argument represents a value for that variable. The set of possible values for a variable is defined by a fact of the form

$$values(t, [v_1, \dots, v_n]).$$

where each  $v_i$  is also a term. The probability distribution over the values of a random variables  $t$  is defined by a directive of the form

$$:- set\_sw(t, [p_1, \dots, p_n]).$$

where  $p_i$  is the probability that variable  $t$  takes value  $v_i$ . Each world is obtained by selecting one value for each grounding of each random switch.

*Example 2.* Example 1 can be encoded in PRISM as:

```
sneezing(X) :- flu(X), msw(flu_sneezing(X), 1).
sneezing(X) :- hay_fever(X), msw(hay_fever_sneezing(X), 1).
flu(bob).
hay_fever(bob).

values(flu_sneezing(_X), [1, 0]).
values(hay_fever_sneezing(_X), [1, 0]).
:- set_sw(flu_sneezing(_X), [0.7, 0.3]).
:- set_sw(hay_fever_sneezing(_X), [0.8, 0.2]).
```

**Logic Programs with Annotated Disjunctions** In Logic Programs with Annotated Disjunctions (LPADs) (Vennekens et al., 2004), the alternatives are expressed by means of annotated disjunctive heads of clauses. An *annotated disjunctive clause* has the form

$$H_{i1} : p_{i1}; \dots; H_{in_i} : p_{in_i} :- B_{i1}, \dots, B_{in_i}$$

where  $H_{i1}, \dots, H_{in_i}$  are logical atoms,  $B_{i1}, \dots, B_{in_i}$  are logical literals and  $p_{i1}, \dots, p_{in_i}$  are real numbers in the interval  $[0, 1]$  such that  $\sum_{k=1}^{n_i} p_{ik} = 1$ . Each world is obtained by selecting one atom from the head of each grounding of each annotated disjunctive clause <sup>4</sup>.

<sup>4</sup> CP-logic (Vennekens et al., 2009) has a similar syntax to LPADs, and the semantics of both languages coincide for stratified Datalog programs.

*Example 3.* Example 1 can be expressed in LPADs as:

```
sneezing(X) : 0.7 ∨ null : 0.3 :- flu(X).
sneezing(X) : 0.8 ∨ null : 0.2 :- hay_fever(X).
flu(bob).
hay_fever(bob).
```

**ProbLog** The design of ProbLog (De Raedt et al., 2007) was motivated by the desire to make as simple a probabilistic extension of Prolog as possible. In ProbLog alternatives are expressed by *probabilistic facts* of the form

$$p_i :: A_i$$

where  $p_i \in [0, 1]$  and  $A_i$  is an atom, meaning that each ground instantiation  $A_i\theta$  of  $A_i$  is true with probability  $p_i$  and false with probability  $1 - p_i$ . Each worlds is obtained by selecting or rejecting each grounding of all probabilistic facts.

*Example 4.* Example 1 can be expressed in ProbLog as:

```
sneezing(X) :- flu(X), flu_sneezing(X).
sneezing(X) :- hay_fever(X), hay_fever_sneezing(X).
flu(bob).
hay_fever(bob).
0.7 :: flu_sneezing(X).
0.8 :: hay_fever_sneezing(X).
```

### 3 Defining the Distribution Semantics

In presenting the distribution semantics, we use the term *probabilistic construct* to refer to disjoint-statements, multi-switches, annotated disjunctive clauses, and probabilistic facts, in order to discuss their common properties.

The distribution semantics applies to unrestricted normal logic programs, however for the purposes of explanation we begin in Section 3.1 by making two simplifications.

- *Datalog Programs*: if a program has no function symbols, the Herbrand universe is finite and so is the set of groundings of each probabilistic construct.
- *Stratified Programs*: in this case, a program has either a total well founded model (Van Gelder et al., 1991) or equivalently a single stable model (Gelfond & Lifschitz, 1988) <sup>5</sup>.

With the distribution semantics thus defined, Section 3.2 discusses the relationships among the languages presented in Section 2. Afterwards, the restriction to Datalog programs is lifted in Section 3.3, while the restriction of stratification is lifted in Section 3.4. We note that throughout this section, all probabilities distributions are discrete; however continuous probability distributions have also been used with the distribution semantics (Islam et al., 2012).

<sup>5</sup> This restriction is sometimes called *soundness* in the PLP literature.

### 3.1 The Distribution Semantics for Stratified Datalog Programs

An *atomic choice* is the selection of the  $i$ -th atom for grounding  $C\theta$  of a probabilistic construct  $C$ . It is represented with the triple  $(C, \theta, i)$ . In Example 1,  $(C_1, \{X/bob\}, 1)$  is an atomic choice relative to disjoint-statement

$$C_1 = \text{disjoint}([\text{flu\_sneezing}(X) : 0.7, \text{null} : 0.3]).$$

Atomic choices for other languages are made similarly: for instance, an atomic choice for a ProbLog fact  $p :: A$  is made by interpreting the fact as  $A : p \vee \text{null} : 1 - p$ .

A *composite choice*  $\kappa$  is a consistent set of atomic choices, i.e., a set of atomic choices such that if  $(C, \theta, i) \in \kappa$  and  $(C, \theta, j) \in \kappa$  then  $i = j$  (only one alternative is selected for a ground probabilistic construct). In Example 1,  $\kappa = \{(C_1, \{X/bob\}, 1), (C_1, \{X/bob\}, 2)\}$  is not consistent. The probability of composite choice  $\kappa$  is

$$P(\kappa) = \prod_{(C, \theta, i) \in \kappa} p_i$$

where  $p_i$  is the probability of the  $i$ -th alternative for probabilistic construct  $C$ .

A *selection*  $\sigma$  is a total composite choice, i.e., it contains one atomic choice for every grounding of each probabilistic construct. A selection in Example 1 is  $\sigma_1 = \{(C_1, \{X/bob\}, 1), (C_2, \{bob\}, 1)\}$ .

A *world*  $w_\sigma$  is a logic program that is identified by a selection  $\sigma$ . The world  $w_\sigma$  is formed by including the atom corresponding to each atomic choice of  $\sigma$ . For instance, given the previous selection  $\sigma_1 = \{(C_1, \{X/bob\}, 1), (C_2, \{bob\}, 1)\}$ , the atoms  $\text{flu\_sneezing}(bob)$  and  $\text{hay\_fever\_sneezing}(bob)$  would be added to the first four clauses of Example 1 to make  $w_{\sigma_1}$ .

The probability of a world  $w_\sigma$  is

$$P(w_\sigma) = P(\sigma) = \prod_{(C, \theta, i) \in \sigma} p_i.$$

Since in this section we are assuming Datalog programs, the set of groundings of each probabilistic construct is finite, and so is set of worlds. Accordingly for a probabilistic logic program  $T$ ,  $W_T = \{w_1, \dots, w_m\}$ . Moreover,  $P(w)$  is a distribution over worlds:  $\sum_{w \in W_T} P(w) = 1$ .

Let  $Q$  be a query in the form of a ground atom. We define the conditional probability of  $Q$  given a world  $w$  as:  $P(Q|w) = 1$  if  $Q$  is true in  $w$  and 0 otherwise. Since in this section we consider only stratified negation (sound programs),  $Q$  can be only true or false in a world. The probability of  $Q$  can thus be computed by summing out the worlds from the joint distribution of the query and the worlds:

$$P(Q) = \sum_w P(Q, w) = \sum_w P(Q|w)P(w) = \sum_{w \models Q} P(w)$$

*Example 5.* The PHA/ICL program of Example 1 has four worlds  $\{w_1, w_2, w_3, w_4\}$ , each containing the certain (non-probabilistic) clauses:

$$\begin{aligned} & sneezing(X) :- flu(X), flu\_sneezing(X). \\ & sneezing(X) :- hay\_fever(X), hay\_fever\_sneezing(X). \\ & flu(bob). \\ & hay\_fever(bob). \end{aligned}$$

The facts from disjoint-statements are distributed among the worlds as:

$$\begin{aligned} w_1 &= flu\_sneezing(bob). & w_2 &= null. \\ & hay\_fever\_sneezing(bob). & & hay\_fever\_sneezing(bob). \\ P(w_1) &= 0.7 \times 0.8 & P(w_2) &= 0.3 \times 0.8 \end{aligned}$$

$$\begin{aligned} w_3 &= flu\_sneezing(bob). & w_4 &= null. \\ & null. & & null. \\ P(w_3) &= 0.7 \times 0.2 & P(w_4) &= 0.3 \times 0.2 \end{aligned}$$

The query  $sneezing(bob)$  is true in 3 worlds and its probability is

$$P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$$

*Example 6.* The LPAD of Example 3 has four worlds  $\{w_1, w_2, w_3, w_4\}$ :

$$\begin{aligned} w_1 &= sneezing(bob) :- flu(bob). & w_2 &= null :- flu(bob). \\ & sneezing(bob) :- hay\_fever(bob). & & sneezing(bob) :- hay\_fever(bob). \\ & flu(bob). & & flu(bob). \\ & hay\_fever(bob). & & hay\_fever(bob). \\ P(w_1) &= 0.7 \times 0.8 & P(w_2) &= 0.3 \times 0.8 \end{aligned}$$

$$\begin{aligned} w_3 &= sneezing(bob) :- flu(bob). & w_4 &= null :- flu(bob). \\ & null :- hay\_fever(bob). & & null :- hay\_fever(bob). \\ & flu(bob). & & flu(bob). \\ & hay\_fever(bob). & & hay\_fever(bob). \\ P(w_3) &= 0.7 \times 0.2 & P(w_4) &= 0.3 \times 0.2 \end{aligned}$$

The query  $sneezing(bob)$  is true in 3 worlds and its probability is  $P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$

The probability of  $sneezing(bob)$  is calculated in a similar manner for PRISM and ProbLog.

*Example 7.* PHA/ICL, PRISM and LPADs can have probabilistic statements with more than two alternatives. For example, the LPAD

$$\begin{aligned} C_1 &= strong\_sneezing(X) : 0.3 \vee moderate\_sneezing(X) : 0.5 :- flu(X). \\ C_2 &= strong\_sneezing(X) : 0.2 \vee moderate\_sneezing(X) : 0.6 :- hay\_fever(X). \\ C_3 &= flu(david). \\ C_4 &= hay\_fever(david). \end{aligned}$$

encodes the fact that flu and hay fever can cause strong sneezing, moderate sneezing or no sneezing. The clauses contain an extra atom *null* in the head that receives the missing probability mass and that is left implicit for brevity.

