

# Probabilistic Inductive Querying Using ProbLog

*Luc De Raedt*  
*Angelika Kimmig*  
*Bernd Gutmann*  
*Kristian Kersting*  
*Vitor Santos Costa*  
*Hannu Toivonen*  
*Report CW 552, June 2009*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Probabilistic Inductive Querying Using ProbLog

*Luc De Raedt*  
*Angelika Kimmig*  
*Bernd Gutmann*  
*Kristian Kersting*  
*Vítor Santos Costa*  
*Hannu Toivonen*  
*Report CW552, June 2009*

Department of Computer Science, K.U.Leuven

## Abstract

We study how probabilistic reasoning and inductive querying can be combined within ProbLog, a recent probabilistic extension of Prolog. ProbLog can be regarded as a database system that supports both probabilistic and inductive reasoning through a variety of querying mechanisms. After a short introduction to ProbLog, we provide a survey of the different types of inductive queries that ProbLog supports, and show how it can be applied to the mining of large biological networks.

**Keywords :** Probabilistic Logic Programming, Statistical Relational Learning, Inductive Querying, Probabilistic Inference, Explanation Based Learning, Analogy Based Reasoning, Local Pattern Mining, Theory Compression, Parameter Estimation.

**CR Subject Classification :** I.2.6

# Probabilistic Inductive Querying Using ProbLog

Luc De Raedt<sup>1</sup>, Angelika Kimmig<sup>1</sup>, Bernd Gutmann<sup>1</sup>, Kristian Kersting<sup>2</sup>,  
Vítor Santos Costa<sup>3</sup>, and Hannu Toivonen<sup>4</sup>

<sup>1</sup> Department of Computer Science, Katholieke Universiteit Leuven  
`{firstname.lastname}@cs.kuleuven.be`

<sup>2</sup> Fraunhofer IAIS, Sankt Augustin `kristian.kersting@iais.fraunhofer.de`

<sup>3</sup> Faculdade de Ciências, Universidade do Porto `vsc@dcc.fc.up.pt`

<sup>4</sup> Department of Computer Science, University of Helsinki  
`hannu.toivonen@cs.helsinki.fi`

**Abstract.** We study how probabilistic reasoning and inductive querying can be combined within ProbLog, a recent probabilistic extension of Prolog. ProbLog can be regarded as a database system that supports both probabilistic and inductive reasoning through a variety of querying mechanisms. After a short introduction to ProbLog, we provide a survey of the different types of inductive queries that ProbLog supports, and show how it can be applied to the mining of large biological networks.

## 1 Introduction

In recent years, both probabilistic and inductive databases have received considerable attention in the literature. Probabilistic databases [1] allow representation of and reasoning about uncertain data, while inductive databases [2] aim at tight integration of data mining primitives in database query languages. Despite the current interest in these types of databases, there have, to the best of the authors' knowledge, been no attempts to integrate these two trends of research. This chapter wants to contribute to a better understanding of the issues involved by providing a survey of the developments around ProbLog [3], an extension of Prolog, which supports both inductive and probabilistic querying. ProbLog has been motivated by the need to develop intelligent tools for supporting life scientists analyzing large biological networks. The analysis of such networks typically involves uncertain data, requiring probabilistic representations and inference, as well as the need to find patterns in data, and hence, supporting data mining. ProbLog can be conveniently regarded as a probabilistic database supporting several types of inductive and probabilistic queries. This paper provides an overview of the different types of queries that ProbLog currently supports.

A ProbLog program defines a distribution over logic programs (or databases) by specifying for each fact (or tuple) the probability that it belongs to a randomly sampled program (or database), where probabilities are mutually independent. The semantics of ProbLog is then defined by the success probability of a query, which corresponds to the probability that the query succeeds in a randomly sampled program (or database). ProbLog is closely related to other probabilistic

logics and probabilistic databases that have been developed over the past two decades to face the general need of combining deductive abilities with reasoning about uncertainty, see e.g. [4–7]. The semantics of ProbLog is studied in Section 2.

We now give a first overview of the types of queries ProbLog currently supports. Throughout the chapter, we use the graph in Figure 1(a) for illustration. It contains several nodes (representing entities) as well as edges (representing relationships). Furthermore, the edges are probabilistic, that is, they are present only with the probability indicated.