### 3.2 Equivalence of Expressive Power

To show that all these languages have the same expressive power for stratified Datalog programs, we discuss transformations to among probabilistic constructs from the various languages. The mapping between PHA/ICL and PRISM translates each PHA/ICL disjoint statement into a multi-switch declaration and vice-versa in the obvious way. The mapping from PHA/ICL and PRISM to LPADs translates each disjoint statement/multi-switch declaration into a disjunctive LPAD fact.

The translation from an LPAD into PHA/ICL (first shown in (Vennekens & Verbaeten, 2003)) rewrites each clause  $C_i$  with  $v$  variables  $\overline{X}$

$$H_1 : p_1 \vee \dots \vee H_n : p_n :- B.$$

into PHA/ICL by adding  $n$  new predicates  $\{choice_{i,1}/v, \dots, choice_{i,n}/v\}$  and a disjoint statement:

$$\begin{aligned} H_1 & :- B, choice_{i,1}(\overline{X}). \\ & \vdots \\ H_n & :- B, choice_{i,n}(\overline{X}). \\ disjoint & ([choice_{i,1}(\overline{X}) : p_1, \dots, choice_{i,n}(\overline{X}) : p_n]). \end{aligned}$$

Finally, as shown in (De Raedt et al., 2008), to convert LPADs to ProbLog, each clause  $C_i$  with  $v$  variables  $\overline{X}$

$$H_1 : p_1 \vee \dots \vee H_n : p_n :- B.$$

is translated into ProbLog by adding  $n - 1$  probabilistic facts for predicates  $\{f_{i,1}/v, \dots, f_{i,n}/v\}$ :

$$\begin{aligned} H_1 & :- B, f_{i,1}(\overline{X}). \\ H_2 & :- B, not(f_{i,1}(\overline{X})), f_{i,2}(\overline{X}). \\ & \vdots \\ H_n & :- B, not(f_{i,1}(\overline{X})), \dots, not(f_{i,n-1}(\overline{X})). \\ \pi_1 & :: f_{i,1}(\overline{X}). \\ & \vdots \\ \pi_{n-1} & :: f_{i,n-1}(\overline{X}). \end{aligned}$$

where  $\pi_1 = p_1$ ,  $\pi_2 = \frac{p_2}{1-\pi_1}$ ,  $\pi_3 = \frac{p_3}{(1-\pi_1)(1-\pi_2)}$ ,  $\dots$ . In general  $\pi_i = \frac{p_i}{\prod_{j=1}^{i-1} (1-\pi_j)}$ . Note that while the translation into ProbLog introduces negation, the introduced negation only involves new probabilistic facts, and so the transformed program will have a two-valued model whenever the original program does.

### Additional Examples

*Example 8.* The following program encodes the Mendelian rules of inheritance of the color of pea plants (Blockeel, 2004). The color of a pea plant is determined by a gene that exists in two forms (alleles), purple,  $p$ , and white,  $w$ . Each plant has two alleles for the color gene that reside on a couple of chromosomes.  $cg(X,N,A)$  indicates that plant  $X$  has allele  $A$  on chromosome  $N$ . The facts of the program express that  $c$  is the offspring of  $f$  and  $m$  and that the alleles of  $m$  are  $ww$  and of  $f$  are  $pw$ . The disjunctive rules encode the fact that an offspring inherits the allele on chromosome 1 from the mother and the allele on chromosome 2 from the father. In particular, each allele of the parent has a probability of 50% of being transmitted. The definite clauses for *color* express the fact that the color of a plant is purple if at least one of the alleles is  $p$ , i.e., that the  $p$  allele is dominant.

$$\begin{aligned} \text{color}(X,\text{white}) &:- \text{cg}(X,1,w), \text{cg}(X,2,w). \\ \text{color}(X,\text{purple}) &:- \text{cg}(X,-A,p). \end{aligned}$$
$$\begin{aligned} \text{cg}(X,1,A):0.5 \vee \text{cg}(X,1,B):0.5 &:- \text{mother}(Y,X), \text{cg}(Y,1,A), \text{cg}(Y,2,B). \\ \text{cg}(X,2,A):0.5 \vee \text{cg}(X,2,B):0.5 &:- \text{father}(Y,X), \text{cg}(Y,1,A), \text{cg}(Y,2,B). \end{aligned}$$
$$\begin{aligned} \text{mother}(m,c). \quad \text{father}(f,c). \\ \text{cg}(m,1,w). \quad \text{cg}(m,2,w). \quad \text{cg}(f,1,p). \quad \text{cg}(f,2,w). \end{aligned}$$

*Example 9.* An interesting application of PLP under the distribution semantics is the computation of the probability of a path between two nodes in a graph in which the presence of each edge is probabilistic:

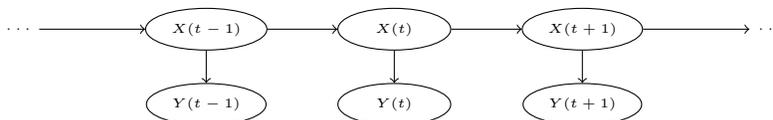
$$\begin{aligned} \text{path}(X,X). \\ \text{path}(X,Y) &:- \text{path}(X,Z), \text{edge}(Z,Y). \end{aligned}$$
$$\text{edge}(a,b):0.3. \quad \text{edge}(b,c):0.2. \quad \text{edge}(a,c):0.6.$$

This program, coded in ProbLog, was used in (De Raedt et al., 2007) for computing the probability that two biological concepts are related in the BIOMINE network (Sevon et al., 2006).

### 3.3 Distribution Semantics for Stratified Programs with Function Symbols

When a program contains functions symbols there is the possibility that its grounding may be infinite. If so, there may be an infinite number of worlds, or equivalently, the number of atomic choices in a selection that defines a world may be infinite. In either case, the probability of some individual worlds may be 0 so the semantics as defined in Section 3 is not well-defined. The distribution semantics with function symbols has been proposed for PRISM (Sato, 1995) and ICL (Poole, 1997) but is easily applicable also to the other languages discussed in Section 2. Before delving into the semantics, we first present a motivating example.

*Example 10.* A Hidden Markov Model (HMM) is a graphical model that represents a time-dependent process in which a system has a state that depends only on the state at the previous time point and an output symbol that depends on the state at the same time point. HMMs, which are used in speech recognition and many other applications, are usually represented as in Figure 1.



**Fig. 1.** Hidden Markov Model.

There are various specialized algorithm for computing the probability of an output, the state sequence that most likely gave a certain output and the values of the parameters. However the HMM of Figure 1 can also be easily encoded as a probabilistic logic program (in this case an LPAD):

```

hmm(S,O) :- hmm(q1,[],S,O).
hmm(end,S,S,[]).
hmm(Q,S0,S,[L|O]) :-
    Q \= end, next_state(Q,Q1,S0),
    letter(Q,L,S0), hmm(Q1,[Q-S0],S,O).

next_state(q1,q1,-S):1/3 ∨ next_state(q1,q2,-S):1/3 ∨ next_state(q1,end,-S):1/3.
next_state(q2,q1,-S):1/3 ∨ next_state(q2,q2,-S):1/3 ∨ next_state(q2,end,-S):1/3.

letter(q1,a,-S):1/4 ∨ letter(q1,c,-S):1/4 ∨ letter(q1,g,-S):1/4 ∨ letter(q1,t,-S):1/4.
letter(q2,a,-S):1/4 ∨ letter(q2,c,-S):1/4 ∨ letter(q2,g,-S):1/4 ∨ letter(q2,t,-S):1/4.

```

Note that the above program uses function symbols for representing the sequence of visited states. While finite, (acyclic) HMMs can be represented by Bayesian networks, if a probabilistic logic program with functions has an infinite grounding or if it has cycles, then it cannot have a direct transformation into a Bayesian network. This theme that will be discussed further in Section 4.3.

**Constructing a  $\sigma$ -Algebra based on Composite Choices** The semantics for a probabilistic logic program  $T$  with function symbols is given by defining a *probability measure*  $\mu$  over the set of worlds  $W_T$ . Informally,  $\mu$  assigns a probability to a finite set of *subsets* of  $W_T$ , rather than to every element of (the infinite set)  $W_T$ . The approach dates back to Kolmogorov (1950) who defined a probability measure  $\mu$  as a real-valued function whose domain is over elements of

a  $\sigma$ -algebra  $\Omega$  of subsets of a set  $\mathcal{W}$  called the *sample space*. Together  $\langle \mathcal{W}, \Omega, \mu \rangle$  is called a *probability space*.

The set  $\Omega$  of subsets of  $\mathcal{W}$  is a  $\sigma$ -algebra of  $\mathcal{W}$  iff

- ( $\sigma$ -1)  $\mathcal{W} \in \Omega$ ;
- ( $\sigma$ -2)  $\Omega$  is closed under complementation, i.e.,  $\omega \in \Omega \rightarrow (\mathcal{W} \setminus \omega) \in \Omega$ ; and
- ( $\sigma$ -3)  $\Omega$  is closed under countable union, i.e., if  $\omega_i \in \Omega$  for  $i = 1, 2, \dots$  then  $\bigcup_i \omega_i \in \Omega$ .

The elements of  $\Omega$  are called *measurable sets*. Importantly for defining the distribution semantics for programs with function symbols, not every subset of  $\mathcal{W}$  need be present in  $\Omega$ .

Given a sample space  $\mathcal{W}$  and a  $\sigma$ -algebra  $\Omega$  of subsets of  $\mathcal{W}$ , a *probability measure* is a function  $\mu : \Omega \rightarrow \mathbb{R}$  that satisfies the following axioms:

- ( $\mu$ -1)  $\mu(\omega) \geq 0$  for all  $\omega \in \Omega$ ,
- ( $\mu$ -2)  $\mu(\mathcal{W}) = 1$ ;
- ( $\mu$ -3)  $\mu$  is countably additive, i.e., if  $O = \{\omega_1, \omega_2, \dots\} \subseteq \Omega$  is a countable collection of pairwise disjoint sets, then  $\mu(O) = \sum_i \mu(\omega_i)$ .

For our purposes, we need only consider the finite additivity version of probability spaces (Halpern, 2003). In this stronger version, condition ( $\sigma$ -3) for  $\sigma$ -algebras replaced by ( $\sigma$ -3')  $\Omega$  is closed under finite union, i.e.,  $\omega_1 \in \Omega, \omega_2 \in \Omega \rightarrow (\omega_1 \cup \omega_2) \in \Omega$ , making  $\Omega$  an algebra rather than a  $\sigma$ -algebra. In addition, axiom ( $\mu$ -3) for probability measures is replaced by ( $\mu$ -3'):  $\mu$  is finitely additive, i.e.,  $\omega_1 \cap \omega_2 = \emptyset \rightarrow \mu(\omega_1 \cup \omega_2) = \mu(\omega_1) + \mu(\omega_2)$  for all  $\omega_1, \omega_2 \in \Omega$ . Towards defining a suitable ( $\sigma$ -)algebra given a probabilistic logic program  $T$ , define the *set of worlds*  $\omega_\kappa$  *compatible with* a composite choice  $\kappa$  as  $\omega_\kappa = \{w_{\kappa'} \in W_T \mid \kappa \subseteq \kappa'\}$ . Thus a composite choice identifies a set of worlds. For programs without function symbols  $P(\kappa) = \sum_{w \in \omega_\kappa} P(w)$ .

*Example 11.* For Example 1, consider the composite choice  $\kappa = \{(C_1, \{X/bob\}, 1)\}$ . The set of worlds compatible with this composite choice is

$$\begin{array}{ll} \omega_\kappa = \text{flu\_sneezing}(bob). & \text{flu\_sneezing}(bob). \\ \text{hay\_fever\_sneezing}(bob). & \text{null}. \\ P(w_1) = 0.7 \times 0.8 & P(w_2) = 0.7 \times 0.2 \end{array}$$

and the probability of  $\kappa$  is  $P(\kappa) = 0.7 = P(w_1) + P(w_2)$

Given a *set* of composite choices  $\mathcal{K}$ , the *set of worlds*  $\omega_\mathcal{K}$  *compatible with*  $\mathcal{K}$  is  $\omega_\mathcal{K} = \bigcup_{\kappa \in \mathcal{K}} \omega_\kappa$ . Two composite choices  $\kappa_1$  and  $\kappa_2$  are *incompatible* if their union is inconsistent. For example,  $\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$  and  $\kappa_2 = \{(C_1, \{X/bob\}, 2), (C_2, \{X/bob\}, 1)\}$  are incompatible. A set  $K$  of composite choices is *pairwise incompatible* if for all  $\kappa_1 \in K, \kappa_2 \in K, \kappa_1 \neq \kappa_2$  implies  $\kappa_1$  and  $\kappa_2$  are incompatible.

Note that in general,

$$\sum_{\kappa \in K} P(\kappa) \neq \sum_{w \in \omega_\mathcal{K}} P(w)$$

as can be seen from the following example.

*Example 12.* The set of composite choices for Example 1

$$K = \{\kappa_1, \kappa_2\} \quad (1)$$

with  $\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$  and  $\kappa_2 = \{(C_2, \{X/bob\}, 1)\}$  is such that  $P(\kappa_1) = 0.7$  and  $P(\kappa_2) = 0.8$  but  $\sum_{w \in \omega_K} P(w) = 0.94$ .

If on the other hand  $K$  is mutually incompatible, then  $\sum_{\kappa \in K} P(\kappa) = \sum_{w \in \omega_K} P(w)$ . For example, consider

$$K' = \{\kappa_1, \kappa'_2\} \quad (2)$$

with  $\kappa'_2 = \{(C_1, \{X/bob\}, 2), (C_2, \{X/bob\}, 1)\}$ .  $P(\kappa'_2) = 0.3 \cdot 0.8 = 0.24$  so the property holds with the probabilities of the worlds summing up to 0.94.

**Computing Pairwise Incompatible Sets of Composite Choices** Regardless of whether a probabilistic logic program has a finite number of worlds or not, obtaining pairwise incompatible sets of composite choices – and so of selections and the worlds they define – is an important problem. This is because the *probability of a pairwise incompatible set  $K$  of composite choices* is defined as

$$P(K) = \sum_{\kappa \in K} P(\kappa). \quad (3)$$

which is easily computed. Two sets  $K_1$  and  $K_2$  of finite composite choices are *equivalent* if they correspond to the same set of worlds:  $\omega_{K_1} = \omega_{K_2}$ .

One way to assign probabilities to a set  $K$  of composite choices is to construct an equivalent set that is pairwise incompatible; such a set can be constructed through the technique of *splitting*. More specifically, if  $F\theta$  is an instantiated formula and  $\kappa$  is a composite choice that does not contain an atomic choice  $(F, \theta, k)$  for any  $k$ , the *split* of  $\kappa$  on  $F\theta$  is the set of composite choices

$$S_{\kappa, F\theta} = \{\kappa \cup \{(F, \theta, 1)\}, \dots, \kappa \cup \{(F, \theta, n)\}\}$$

where  $n$  is the number of alternatives in  $F$ . It is easy to see that  $\kappa$  and  $S_{\kappa, F\theta}$  identify the same set of possible worlds, i.e., that  $\omega_\kappa = \omega_{S_{\kappa, F\theta}}$ . For example, the split of  $\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$  on  $C_2\{X/bob\}$  is

$$\{\{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 1)\}, \{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 2)\}\}$$

The technique of splitting composite choices on formulas is used for the following result (Poole, 2000).

**Theorem 1 (Existence of a pairwise incompatible set of composite choices (Poole, 2000)).** *Given a finite set  $K$  of finite composite choices, there exists a finite set  $K'$  of pairwise incompatible finite composite choices such that  $K$  and  $K'$  are equivalent.*

*Proof.* Given a finite set of finite composite choices  $K$ , there are two possibilities to form a new set  $K'$  of composite choices so that  $K$  and  $K'$  are equivalent:

1. **removing dominated elements:** if  $\kappa_1, \kappa_2 \in K$  and  $\kappa_1 \subset \kappa_2$ , let  $K' = K \setminus \{\kappa_2\}$ .
2. **splitting elements:** if  $\kappa_1, \kappa_2 \in K$  are compatible (and neither is a superset of the other), there is a  $(F, \theta, k) \in \kappa_1 \setminus \kappa_2$ . We replace  $\kappa_2$  by the split of  $\kappa_2$  on  $F\theta$ . Let  $K' = K \setminus \{\kappa_2\} \cup S_{\kappa_2, F\theta}$ .

In both cases  $\omega_K = \omega_{K'}$ . If we repeat this two operations until neither is applicable we obtain a splitting algorithm (see Figure 2) that terminates because  $K$  is a finite set of finite composite choices. The resulting set  $K'$  is pairwise incompatible and is equivalent to the original set. For example, the splitting algorithm applied to  $K$  (1) can return  $K'$  (2).

```

1: procedure SPLIT( $K$ )
2:   Input: set of composite choices  $K$ 
3:   Output: pairwise incompatible set of composite choices equivalent to  $K$ 
4:   loop
5:     if  $\exists \kappa_1, \kappa_2 \in K$  and  $\kappa_1 \subset \kappa_2$  then
6:        $K \leftarrow K \setminus \{\kappa_2\}$ 
7:     else
8:       if  $\exists \kappa_1, \kappa_2 \in K$  compatible then
9:         choose  $(F, \theta, k) \in \kappa_1 \setminus \kappa_2$ 
10:         $K \leftarrow K \setminus \{\kappa_2\} \cup S_{\kappa_2, F\theta}$ 
11:       else
12:         exit and return  $K$ 
13:       end if
14:     end if
15:   end loop
16: end procedure

```

**Fig. 2.** Splitting Algorithm.

**Theorem 2 (Equivalence of the probability of two equivalent pairwise incompatible finite set of finite composite choices (Poole, 1993b)).** *If  $K_1$  and  $K_2$  are both pairwise incompatible finite sets of finite composite choices such that they are equivalent then  $P(K_1) = P(K_2)$ .*

*Proof.* Consider the set  $D$  of all instantiated formulas  $F\theta$  that appear in an atomic choice in either  $K_1$  or  $K_2$ . This set is finite. Each composite choice in  $K_1$  and  $K_2$  has atomic choices for a subset of  $D$ . For both  $K_1$  and  $K_2$ , we repeatedly replace each composite choice  $\kappa$  of  $K_1$  and  $K_2$  with its split  $S_{\kappa, F_i\theta_j}$  on an  $F_i\theta_j$  from  $D$  that does not appear in  $\kappa$ . This procedure does not change the total probability as the probabilities of  $(F_i, \theta_j, 1), \dots, (F_i, \theta_j, n)$  sum to 1.

At the end of this procedure the two sets of composite choices will be identical. In fact, any difference can be extended into a possible world belonging to  $\omega_{K_1}$  but not to  $\omega_{K_2}$  or vice versa.

*Example 13.* Recall from Example 12 the set of composite choices  $K' = \{\kappa_1, \kappa'_2\}$  with  $\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$  and  $\kappa'_2 = \{(C_1, \{X/bob\}, 2), (C_2, \{X/bob\}, 1)\}$ . Consider also the composite choices  $\kappa'_{1.1} = \{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 1)\}$ ,  $\kappa'_{1.2} = \{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 2)\}$  and the set  $K'' = \{\kappa'_{1.1}, \kappa'_{1.2}, \kappa'_2\}$ . Note that  $K'$  and  $K''$  are equivalent and are both pairwise incompatible. By Theorem 2 their probabilities are equivalent:

$$P(K') = 0.7 + 0.3 \times 0.8 = 0.94$$

while

$$P(K'') = 0.7 \times 0.8 + 0.7 \times 0.2 + 0.3 \times 0.8 = 0.94.$$

For a probabilistic logic program  $T$ , we can thus define a unique probability measure  $\mu : \Omega_T \rightarrow [0, 1]$  where  $\Omega_T$  is defined as the set of sets of worlds identified by finite sets of finite composite choices:

$$\Omega_T = \{\omega_K \mid K \text{ is a finite set of finite composite choices}\}.$$

It is easy to see that  $\Omega_T$  is a ( $\sigma$ -)algebra over  $W_T$ . The corresponding measure  $\mu$  is defined by  $\mu(\omega_K) = P(K')$  where  $K'$  is a pairwise incompatible set of composite choices equivalent to  $K$ .  $\langle W_T, \Omega_T, \mu \rangle$  is a probability space according to Kolmogorov's definition.