**Probabilistic Inference** *What is the probability that a query succeeds?*

Given a ProbLog program and a query, the inference task is to compute the success probability of the query, that is, the probability that the query succeeds in a randomly sampled non-probabilistic subprogram of the ProbLog program. As one example query, consider computing the probability that there exists a proof of  $path(c, d)$  in Figure 1(a), that is, the probability that there is a path from  $c$  to  $d$  in the graph, which will have to take into account the probabilities of both possible paths. Computing and approximating the success probability of queries will be discussed in Section 3.

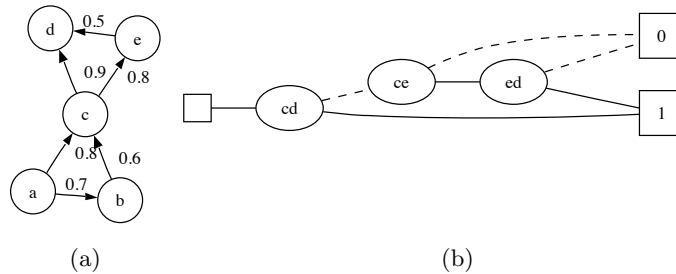
**Most Likely Explanation** *What is the most likely explanation for a query?*

There can be many possible explanations (or reasons) why a certain query may succeed. For instance, in the  $path(c, d)$  example, there are two explanations, corresponding to the two different paths from  $c$  to  $d$ . Often, one is interested in the most likely such explanations, as this provides insight into the problem at hand (here, the direct path from  $c$  to  $d$ ). Computing the most likely explanation realizes a form of probabilistic abduction, cf. [8], as it returns the most likely cause for the query to succeed.

The above two types of queries are *probabilistic*, that is, they use standard probabilistic inference methods adapted to the context of the ProbLog framework. The types of queries presented next are *inductive*, which means that they start from one or more examples (typically, ground facts such as  $path(c, d)$ ) describing particular relationships, and perform inferences about other examples or about patterns holding in the database.

**Analogy and Similarity Based Reasoning** *Which examples are most similar to a given example?*

In explanation based learning the goal is to find the most likely explanation for a particular example in the light of a background theory. The found explanation can then be used to retrieve similar examples, or to reason by analogy. The most likely explanation acts as a kind of local pattern that is specific to the given example(s), thereby allowing the user to get insight into particular relationships. Within ProbLog, the traditional approach on explanation based learning is put into a new probabilistic perspective. In our example graph, given the definition of  $path$  in the background theory and an example such as  $path(c, d)$ , probabilistic explanation based learning finds



**Fig. 1.** (a) Example of a probabilistic graph: edge labels indicate the probability that the edge is part of the graph. (b) Binary Decision Diagram encoding the DNF formula  $cd \vee (ce \wedge ed)$ , corresponding to the two proofs of query  $path(c,d)$  in the graph. An internal node labeled  $xy$  represents the Boolean variable for the edge between  $x$  and  $y$ , solid/dashed edges correspond to values true/false.

that a direct connection is the most likely explanation, which can then be used to retrieve and rank other directly connected examples.

**Local Pattern Mining** *Which queries are likely to succeed for a given set of examples?*

In local pattern mining the goal is to find those patterns that are likely to succeed on a set of examples, that is, instances of a specific relation *key*. This setting is a natural variant of the explanation based learning setting, but without the need for a background theory. The result is a kind of probabilistic relational association rule miner. On our example network, the local pattern miner could start, for instance, from the examples  $key(c,d)$  and  $key(a,c)$  and infer that there is a direct connection that is likely to exist for these examples. Again, resulting patterns can be used to retrieve similar examples and to provide insights into the likely commonalities amongst the examples.

**Theory Compression** *Which small theory best explains a set of examples?*

Theory compression aims at finding a small subset of a ProbLog theory (or network) that maximizes the likelihood of a given set of positive and negative examples. This problem is again motivated by the biological application, where scientists try to analyze enormous networks of links in order to obtain an understanding of the relationships amongst a typically small number of nodes. The idea now is to compress these networks as much as possible using a set of positive and negative examples. The examples take the form of relationships that are either interesting or uninteresting to the scientist. The result should ideally be a small network that contains the essential links and assigns high probabilities to the positive and low probabilities to the negative examples. This task is analogous to a form of theory revision [9, 10] where the only operation allowed is the deletion of rules or facts. Within the ProbLog theory compression framework, examples are true and false ground facts, and the task is to find a subset of a given ProbLog program that maximizes the likelihood of the examples.