Given a query  $Q$ , a composite choice  $\kappa$  is an *explanation* for  $Q$  if

$$\forall w \in \omega_\kappa, \quad w \models Q$$

A set  $K$  of composite choices is *covering* wrt  $Q$  if every world in which  $Q$  is true belongs to  $\omega_K$

**Definition 1.** For a probabilistic logic program  $T$ , the probability of a ground atom  $Q$  is given by

$$P(Q) = \mu(\{w \mid w \in W_T, w \models Q\})$$

If  $Q$  has a finite set  $K$  of finite explanations such that  $K$  is covering then  $\{w \mid w \in W_T \wedge w \models Q\} = \omega_K \in \Omega_T$  and we say that  $P(Q)$  is *well-defined* for the Distribution Semantics. A program  $T$  is well-defined if the probability of all ground atoms in the instantiation of  $T$  is well-defined.

*Example 14.* Consider the PHA/ICL program of Example 1. The two composite choices:

$$\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$$

and

$$\kappa_2 = \{(C_1, \{X/bob\}, 2), (C_2, \{X/bob\}, 1)\}$$

are such that  $K = \{\kappa_1, \kappa_2\}$  is pairwise incompatible finite set of finite explanations that are covering for the query  $Q = \text{sneezing}(bob)$ . Definition 1 therefore applies, and  $P(Q) = P(\kappa_1) + P(\kappa_2) = 0.7 + 0.3 \cdot 0.8 = 0.94$

**Programs for which the Distribution Semantics is Well-defined** Given the definitions of the previous section, the important questions arise of when a program or query is well-defined, and whether there are decidable algorithms to determine this.

To provide some basic intuition, it can be seen that the well-definedness conditions are related to a program having a finite model. To see this, consider probabilistic facts in ProbLog, syntactically the simplest type of probabilistic construct. For each probabilistic fact in a ProbLog program  $T$

$$p_i :: A_i$$

We simply assert  $A_i$ , creating the *saturation*  $T_S$  of  $T$ .

**Proposition 1.** *Let  $T$  be a definite ProbLog program. Then  $T$  is well-defined if the saturation of  $T_S$  has a finite model.*

*Proof.* If  $T$  has a finite model, then consider the world  $w_s$  constructed by composite choice that included no null choice for any probabilistic fact.  $T$  must have a finite model since  $T_S$  does. By monotonicity, all other worlds must have fewer true facts than  $T_S$ , so there may only be a finite number of such worlds. Since there are a finite set of finite models, for any query  $Q$  to  $T$ ,  $Q$  has a finite set of finite explanations that is covering.

More generally, it is easy to see that a (stratified) probabilistic logic program is well-defined iff it has a finite number of worlds, each of which has a finite model. However, even this condition is stronger than necessary, as a world may have an infinite model as long as each world contains only a finite number of composite choices. This following program fragment satisfies this last case along with providing a counter-example to the converse of Proposition 1:

$$\begin{aligned} q(s(X)) &:- q(X), p_i. \\ q(0). \end{aligned}$$

where  $p_i$  is an atom defined by a probabilistic fact.

In PRISM well-definedness of a program is explicitly required. (Sato & Kameya, 2001). In PHA/ICL the program (excluding disjoint statements) is required to be acyclic (Apt & Bezem, 1991). The condition of modular acyclicity is proposed in (Riguzzi, 2009) to enlarge the set of programs. This condition was weakened in (Riguzzi & Swift, 2013) to the set of programs that are bounded term-size a property whose definition is based on dynamic stratification. While the property of being bounded term-size is semi-decidable, such programs include a number of recent static classes for programs with finite models (cf.e Alviano et al. (2010); Baselice & Bonatti (2010); Calimeri et al. (2011); Greco et al. (2013) for some recent work on decidability of stable models).

An alternate approach to well-definedness is presented in (Sato & Meyer, 2012; Gorlin et al., 2012), which shows that if a covering set of explanations for a query  $Q$  is not finite or contains infinite explanations, the probability  $P(Q)$  may be defined in the limit.

### 3.4 The Distribution Semantics for Non-Stratified Programs

**The Stable Model Semantics** P-log (Baral et al., 2009) is a formalism for introducing probability in Answer Set Programming (ASP). P-log has a rich syntax that allows expression of a variety of stochastic and non-monotonic information. The semantics of a P-log program  $T$  is given in terms of its translation of into an Answer Set program  $\pi(T)$  whose stable models are the possible worlds of  $T$ . A probability is then assigned to each stable model; the unnormalized probability of a query  $Q$  is given, as for other distribution semantics languages, by the sum of the probabilities of the possible worlds where  $Q$  is true. P-log differs from the languages mentioned before because the possible worlds are generated not only because of stochastic choices but also because of disjunctions and non-stratified negations appearing in the logical part of a P-log program. As a consequence, the distribution obtained by multiplying all the probability factors of choices that are true in a stable model is not normalized. In order to get a probability distribution over possible worlds, the unnormalized probability of each stable model must be divided by the sum of the unnormalized probabilities of each possible world.

Moreover, special attention is devoted to the representation of the effects of actions according to the most recent accounts of causation (Pearl, 2000). In most of the literature on P-log, function symbols are not handled by the semantics, although Gelfond & Rushton (2010) provides recent work towards this end. Furthermore, recent work that extends stable models to allow function symbols: Alviano et al. (2010); Baselice & Bonatti (2010); Calimeri et al. (2011); Greco et al. (2013) may also lead to well-definedness conditions for P-log programs containing function symbols.

*Example 15.* An example P-Log program for computing the probability of a path between two nodes in a graph is given below:

```

bool={t,f}.                node={a,b,c,...}.
edge: node,node → bool.
#domain node(X),node(Y),node(Z).

path(X,Y):- edge(X,Y,t).
path(X,Y):- edge(X,Z,t), path(Z,Y).

[r(a,b)] random(edge(a,b)).
[r(a,b)] pr(edge(a,b,t))=4/10.
.....

```

The two sorts are *bool* and *node*, and *edge* is defined as a function from a tuple of nodes to *bool*. The domains of variables  $X$ ,  $Y$  and  $Z$  are the set of nodes;  $edge(a,b)$  has a random value assigned by experiment  $r(a,b)$ , and  $edge(a,b)$  assumes value  $t$  with probability  $4/10$ . As can be seen from the example, the syntax of P-log is highly structured, and reflects in part pragmas used by ASP grounders.

**The Well-Founded Semantics** The distribution semantics can be extended in a straightforward manner to the well-founded semantics<sup>6</sup>. In the following,  $w_\sigma \models L$  means that the ground *literal*  $L$  is true in the well-founded model of the program  $w_\sigma$ .

For a literal  $L_j$ , let  $t(L_j)$  stand as shorthand for  $L_j = true$ . We extend the probability distribution on programs to ground literals by assuming  $P(t(L_j)|w) = 1$  if  $L_j$  is true in  $w$  and 0 otherwise (regardless of whether  $L_j$  is false or undefined in  $w$ ). Thus the probability of  $L_j$  being true is

$$P(t(L_j)) = \sum_{w \in \mathcal{W}_T} P(t(L_j), w) = \sum_{w \in \mathcal{W}_T} P(t(L_j)|w)P(w) = \sum_{w \in \mathcal{W}_T, w \models L_j} P(w).$$

*Example 16.* The barber paradox, introduced by Bertrand Russell (Russell, 1967), is expressed as:

*The village barber shaves everyone in the village who does not shave himself.*

The paradox was modeled as a logic program under WFS in Dung (1991). Making things probabilistic, the paradox can be modeled as the LPAD:

$shaves(barber, Person):- villager(Person), not shaves(Person, Person).$   
 $\mathcal{C}_1 \quad shaves(barber, barber):0.25.$   
 $\mathcal{C}_2 \quad shaves(doctor, doctor):0.25.$

$villager(barber). \quad villager(mayor). \quad villager(doctor).$

where the facts that the barber and the doctor shave themselves are probabilistic.

There are four different worlds associated with this LPAD, each of which contain the villager facts along with different composite choices of the probabilistic facts..

- $w_1$  is defined by  $\{\mathcal{C}_1, \mathcal{C}_2\}$ ;  $P(w_1) = \frac{1}{16}$
- $w_2$  is defined by  $\{\mathcal{C}_1\}$ ;  $P(w_2) = \frac{3}{16}$
- $w_3$  is defined by  $\{\mathcal{C}_2\}$ ;  $P(w_3) = \frac{3}{16}$
- $w_4$  is defined by  $\emptyset$ ;  $P(w_4) = \frac{9}{16}$

Given the probabilities of each world, the probability of each literal can be computed:

- $P(shaves(doctor, doctor)) = P(w_1) + P(w_3) = \frac{1}{4}$ ;
- $P(not shaves(doctor, doctor)) = P(w_2) + P(w_4) = \frac{3}{4}$ ;
- $P(shaves(barber, doctor)) = P(w_2) + P(w_4) = \frac{3}{4}$ ;
- $P(not shaves(barber, doctor)) = P(w_1) + P(w_3) = \frac{1}{4}$ ;
- $P(shaves(barber, barber)) = P(w_1) + P(w_2) = \frac{1}{4}$ ;

<sup>6</sup> As an alternative approach, Sato et al. (2005) provides a semantics for negation in probabilistic programs based on the three-valued Fitting semantics for logic programs.

- $P(\text{not shaves}(\text{barber}, \text{barber})) = 0$

Note that  $P(A) = 1 - P(\text{not } A)$  except for the case where  $A$  is  $\text{shaves}(\text{barber}, \text{barber})$ .

From the perspective of modeling, the use of the well-founded semantics provides an approximation of the probability of an atom and of its negation, and thus may prove useful for domains in which a cautious under-approximation of probabilities is necessary. In addition, as discussed in Section 5.4, using the third truth value of the well-founded semantics offers a promising approach to semantically sound approximation of probabilistic inference.

## 4 Probabilistic Logic Programs and Bayesian Networks

In this section, we first present two examples of probabilistic logic programs whose semantics is explicitly related to Bayesian Networks: Knowledge Base Model Construction, and Bayesian Logic Programs. Making use of the formalism of Bayesian Logic Programs, we then discuss the relationship of Bayesian Networks to the Distribution Semantics

### 4.1 Knowledge Base Model Construction

In Knowledge Base Model Construction (KBMC) (Wellman et al., 1992; Bacchus, 1993), PLP is a template for building a complex Bayesian network (for background on Bayesian networks (cf. Pearl (1988) or similar texts). The semantics of the probabilistic logic program is then given by the semantics of the generated network. For example, in a CLP(BN) program (Santos Costa et al., 2003) logical variables can be random. Their domain, parents and conditional probability tables (CPTs) are defined by the program. Probabilistic dependencies are expressed by means of CLP constraints:

$$\begin{aligned} & \{ \text{Var} = \text{Function with } p(\text{Values}, \text{Dist}) \} \\ & \{ \text{Var} = \text{Function with } p(\text{Values}, \text{Dist}, \text{Parents}) \} \end{aligned}$$

The first form indicates that the logical variable  $\text{Var}$  is random with domain  $\text{Values}$  and CPT  $\text{Dist}$  but without parents; the second form defines a random variable with parents. In both forms,  $\text{Function}$  is a term over logical variables that is used to parametrize the random variable: a different random variable is defined for each instantiation of the logical variables in the term. For example, the following snippet from a school domain:

$$\begin{aligned} & \text{course\_difficulty}(\text{CKey}, \text{Dif}) :- \\ & \{ \text{Dif} = \text{difficulty}(\text{CKey}) \text{ with } p([h, m, l], \\ & \quad [0.25, 0.50, 0.25]) \}. \end{aligned}$$

defines the random variable  $\text{Dif}$  with values  $h$ ,  $m$  and  $l$  representing the difficulty of the course identified by  $\text{CKey}$ . There is a different random variable for every instantiation of  $\text{CKey}$  — i.e., for each course. In a similar manner, the intelligence  $\text{Int}$  of a student identified by  $\text{IKey}$  is given by

```

student_intelligence(SKey, Int) :-
  { Int = intelligence(SKey) with p([h, m, l],
    [0.5,0.4,0.1]) }.

```

Using the above predicates, the following snippet predicts the grade received by a student when taking the exam of a course

```

registration_grade(Key, Grade) :-
  registration(Key, CKey, SKey),
  course_difficulty(CKey, Dif),
  student_intelligence(SKey, Int),
  { Grade = grade(Key) with p(['A', 'B', 'C', 'D'],
    %h h h m h l m h m m m l l h l m l l
    [0.20,0.70,0.85,0.10,0.20,0.50,0.01,0.05,0.10,
    0.60,0.25,0.12,0.30,0.60,0.35,0.04,0.15,0.40,
    0.15,0.04,0.02,0.40,0.15,0.12,0.50,0.60,0.40,
    0.05,0.01,0.01,0.20,0.05,0.03,0.45,0.20,0.10 ],
    [Int,Dif]) }.

```

Here *Grade* indicates a random variable parametrized by the identifier *Key* of a registration of a student to a course. The code states that there is a different random variable *Grade* for each student's registration in a course, and each such random variable has possible values 'A', 'B', 'C' and 'D'. The actual value of the random variable depends on the intelligence of the student and on the difficulty of the course, that are thus its parents. Together with facts for *registration/3* such as

```

registration(r0,c16,s0).
registration(r1,c10,s0).
registration(r2,c57,s0).
registration(r3,c22,s1).

```

....

the code defines a Bayesian network with a *Grade* random variable for each registration. and with the probability of a given random variable defined by the network.

CLP(BN) is implemented as a library of YAP Prolog (Santos Costa et al., 2012). The library performs query answering by constructing the (sub-)network that is relevant to the query and then applying a Bayesian network inference algorithm. The unconditional probability of a random variable can be computed by simply asking a query the YAP command line. The answer will be a probability distribution over the values of the logical variables of the query that represent random variables, as in

```
?- registration_grade(r0,G).
```

```

p(G=a)=0.4115,
p(G=b)=0.356,

```

$$p(G=c)=0.16575,$$

$$p(G=d)=0.06675 \text{ ?}$$

Conditional queries can be posed by including in the query atoms representing the evidence. For example, the probability distribution of the grades of registration  $r0$  given that the intelligence of the student is high ( $h$ ) is given by

$$\text{?- registration\_grade}(r0,G), \text{ student\_intelligence}(s0,h).$$

$$p(G=a)=0.6125,$$

$$p(G=b)=0.305,$$

$$p(G=c)=0.0625,$$

$$p(G=d)=0.02 \text{ ?}$$

## 4.2 Bayesian Logic Programs

Bayesian Logic Programs (BLPs) (Kersting & Raedt, 2001) also use logic programming to compactly encode a large Bayesian network. In BLPs, each ground atom represents a random variable and the clauses define the dependencies between ground atoms. A clause of the form

$$A|A_1, \dots, A_m$$

indicates that, for each of its groundings  $(A|A_1, \dots, A_m)\theta$ ,  $A\theta$  has  $A_1\theta, \dots, A_m\theta$  as parents. The domains and CPTs for the ground atom/random variables are defined in a separate portion of the model. In the case where a ground atom  $A\theta$  appears in the head of more than one clause, a *combining rule* is used to obtain the overall CPT from those given by individual clauses.

For example, in the Mendelian genetics program of Example 8, the dependency that gives the value of the color gene on chromosome 1 of a plant as a function of the color genes of its mother can be expressed as

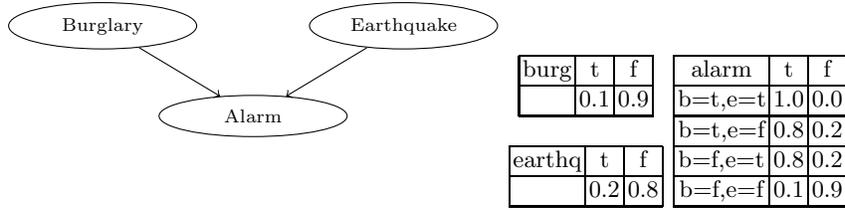
$$cg(X,1)|mother(Y,X),cg(Y,1),cg(Y,2).$$

where the domain of atoms built on predicate  $cg/2$  is  $\{p,w\}$  and the domain of  $mother(Y,X)$  is Boolean. A suitable CPT should then be defined that assigns equal probability to the alleles of the mother to be inherited by the plant.

## 4.3 Conversion of PLP under the Distribution Semantics to Bayesian Networks

In (Vennekens & Verbaeten, 2003) the relationship between LPADs and BLPs is investigated in detail. The authors show that ground BLPs can be converted to ground LPADs and that ground acyclic LPADs can be converted to ground BLPs. A logic program is *acyclic* (Apt & Bezem, 1991) if its atom dependency graph is acyclic. As a BLP directly encodes a Bayesian network, the results of (Vennekens & Verbaeten, 2003) allow us to draw a connection between LPADs and Bayesian networks.

*Example 17.* Figure 3 shows a simple Bayesian network for reasoning about the causes of a building alarm. This network can be encoded as the following LPAD:

$$\begin{aligned}
& \text{alarm}(t) \text{ :- } \text{burglary}(t), \text{earthquake}(t). \\
& \text{alarm}(t):0.8 \vee \text{alarm}(f):0.2 \text{ :- } \text{burglary}(t), \text{earthquake}(f). \\
& \text{alarm}(t):0.8 \vee \text{alarm}(f):0.2 \text{ :- } \text{burglary}(f), \text{earthquake}(t). \\
& \text{alarm}(t):0.1 \vee \text{alarm}(f):0.9 \text{ :- } \text{burglary}(f), \text{earthquake}(f). \\
& \text{burglary}(t):0.1 \vee \text{burglary}(f):0.9. \\
& \text{earthquake}(t):0.2 \vee \text{earthquake}(f):0.8.
\end{aligned}$$


**Fig. 3.** Bayesian network

In general, given a Bayesian network  $B = \{P(X_i|I_i) | i = 1, \dots, n\}$ , we obtain a ground LPAD,  $\alpha(B)$ , as follows. For each variable  $X_i$  and value  $v$  in the domain  $D_i = \{v_{i1}, \dots, v_{im}\}$  of  $X_i$  we have one atom  $X_i^v$  in the Herbrand base of  $\alpha(B)$ . For each row  $P(X_i | X_{i1} = v_1, \dots, X_{il} = v_l)$  of the Conditional Probability Table (CPT) for  $X_i$  with probabilities  $p_1, \dots, p_{im}$  for the values of  $X_I$ , we have an LPAD clause

$$X_i^{v_{i1}} : p_1 \vee \dots \vee X_i^{v_{im}} : p_m \text{ :- } X_{i1}^{v_1}, \dots, X_{il}^{v_l}.$$

in  $\alpha(B)$ .

**Theorem 3.** *Bayesian network  $B$  and  $\alpha(B)$  define the same probability distribution.*

*Proof.* There is an immediate translation from a Bayesian network  $B$  to a ground BLP  $B'$ . Theorem 4 of (Vennekens & Verbaeten, 2003) states the equivalence of the semantics of a ground BLP  $B'$  and a ground LPAD  $\gamma(B')$  obtained from  $B'$  with a translation  $\gamma$ . This translation is in close correspondence with  $\alpha$  so the theorem is proved.

Extending Theorem 3, acyclic Datalog LPADs under the distribution semantics can be translated to Bayesian networks. To do so, first the grounding of the program must be generated. Consider an LPAD  $T$  and let  $\text{ground}(T)$  be its grounding. For each atom  $A$  in the Herbrand base  $\mathcal{H}_T$  of  $T$ , the network contains a binary variable  $A_{bv}$ . For each clause  $C_i$  in  $\text{ground}(T)$

$$H_1 : p_1 \vee \dots \vee H_n : p_n \text{ :- } B_1, \dots, B_m, \neg C_1, \dots, \neg C_l$$

the network contains a variable  $CH_i$  with  $H_1, \dots, H_n$  and *null* as values.  $CH_i$  has  $B_1, \dots, B_m, C_1, \dots, C_l$  as parents. The CPT of  $CH_i$  is

	...	$B_1 = 1, \dots, B_m = 1, C_1 = 0, \dots, C_l = 0$	...
$CH_i = H_1$	0.0	$p_1$	0.0
...			
$CH_i = H_n$	0.0	$p_n$	0.0
$CH_i = null$	1.0	$1 - \sum_{i=1}^n p_i$	1.0

Basically, if the body assumes a false value, then  $CH_i$  assumes value *null* with certainty, otherwise the probability is distributed over atoms in the head of  $C_i$  according to the annotations.