### Parameter Estimation *Which parameters best fit the data?*

The goal is to learn the probabilities of facts from a given set of training examples. Each example consists of a query and target probability. This setting is challenging because the explanations for the queries, namely the proofs, are unknown. Using a modified version of the probabilistic inference algorithm, a standard gradient search can be used to find suitable parameters efficiently.

To demonstrate the usefulness of ProbLog for inductive and probabilistic querying, we have evaluated the different types of queries in the context of mining a large biological network containing about 1 million entities and about 7 million edges [11]. We will discuss this in more detail in Section 9.

This paper is organized as follows. In Section 2, we introduce the semantics of ProbLog and define the probabilistic queries; Section 3 discusses computational aspects and presents several algorithms (including approximation and Monte Carlo algorithms) for computing probabilities of queries. The following sections in turn consider each of the inductive queries listed above. Finally, Section 8 discusses the integration of ProbLog in the well-known implementation of YAP-Prolog, Section 9 provides a perspective on applying ProbLog on biological network mining, and Section 10 concludes.

## 2 ProbLog: Probabilistic Prolog

In this section, we present ProbLog and its semantics and then introduce two types of probabilistic queries: the *success probability* and the *explanation probability*.

A ProbLog program consists of a set of labeled facts  $p_i :: c_i$  together with a set of definite clauses. Each ground instance (that is, each instance not containing variables) of such a fact  $c_i$  is true with probability  $p_i$ , where all probabilities are assumed mutually independent. The definite clauses allow the user to add arbitrary *background knowledge* (BK).

Figure 1(a) shows a small probabilistic graph that we shall use as running example in the text. It can be encoded in ProbLog as follows:

$$\begin{array}{lll} 0.8 :: \text{edge}(\text{a}, \text{c}). & 0.7 :: \text{edge}(\text{a}, \text{b}). & 0.8 :: \text{edge}(\text{c}, \text{e}). \\ 0.6 :: \text{edge}(\text{b}, \text{c}). & 0.9 :: \text{edge}(\text{c}, \text{d}). & 0.5 :: \text{edge}(\text{e}, \text{d}). \end{array}$$

Such a probabilistic graph can be used to sample subgraphs by tossing a coin for each edge. A ProbLog program  $T = \{p_1 :: c_1, \dots, p_n :: c_n\} \cup BK$  defines a probability distribution over subprograms  $L \subseteq L_T = \{c_1, \dots, c_n\}$ :

$$P(L|T) = \prod_{c_i \in L} p_i \prod_{c_i \in L_T \setminus L} (1 - p_i).$$

We extend our example with the following background knowledge:

$$\begin{array}{l} \text{path}(\text{X}, \text{Y}) : - \text{edge}(\text{X}, \text{Y}). \\ \text{path}(\text{X}, \text{Y}) : - \text{edge}(\text{X}, \text{Z}), \text{path}(\text{Z}, \text{Y}). \end{array}$$

We can then ask for the probability that there exists a path between two nodes, say  $c$  and  $d$ , in our probabilistic graph, that is, we query for the probability that a randomly sampled subgraph contains the edge from  $c$  to  $d$ , or the path from  $c$  to  $d$  via  $e$  (or both of these). Formally, the *success probability*  $P_s(q|T)$  of a query  $q$  in a ProbLog program  $T$  is defined as

$$P_s(q|T) = \sum_{L \subseteq L_T} P(q|L) \cdot P(L|T), \quad (1)$$

where  $P(q|L) = 1$  if there exists a  $\theta$  such that  $L \cup BK \models q\theta$ , and  $P(q|L) = 0$  otherwise. In other words, the success probability of query  $q$  is the probability that the query  $q$  is *provable* in a randomly sampled logic program.

As a consequence, the probability of a *specific* proof, also called *explanation*, corresponds to that of sampling a logic program  $L$  that contains all the facts needed in that explanation or proof. The *explanation probability*  $P_x(q|T)$  is defined as the probability of the most likely explanation or proof of the query  $q$

$$P_x(q|T) = \max_{e \in E(q)} P(e|T) = \max_{e \in E(q)} \prod_{c_i \in e} p_i, \quad (2)$$

where  $E(q)$  is the set of all explanations for query  $q$  [12].

In our example, the set of all explanations for  $path(c, d)$  contains the edge from  $c$  to  $d$  (with probability 0.9) as well as the path consisting of the edges from  $c$  to  $e$  and from  $e$  to  $d$  (with probability  $0.8 \cdot 0.5 = 0.4$ ). Thus,  $P_x(path(c, d)|T) = 0.9$ .

The ProbLog semantics is essentially a distribution semantics [13]. Sato has rigorously shown that this class of programs defines a joint probability distribution over the set of possible least Herbrand models of the program, where each possible least Herbrand model corresponds to the least Herbrand model of the background knowledge  $BK$  together with a subprogram  $L \subseteq L_T$ ; for further details we refer to [13]. The distribution semantics has been used widely in the literature; see e.g. [4–7].

### 3 Probabilistic Inference

In this section, we present various algorithms and techniques for performing probabilistic inference in ProbLog, that is computing the success and explanation probabilities of queries.

Computing the *success probability* of a query using Equation (1) directly is infeasible for all but the tiniest programs; [3] presents a method involving two steps. The first step computes the proofs of the query  $q$  in the logical part of the theory  $T$ , that is, in  $L_T \cup BK$ . This step is akin to that performed for pD by [6]. The result will be a DNF formula. The second component employs Binary Decision Diagrams [14] to compute the probability of this formula. Let us now explain these two steps in more detail.

The first step employs SLD-resolution, as in Prolog, to obtain all different proofs. As an example, the SLD-tree for the query  $?- path(c, d)$ . is depicted in Figure 2. Each successful proof in the SLD-tree uses a set of facts  $\{p_1 \dots$

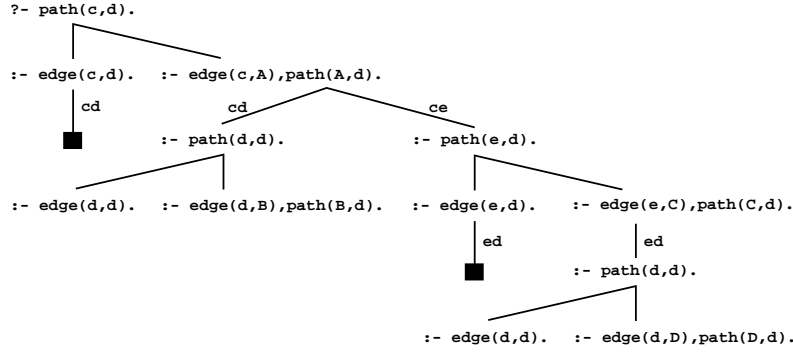


Fig. 2. SLD-tree for query  $\text{path}(c,d)$ .

$d_1, \dots, p_k :: d_k \} \subseteq T$ . These facts are necessary for the proof, and the proof is independent of other probabilistic facts in  $T$ .

Let us now introduce a Boolean random variable  $b_i$  for each fact  $p_i :: c_i \in T$ , indicating whether  $c_i$  is in logic program, that is,  $b_i$  has probability  $p_i$  of being true. The probability of a particular proof involving facts  $\{p_{i_1} :: d_{i_1}, \dots, p_{i_k} :: d_{i_k} \} \subseteq T$  is then the probability of the conjunctive formula  $b_{i_1} \wedge \dots \wedge b_{i_k}$ . Since a goal can have multiple proofs, the success probability of query  $q$  equals the probability that the disjunction of these conjunctions is true. This yields

$$P_s(q|T) = P \left( \bigvee_{e \in E(q)} \bigwedge_{b_i \in cl(e)} b_i \right) \quad (3)$$

where  $E(q)$  denotes the set of proofs or explanations of the goal  $q$  and  $cl(e)$  denotes the set of Boolean variables representing ground facts used in the explanation  $e$ . Thus, the problem of computing the success probability of a ProbLog query can be reduced to that of computing the probability of a DNF formula. The formula corresponding to our example query  $\text{path}(c,d)$  is  $cd \vee (ce \wedge ed)$ , where we use  $xy$  as Boolean variable representing  $\text{edge}(x,y)$ .