Each variable  $A_{bv}$  corresponding to atom  $A$  has as parents all the variables  $CH_i$  of clauses  $C_i$  that have  $A$  in the head. The CPT for  $A_{bv}$  is:

	at least one parent equal to $A$	remaining columns
$A = 1$	1.0	0.0
$A = 0$	0.0	1.0

This table encodes a deterministic function:  $A$  assumes value 1 with certainty if at least one parent assumes value  $A$ , otherwise it assumes value 0 with certainty. Let us call  $\lambda(T)$  the Bayesian network obtained with the above translation from an LPAD  $T$ . Then the following theorem holds.

**Theorem 4.** *Given an acyclic Datalog LPAD  $T$ , the Bayesian network  $\lambda(T)$  defines the same probability distribution over the atoms of  $\mathcal{H}_T$ .*

*Proof.* The proof uses Theorem 5 of (Vennekens & Verbaeten, 2003) that states the equivalence of the semantics of a ground LPAD  $T$  and a ground BLP  $\beta(T)$  obtained from  $T$  with a translation  $\beta$  in close correspondence with  $\lambda$ . Since there is an immediate translation from a ground BLP to a Bayesian network, the theorem is proved.

Together, Theorems 3 and 4 show the equivalence of the Distribution Semantics with that of Bayesian networks for the special case of acyclic probabilistic logic programs. As discussed in previous sections, however, the Distribution Semantics is defined for larger classes of programs, indicating its generality.

## 5 Inferencing in Probabilistic Logic Programs

So far, we have focused mainly on definitions and expressivity of the distribution semantics, and this presentation has had a somewhat model-theoretic flavor. This section focuses primarily on the main inference task for probabilistic logic programs: that of query evaluation. In its simplest form, query evaluation means determining the probability of a ground query  $Q$  when no evidence is given. This task is equivalent to finding a finite set of finite explanations for  $Q$  that is covering (if one exists).

Section 5.1 discusses the computational complexity of query evaluation which, perhaps not surprisingly, is high. Current techniques for computing the distribution semantics for stratified programs are discussed in Section 5.2. Because of the high computational complexity, these general techniques are not always scalable. Section 5.3 discusses a restriction of the distribution semantics, pioneered by the PRISM system, for which query evaluation is tractable. Another approach is to only approximate the point intervals of the distribution semantics, as discussed in Section 5.4. Section 5.4 also briefly discusses other inferencing tasks, such as computing the Viterbi probability for a query.

### 5.1 The Complexity of Query Evaluation

To understand the complexity of query evaluation for PLPs, let  $Q$  be a ground query to a probabilistic logic program  $T$  for which the distribution semantics is well-defined. A simple approach might be to somehow save the probabilistic choices made for each proof of  $Q$ . For instance, each time a probabilistic atom was encountered as a subgoal, the corresponding atomic choice  $(C, \theta, i)$  would be added to a data structure. As a result each proof of  $Q$  would be associated with an explanation  $E_j$ , and when all  $n$  proofs of  $Q$  had been exhausted the set of explanations  $\mathcal{E} = \cup_{j \leq n} E_j$  would cover  $Q$ . While this approach was sketched for top-down evaluations, a similar approach could be constructed in a bottom-up manner.

If all  $E_j$  were known to be mutually exclusive, the probability of  $Q$  ( $= P(\mathcal{E})$ ) could be computed simply by computing the probability of each explanation and summing them up; but this is not generally the case. Usually, explanations are not pairwise independent, requiring a technique such as the principle of inclusion-exclusion to be used (cf. e.g., Rauzy et al. (2003)):

$$P(\mathcal{E}) = \sum_{1 \leq i \leq n} P(E_i) - \sum_{1 \leq i < j \leq n} P(E_i, E_j) + \sum_{1 \leq i < j < k \leq n} P(E_i, E_j, E_k) - \dots + (-1)^{n+1} P(E_i, \dots, E_n) \quad (4)$$

Unfortunately, use of the inclusion-exclusion algorithm is exponential in  $n$ . Is there a better way?  $\mathcal{E}$  can also be viewed as a logical formula,  $formula(\mathcal{E})$ , in disjunctive normal form. The difficulty of determining the number of solutions to a propositional formula such as  $formula(\mathcal{E})$  is the canonical  $\#P$ -complete problem, and computing the probability of  $\mathcal{E}$  is at least as difficult as computing the number of solutions of  $formula(\mathcal{E})$ . It can easily be shown that computing the probability of  $\mathcal{E}$  also is in  $\#P$  so that it is a  $\#P$ -complete problem. For practical purposes, computing the probability of  $\mathcal{E}$  can be thought of as equivalent to a  $FPspace$  complete problem (where an  $FPspace$  problem outputs a value, unlike a  $Pspace$  problem) <sup>7</sup>.

<sup>7</sup> It is easy to see that counting solutions to a  $\#P$ -complete problem can be done in polynomial space. By Toda's Theorem, every problem in  $FPspace$  is reducible in polynomial time to a problem in  $\#P$  (cf. Papadimitriou (1994)).

## 5.2 Exact Query Evaluation for Unrestricted Programs

At this point, there have been two classes of approaches to exact query evaluation for programs in which the use of the distribution semantics is unrestricted (although the programs themselves may be restricted): transformational and direct approaches. As mentioned above, we focus on probabilistic queries without evidence.

**Transformational Approaches** Given the relationship between an acyclic Datalog probabilistic logic program  $T$  and a Bayesian Network as stated in Theorem 4 of Section 4, one approach is to transform  $T$  into a Bayesian network, use Bayesian Network inference algorithms to evaluate the query, and then translate back the results. Given the large amount of work on efficiently evaluating Bayesian networks (cf. Koller & Friedman (2009)), such an approach could lead to efficient evaluations.

This approach was used in CVE inferencing (Meert et al., 2008, 2009)), which evaluated CP-logic (Vennekens et al., 2009), a formalism closely related to LPADs. Some of the factors of the Bayesian network that results from the translation contain redundant information since they have many identical columns. To reduce the size of the generated network, this situation, called *contextual independence*, can be exploited during inference using a special technique called contextual variable elimination (Poole & Zhang, 2003). CVE applies this technique to compute the probability of queries to CP-logic programs.

An alternative approach is taken by a very recent implementation of the ProbLog system (called ProbLog2 to distinguish it from previous implementations, Fierens et al. (2014)), which converts a program, queries and evidence (if any) to a weighted Boolean formula (cf. Chavira & Darwiche (2008)). Once transformed, the program can be evaluated by an external weighted model counting or max-SAT solver.

**Direct Approaches Based on Explanation** A more direct approach is to find a set of explanations that is covering for a query  $Q$  and then to make the explanations pairwise incompatible. Explanations can be made pairwise incompatible in a number of ways. The pD engine (Fuhr, 2000) uses inclusion-exclusion (Equation 4) directly. The Ailog2 system for Independent Choice Logic (Poole, 2000)), iteratively applies the Splitting Algorithm (Section 3.3, Figure 2). More commonly however, Binary Decision Diagrams (BDDs) (Bryant, 1992) are used to ensure pairwise incompatibility. This approach was first used in the ProbLog system (De Raedt et al., 2007), and later adopted by several other systems including `cpint` (Riguzzi, 2007, 2009) and PITA (Riguzzi & Swift, 2013)

The BDD data structure was designed to efficiently store Boolean functions (i.e., formulas), which makes it a natural candidate to store explanations. A BDD is a directed acyclic graph, with a root node representing the start of the function, and with terminal nodes 0 (false) and 1 (true). An interior node,  $n_i$ , sometimes called a decision node, represents a variable  $v_i$  in the Boolean

function. Each such  $n_i$  has a 0-child representing the next node whose truth value will be examined if  $v_i$  is false, and a 1-child representing the next node whose truth value will be examined if  $v_i$  is true. Accordingly, each path from root to terminal node in a BDD represents a (partial or total) truth assignment to the variables leading to the truth or falsity of the formula. What gives a BDD its power are the following operations.

- *Ordering*: all paths through the BDD traverse variables in the same order. This ensures that each variable is traversed at most once on a given path.
- *Reduction*: within a BDD isomorphic subgraphs are merged, and any node whose two children root isomorphic subgraphs (or the same subgraph) is considered redundant and removed from the BDD. These operations ensure that once enough variables have been traversed to determine the value of the Boolean function, no other variables will be traversed (or need to be stored).

Although performing these operations when building a BDD can be expensive, the resulting BDD has the property that any two distinct paths differ in the truth value of at least one variable, so that BDDs are an efficient way to store and manipulate pairwise incompatible explanations as described in Section 3.3.

To explain the details, consider an application to ProbLog, where in each probabilistic fact, either an atom or its (default) negation may be chosen. Let  $(C, \theta, i)$  be an atomic choice for selection of the ground probabilistic fact:  $(C, \theta, 1)$  means that  $C\theta$  was chosen, and  $(C, \theta, 2)$  means that  $\text{not } C\theta$  was chosen. If we consider these atomic choices as Boolean random variables, then a set of explanations is simply a DNF formula, and storing this formula in a BDD will ensure pairwise incompatibility of the explanations in the set. Recall that if  $\mathcal{K}$  is a pairwise incompatible set of explanations that is covering for a probabilistic query  $Q$ , then probability of  $Q$  is given by

$$P(Q) = \sum_{\kappa \in \mathcal{K}} \prod_{(C, \theta, i) \in \kappa} (P((C, \theta, i))).$$

Accordingly, once  $\mathcal{K}$  is stored in a BDD, a simple traversal of the BDD suffices to compute the probability of  $Q$  as shown in Figure 4.