Computing the probability of DNF formulae is an NP-hard problem, as the different conjunctions need not be independent. Indeed, even under the assumption of independent variables used in ProbLog, the different conjunctions are not mutually exclusive and may overlap. Various algorithms have been developed to tackle this problem, which is known as the disjoint-sum-problem. The pD-engine HySpirit [6] uses the inclusion-exclusion principle to tackle this problem, which is reported to scale to about ten proofs. As the type of application considered here often requires to deal with hundreds or thousands of proofs, the second step of our implementation employs Binary Decision Diagrams (BDDs) [14], an efficient graphical representation of a Boolean function over a set of variables which scales to tens of thousands of proofs, see Section 8 for more details. Figure 1(b) shows the BDD corresponding to  $cd \vee (ce \wedge ed)$ , the formula of the example query  $\text{path}(c,d)$ . Given a BDD, it is easy to compute the probability of the correspond-

---

**Algorithm 1** Calculating success probability by traversing BDD.

---

```
function PROBABILITY(BDD node  $n$ )  
  If  $n$  is the 1-terminal return 1  
  If  $n$  is the 0-terminal return 0  
  let  $h$  and  $l$  be the high and low children of  $n$   
   $prob(h) :=$  call PROBABILITY( $h$ )  
   $prob(l) :=$  call PROBABILITY( $l$ )  
  return  $p_n \cdot prob(h) + (1 - p_n) \cdot prob(l)$ 
```

---

ing Boolean function by traversing the BDD from the root node to a leaf. At each inner node, probabilities from both children are calculated recursively and combined afterwards as shown in Algorithm 1. In practice, memorization of intermediate results is used to avoid the recomputation at nodes that are shared between multiple paths.

Whereas the computation of the *success probability*  $P_s$  is computationally hard because of the DNF formula, the *explanation probability*  $P_x$  is much easier to compute because it corresponds to computing the probability of a conjunctive formula only, so that the disjoint-sum-problem does not arise. Calculating the explanation probability can easily be realized using a best-first search, guided by the probability of the current derivation, through standard logic programming techniques based on the SLD-tree [15].

As the size of the DNF formula grows with the number of proofs, its evaluation can become fairly expensive, and finally infeasible. For instance, when searching for paths in graphs or networks, even in small networks with a few dozen edges there are easily  $O(10^6)$  possible paths between two nodes. ProbLog therefore includes several approximation methods.

*Bounded Approximation* The first approximation algorithm, similar to the one proposed in [3], uses DNF formulae to obtain both an upper and a lower bound on the probability of a query. It is related to work by [8] in the context of PHA, but adapted towards ProbLog. The algorithm uses an incomplete SLD-tree, i.e. an SLD-tree where branches are only extended up to a given probability threshold<sup>1</sup>, to obtain DNF formulae for the two bounds. The lower bound formula  $d_1$  represents all proofs with a probability above the current threshold. The upper bound formula  $d_2$  additionally includes all derivations that have been stopped due to reaching the threshold, as these still *may* succeed. The algorithm proceeds in an iterative-deepening manner, starting with a high probability threshold and successively multiplying this threshold with a fixed shrinking factor until the difference between the current bounds becomes sufficiently small. As  $d_1 \models d \models d_2$ , where  $d$  is the Boolean DNF formula corresponding to the full SLD-tree of the query, the success probability is guaranteed to lie in the interval  $[P(d_1), P(d_2)]$ .

As an illustration, consider a probability bound of 0.9 for the SLD-tree in Figure 2. In this case,  $d_1$  encodes the left success path while  $d_2$  additionally

---

<sup>1</sup> Using a probability threshold instead of the depth bound of [3] has been found to speed up convergence, as upper bounds are tighter on initial levels.

encodes the path up to  $path(e, d)$ , i.e.  $d_1 = cd$  and  $d_2 = cd \vee ce$ , whereas the formula for the full SLD-tree is  $d = cd \vee (ce \wedge ed)$ .