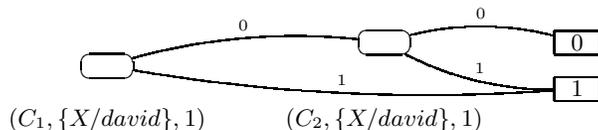
```

node_prob(BDD node n)
  if n is the 1-terminal return 1
  if n is the 0-terminal return 0
  let t_child be the 1-child of n and let f_child be the 0-child of n and
  return P((C, θ, 1)) × node_prob(t_child) + (1 - P((C, θ, 1))) × node_prob(f_child)

```

**Fig. 4.** Determining the probability of a node in a BDD used to store a covering set of explanations (De Raedt et al., 2007)

*Example 18.* Returning to the sneezing example of Section 2, a set of covering explanations for  $sneezing(david)$  is  $\mathcal{K} = \{\kappa_1, \kappa_2\}$ , where  $\kappa_1 = \{(C_1, \{X/david\}, 1)\}$  and  $\kappa_2 = \{(C_2, \{X/david\}, 1)\}$ . A BDD representing  $\mathcal{K}$  is shown in Figure 5: while  $\mathcal{K}$  is not pairwise incompatible, note that the ordering and reduction operations used in constructing the BDD result in the fact that all paths through the BDD represent pairwise incompatible explanations. Using this BDD, the probability of  $sneezing(david)$  can be calculated by the simple algorithm of Figure 4.



**Fig. 5.** A BDD Representing a Pairwise Incompatible Set of Explanations for  $sneezing(david)$

*Implementations of BDD-Based Explanation Approaches* The ProbLog system (Kimig et al., 2011), implemented using YAP Prolog (Santos Costa et al., 2012), has a two-phase approach to computing probabilities for ProbLog programs. A source-code transformation is made of a ProbLog program so that during the SLD-proof phase each atomic choice is added to a running list representing the explanation of the proof; when the proof is completed the explanation is stored in a trie of the style used for tabling in YAP and XSB. Once all proofs have been completed the trie is traversed so that a BDD can be efficiently created using an external BDD package <sup>8</sup>.

The `cpaint` system Riguzzi (2007) is also implemented using YAP Prolog and an external BDD package, but implements LPADs. To directly implement LPADs, two extensions must be made. First, the BDD interface must be modified to support ground atomic choices that allow more than 2 outcomes; second default negation is supported via SLDNG. The PITA system (Riguzzi & Swift, 2013), is based on XSB Prolog (Swift & Warren, 2012) and uses tabling extended with answer subsumption in order to combine different explanations. As each new explanation is derived for a given subgoal  $G$ , it is added to the current BDD for  $G$ . When a tabling technique termed call subsumption is also used, PITA can be shown to theoretically terminate on *any* well-defined LPAD that is stratified and for which all worlds have finite models.

Papers about CVE, BDD-based ProbLog, `cpaint`, and PITA have compared the systems on certain probabilistic programs. Not surprisingly, source code transformations outperform meta-interpretations. The current generation of BDD-based systems usually — but not always — outperform the transformation-based

<sup>8</sup> The Cudd package (<http://vlsi.colorado.edu/~fabio/CUDD/node7.html>) is used for ProbLog as well as for `cpaint` and for PITA.

CVE, while recent experiments in (Fierens et al., 2014) indicates that translation of probabilistic logic programs into weighted Boolean formulas outperforms the use of BDDs on certain programs. ProbLog and PITA, which are more closely related, show conflicting experimental results. Timings based on PITA have shown that for traversals of probabilistic networks, the vast majority of time is spent in BDD manipulation. Based on the current set of experiments and benchmarks, there is no clear evidence about whether it is more efficient to construct a BDD during the course of evaluation as with PITA or to wait until the end as with ProbLog. In short, much more implementational and experimental work is needed to determine the best way to evaluate queries for unrestricted probabilistic logic programs.

Overall, implementation of probabilistic reasoning for the ASP-based P-Log has received less attention, although (Gelfond et al., 2006) describes a prototype implementation; while Anh et al. (2008) describe an approach that grounds a P-Log program using XSB, then sending it to an ASP solver.

### 5.3 Exact Query Evaluation for Restricted Programs

As an alternative to inference for unrestricted programs, an implementation may be restricted to programs for which the probability of queries is easy to compute. In particular, an implementation may assume axioms of *exclusion* and *independence*. This approach has been followed in an early implementation of PHA (Poole, 1993a), and in a module of the PITA system (Riguzzi & Swift, 2011); however this approach has been most thoroughly developed in the PRISM system (Sato et al., 2010), implemented using B Prolog (Zhou, 2012)<sup>9</sup>.

The assumption of exclusion allows the probability of a disjunction  $A \vee B$  to be computed as the sum of the probabilities of  $A$  and  $B$ . In other words, a covering set of explanations may be obtained, and the probability computed without having to ensure that these explanations are pairwise independent. As an example, the program

$$\begin{array}{ll} q \text{ :- } a. & a:0.2. \\ q \text{ :- } a,b.. & b:0.4. \end{array}$$

violates the exclusiveness assumption as the two clauses for the ground atom  $q$  have non-exclusive bodies

The assumption of independence allows the probability of a conjunction  $A \wedge B$  to be computed as the product of the probabilities of  $A$  and  $B$ . As an example, the program

$$\begin{array}{lll} q \text{ :- } a,b. & & \\ a \text{ :- } c. & b \text{ :- } c. & c:0.2. \end{array}$$


---

<sup>9</sup> The assumptions of exclusion and independence are made in the PRISM system, but not in the PRISM language (Sato & Kameya, 1997).

does not satisfy the independence assumption because  $a$  and  $b$  both depend on  $c$ . In the distribution semantics the probability is 0.2, but if an independence assumption is made the probability is 0.04.

To get an idea about how restrictive these assumptions are in practice, consider the examples introduced so far in this paper. The sneezing examples (Examples 1–7) violate the exclusion assumption; the path example (Examples 9 and 15) violates both independence and exclusion; the barber example (Example 16) violates independence. However the examples about Mendelian inheritance (Example 8), Hidden Markov Models (Example 10) and alarm (Example 4.1) satisfy both assumptions. In terms of practical applications, programs with the independence and exclusion assumptions have been used for parameter learning (Sato & Kameya, 2001), and for numerous forms of generative modeling (Sato & Kameya, 2008).

PRISM implements queries to probabilistic logic programs with the independence and exclusion assumptions by using tabling to collect a set of explanations that has any duplicates filtered out. Probabilities are then collected directly from this set. PITA also uses tabling, but with the addition of answer subsumption to combine probabilities of different explanations as query evaluation progresses. In either case, computation is much faster than if the independence and exclusion assumptions do not hold. Additionally, projecting out superfluous (non-discriminating) arguments from subgoals using the technique of (Christiansen & Gallagher, 2009) can lead to significant speed improvement for Hidden Markov Model examples. Finally, Riguzzi (2011) presents approaches for efficient evaluation of probabilistic logic programs that do not use the full independence and exclusion assumptions.

#### 5.4 Approximation and Other Inferencing Tasks

For programs that violate the independence or exclusion assumptions, and for which exact inference may be too expensive, approximate inference may be considered. Recall from Equation 4 that, using the inclusion-exclusion principle, computing probability is exponential in the number of explanations. Accordingly ProbLog (Kimmig et al., 2011) supports an optimization that retains only the  $k$  most likely explanations, thus reducing the cost of building a BDD to make the explanations pairwise incompatible. ProbLog also offers an approach similar to iterative deepening, where lower and upper bounds on the probability are iteratively computed and inference terminates when their difference is below a given threshold. Both of these approximations are sound only for definite programs; if negation is used, sound approximation requires a three-valued semantics (Section 3.4) to distinguish the known probabilities of a query and negation from the range that is still unknown due to approximation.

Monte Carlo simulations are also used by various systems. Monte Carlo in PRISM performs Bayesian inference (updating a prior probability distribution in the light of evidence) by updating a Metropolis-Hastings algorithm for Probabilistic Context Free Grammars (Sato, 2011). ProbLog and PITA perform plain

Monte Carlo by sampling the worlds and counting the fraction where the query is true, exploiting tabling to save computation.

Lastly, while this section has focused on evaluation of ground queries when there is no additional supporting evidence, this is by no means the only inference problem that has been studied. ProbLog2 (Fierens et al., 2014) evaluates queries with and without supporting evidence. PRISM supports Maximum A Posteriori (or Most Probable Explanation) inference, which finds the most likely state of a set of query atoms given some evidence. In Hidden Markov Models, this inference reduces to finding the most likely sequence of the state variables also called the Viterbi path (also supported by PITA). Finally, recent work seeks to perform inference in a lifted way, i.e., by avoiding grounding the model as much as possible. This technique can lead to exponential savings in some cases (Van den Broeck et al., 2014; Bellodi et al., 2014).

## 6 Discussion

This paper has described the distribution semantics for logic programs, starting with stratified Datalog programs, then showing how the semantics has been extended to programs that include function symbols and non-stratified negation (Section 3). Various PLP languages have been described and their inter-translatability has been discussed (Section 2). The relationship of PLPs and Bayesian networks has also been shown (Section 4). Finally, the intractable problem of inferencing with the distribution semantics was discussed in Section 5 along with implementations that either directly address the full distribution semantics; make simplifying restrictions about the types of programs for which they provide inference; or perform heuristic approximations.

We believe that this material provides necessary background for much of the current research into PLP. However as noted, our focus on the distribution semantics leaves out many interesting and important languages and systems (a few of which were summarized in Section 1.1). In addition, we have not covered the important problem of using these languages for machine learning. Indeed, the support for machine learning has been an important motivation for PLPs since the very first proposals and nowadays a variety of systems are available for learning either the parameters or the structure of programs under the distribution semantics.

To mention a very few such systems, PRISM (Sato & Kameya, 2001), LeP-robLog (Gutmann et al., 2008), LFI-ProbLog (Gutmann et al., 2011), EMBLEM (Bellodi & Riguzzi, 2013) and ProbLog2 (Fierens et al., 2014) learn the parameters either by using an EM algorithm or by gradient descent. SEM-CP-logic (Meert et al., 2008), SLIPCASE (Bellodi & Riguzzi, 2012) and SLIPCOVER (Bellodi & Riguzzi, 2014) learn both the structure and the parameters by performing a search in the space of possible programs and using parameter learning as a subroutine.

All these systems have been successfully applied to a variety of domains, including biology, medicine, link prediction and text classification. The results

obtained show that these systems are competitive with systems at the state of the art of statistical relational learning such as Alchemy (Richardson & Domingos, 2006) and others.

## Bibliography

- Alviano, M., Faber, W., & Leone, N. (2010). Disjunctive ASP with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming*, 10(4-6), 497–512.
- Anh, H., Ramli, C., & Damásio, C. (2008). An implementation of extended P-Log using XASP. In *International Conference on Logic Programming*, (pp. 739–743).
- Apt, K. R. & Bezem, M. (1991). Acyclic programs. *New Generation Computing*, 9(3/4), 335–364.
- Bacchus, F. (1993). Using first-order probability logic for the construction of bayesian networks. In *International Conference on Uncertainty in Artificial Intelligence*, (pp. 219–226).
- Baral, C., Gelfond, M., & Rushton, N. (2009). Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming*, 9(1), 57–144.
- Baselice, S. & Bonatti, P. (2010). A decidable subclass of finitary programs. *Theory and Practice of Logic Programming*, 10(4-6), 481–496.
- Bellodi, E., Lamma, E., Riguzzi, F., Santos Costa, V., & Zese, R. (2014). Lifted probabilistic logic programming. *Theory and Practice of Logic Programming*, 14.
- Bellodi, E. & Riguzzi, F. (2012). Learning the structure of probabilistic logic programs. In *International Conference on Inductive Logic Programming*, (pp. 61–75).
- Bellodi, E. & Riguzzi, F. (2013). Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis*, 17(2), 343–363.
- Bellodi, E. & Riguzzi, F. (2014). Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*. To appear, available at <http://arxiv.org/abs/1309.2080>.
- Blockeel, H. (2004). Probabilistic logical models for Mendel’s experiments: An exercise. In *International Conference on Inductive Logic Programming*. Work in Progress Track.
- Bryant, R. (1992). Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), 293–318.
- Calimeri, F., Cozza, S., Ianni, G., & Leone, N. (2011). Finitely recursive programs: Decidability and bottom-up computation. *AI Communication*, 24(4), 311–334.
- Chavira, M. & Darwiche, A. (2008). On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7), 772–799.
- Christiansen, H. & Gallagher, J. (2009). Non-discriminating arguments and their uses. In *International Conference on Logic Programming*, (pp. 55–69).
- Dantsin, E. (1991). Probabilistic logic programs and their semantics. In *Russian Conference on Logic Programming*, volume 592 of *LNCS*, (pp. 152–164). Springer.

- De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., Kimmig, A., Landwehr, N., Mantadelis, T., Meert, W., Rocha, R., Santos Costa, V., Thon, I., & Vennekens, J. (2008). Towards digesting the alphabet-soup of statistical relational learning. In *NIPS\*2008 Workshop on Probabilistic Programming*.
- De Raedt, L., Kimmig, A., & Toivonen, H. (2007). ProbLog: A probabilistic Prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*, (pp. 2462–2467).
- Dung, P. (1991). Negation as hypothesis: An abductive foundation for logic programming. In *International Conference on Logic Programming*, (pp. 1–17).
- Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., & De Raedt, L. (2014). Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, *arXiv preprint arXiv:1304.6810*.
- Fuhr, N. (2000). Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society of Information Sciences*, *51*(2), 95–110.
- Gelfond, M. & Lifschitz, V. (1988). The stable model semantics for logic programming. In *International Conference and Symposium on Logic Programming*, (pp. 1070–1080).
- Gelfond, M., N, N. R., & Zhu, W. (2006). Combining logical and probabilistic reasoning. In *Proceedings of AAAI 06 Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, (pp. 50–55).
- Gelfond, M. & Rushton, N. (2010). Causal and probabilistic reasoning in p-log: Heuristics, probabilities and causality. In R. Dechter, H. Geffner, & J. Halpern (Eds.), *A Tribute to Judea Pearl* (pp. 337–359). College Publications.
- Gorlin, A., Ramakrishnan, C. R., & Smolka, S. A. (2012). Model checking with probabilistic tabled logic programming. *Theory and Practice of Logic Programming*, *12*(4-5), 681–700.
- Greco, S., Molinaro, C., & Trubitsyna, I. (2013). Bounded programs: A new decidable class of logic programs with function symbols. In *International Joint Conference on Artificial Intelligence*, (pp. 926–932).
- Gutmann, B., Kimmig, A., Kersting, K., & De Raedt, L. (2008). Parameter learning in probabilistic databases: A least squares approach. In *European Conference on Machine Learning and Knowledge Discovery in Databases*, (pp. 473–488).
- Gutmann, B., Thon, I., & Raedt, L. D. (2011). Learning the parameters of probabilistic logic programs from interpretations. In *European Conference on Machine Learning and Knowledge Discovery in Databases*, (pp. 581–596).
- Halpern, J. H. (2003). *Reasoning About Uncertainty*. MIT Press.
- Islam, M., Ramakrishnan, C. R., & Ramkrishnan, I. V. (2012). Inference in probabilistic logic programs with continuous random variables. *Theory and Practice of Logic Programming*, *12*(4-5), 505–523.
- Kersting, K. & Raedt, L. D. (2001). Towards combining inductive logic programming with Bayesian networks. In *International Conference on Inductive Logic Programming*, (pp. 118–131).

- Kimig, A., Demoen, B., De Raedt, L., Costa, V. S., & Rocha, R. (2011). On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11(2-3), 235–262.
- Koller, D. & Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
- Kolmogorov, A. N. (1950). *Foundations of the Theory of Probability*. New York: Chelsea Publishing Company.
- Kyburg, H. & Teng, C. (2001). *Uncertain Inference*. Cambridge University Press.
- Meert, W., Struyf, J., & Blockeel, H. (2008). Learning ground CP-Logic theories by leveraging Bayesian network learning techniques. *Fundamenta Informaticae*, 89(1), 131–160.
- Meert, W., Struyf, J., & Blockeel, H. (2009). CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In *International Conference on Inductive Logic Programming*.
- Nilsson, N. J. (1986). Probabilistic logic. *Artificial Intelligence*, 28(1), 71–87.
- Papadimitriou, C. (1994). *Computational Complexity*. Addison-Wesley.
- Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.
- Pearl, J. (2000). *Causality*. Cambridge University Press.
- Poole, D. (1993a). Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3), 377–400.
- Poole, D. (1993b). Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1).
- Poole, D. (1997). The Independent Choice Logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2), 7–56.
- Poole, D. (2000). Abducing through negation as failure: stable models within the independent choice logic. *Journal of Logic Programming*, 44(1-3), 5–35.
- Poole, D. & Zhang, N. (2003). Exploiting contextual independence in probabilistic inference. *Journal of Artificial Intelligence Research*, 18, 266–313.
- Rauzy, A., Châtelet, E., Dutuit, Y., & Bérenguer, C. (2003). A practical comparison of methods to assess sum-of-products. *Reliability Engineering and System Safety*, 79(1), 33–42.
- Richardson, M. & Domingos, P. (2006). Markov logic networks. *Machine Learning*, 62(1-2), 107–136.
- Riguzzi, F. (2007). A top-down interpreter for LPAD and CP-logic. In *Congress of the Italian Association for Artificial Intelligence*, volume 4733 of *LNAI*, (pp. 109–120). Springer.
- Riguzzi, F. (2009). Extended semantics and inference for the Independent Choice Logic. *Logic Journal of the IGPL*, 17(6), 589–629.
- Riguzzi, F. (2011). Optimizing inference for probabilistic logic programs exploiting independence and exclusiveness. In *Italian Convention on Computational Logic*.
- Riguzzi, F. & Swift, T. (2011). The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming*, 11(4-5), 433–449.

- Riguzzi, F. & Swift, T. (2013). Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming*, 13(2), 279–302.
- Russell, B. (1967). Mathematical logic as based on the theory of types. In J. van Heikenoort (Ed.), *From Frege to Godel* (pp. 150–182). Harvard Univ. Press.
- Santos Costa, V., Damas, L., & Rocha, R. (2012). The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2), 5–34.
- Santos Costa, V., Page, D., Qazi, M., & Cussens, J. (2003). CLP(BN): Constraint logic programming for probabilistic knowledge. In *Conference on Uncertainty in Artificial Intelligence*, (pp. 517–524).
- Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*, (pp. 715–729).
- Sato, T. (2011). A general MCMC method for Bayesian inference in logic-based probabilistic modeling. In *International Joint Conference on Artificial Intelligence*.
- Sato, T. & Kameya, Y. (1997). PRISM: A language for symbolic-statistical modeling. In *International Joint Conference on Artificial Intelligence*, (pp. 1330–1339).
- Sato, T. & Kameya, Y. (2001). Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15, 391–454.
- Sato, T. & Kameya, Y. (2008). New advances in logic-based probabilistic modeling by PRISM. In *Probabilistic Inductive Logic Programming*, (pp. 118–155).
- Sato, T., Kameya, Y., & Zhou, N.-F. (2005). Generative modeling with failure in PRISM. In *International Joint Conference on Artificial Intelligence*, (pp. 847–852).
- Sato, T. & Meyer, P. (2012). Tabling for infinite probability computation. In *International Conference on Logic Programming*, volume 17 of *LIPICs*, (pp. 348–358).
- Sato, T., Zhou, N.-F., Kameya, Y., & Izumi, Y. (2010). PRISM User’s Manual (Version 2.0). <http://sato-www.cs.titech.ac.jp/prism/download/prism20.pdf>.
- Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., & Toivonen, H. (2006). Link discovery in graphs derived from biological databases. In *International Workshop on Data Integration in the Life Sciences*, volume 4075 of *LNCs*, (pp. 35–49). Springer.
- Swift, T. & Warren, D. S. (2012). XSB: Extending the power of Prolog using tabling. *Theory and Practice of Logic Programming*, 12(1-2), 157–187.
- Van den Broeck, G., Meert, W., & Darwiche, A. (2014). Skolemization for weighted first-order model counting. In *KR 2014*.
- Van Gelder, A., Ross, K. A., & Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3), 620–650.
- Vennekens, J., Denecker, M., & Bruynooghe, M. (2009). CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3), 245–308.

- Vennekens, J. & Verbaeten, S. (2003). Logic programs with annotated disjunctions. Technical Report CW386, KU Leuven.
- Vennekens, J., Verbaeten, S., & Bruynooghe, M. (2004). Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, (pp. 195–209).
- Wellman, M. P., Breese, J. S., & Goldman, R. P. (1992). From knowledge bases to decision models. *The Knowledge Engineering Review*, 7(01), 35–53.
- Zhou, N. (2012). The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2), 189–218.