*K-Best* Using a fixed number of proofs to approximate the probability allows for better control of the overall complexity, which is crucial if large numbers of queries have to be evaluated e.g. in the context of parameter learning, cf. Section 7. [16] therefore introduce the  $k$ -probability  $P_k(q|T)$ , which approximates the success probability by using the  $k$  best (that is, most likely) explanations instead of all proofs when building the DNF formula used in Equation (3):

$$P_k(q|T) = P \left( \bigvee_{e \in E_k(q)} \bigwedge_{b_i \in cl(e)} b_i \right) \quad (4)$$

where  $E_k(q) = \{e \in E(q) | P_x(e) \geq P_x(e_k)\}$  with  $e_k$  the  $k$ th element of  $E(q)$  sorted by non-increasing probability. Setting  $k = \infty$  and  $k = 1$  leads to the success and the explanation probability respectively. Finding the  $k$  best proofs can be realized using a simple branch-and-bound approach; cf. also [7].

To illustrate  $k$ -probability, we consider again our example graph, but this time with query  $path(a, d)$ . This query has four proofs, represented by the conjunctions  $ac \wedge cd$ ,  $ab \wedge bc \wedge cd$ ,  $ac \wedge ce \wedge ed$  and  $ab \wedge bc \wedge ce \wedge ed$ , with probabilities 0.72, 0.378, 0.32 and 0.168 respectively. As  $P_1$  corresponds to the explanation probability  $P_x$ , we obtain  $P_1(path(a, d)) = 0.72$ . For  $k = 2$ , overlap between the best two proofs has to be taken into account: the second proof only adds information if the first one is absent. As they share edge  $cd$ , this means that edge  $ac$  has to be missing, leading to  $P_2(path(a, d)) = P((ac \wedge cd) \vee (\neg ac \wedge ab \wedge bc \wedge cd)) = 0.72 + (1 - 0.8) \cdot 0.378 = 0.7956$ . Similarly, we obtain  $P_3(path(a, d)) = 0.8276$  and  $P_k(path(a, d)) = 0.83096$  for  $k \geq 4$ .

*Monte Carlo* As an alternative approximation technique without BDDs, [17] propose a Monte Carlo method. The algorithm repeatedly samples a logic program from the ProbLog program and checks for the existence of some proof of the query of interest. The fraction of samples where the query is provable is taken as an estimate of the query probability, and after each  $m$  samples the 95% confidence interval is calculated. Although confidence intervals do not directly correspond to the exact bounds used in bounded approximation, the same stopping criterion is employed, that is, the Monte Carlo simulation is run until the width of the confidence interval is at most  $\delta$ . Such an algorithm (without the use of confidence intervals) was suggested already by Dantsin [4], although he does not report on an implementation. It was also used in the context of networks (not Prolog programs) by [11].

## 4 Probabilistic Explanation Based Learning

In this section, we address the question as to how to find examples that are similar or analogous to a given example. To this aim, we employ a background

theory that allows us to compute a most likely explanation for the example, to generalize that explanation and to use it to retrieve (and possibly rank) other examples matching the generalized explanation. As such, ProbLog’s probabilistic explanation based learning technique (PEBL) [12] combines two types of queries, namely finding the *most likely explanation* for an example and *reasoning by analogy*, which is the process of finding examples with a similar explanation. Probabilistic explanation based learning extends the idea of explanation based learning (EBL) to a probabilistic framework.

The central idea of explanation based learning [18, 19] as conveniently formalized for Prolog [20, 21] is to compute a generalized explanation from a concrete proof of an example. Explanations use only so-called *operational* predicates, i.e. predicates that capture essential characteristics of the domain of interest and should be easy to prove. Operational predicates are to be declared by the user as such.

The problem of probabilistic explanation based learning can be sketched as follows.

- Given** a positive example  $e$  (a ground fact), a ProbLog theory  $T$ , and declarations that specify which predicates are operational,  
**Find** a clause  $c$  such that  $T \models c$  (in the logical sense, so interpreting  $T$  as a Prolog program),  $body(c)$  contains only operational predicates, there exists a substitution  $\theta$  such that  $head(c)\theta = e$  and  $body(c)\theta$  is the most likely explanation for  $e$  given  $T$ .

Following the work by [20, 21], explanation based learning starts from a definite clause theory  $T$ , that is a pure Prolog program, and an example in the form of a ground atom  $p(t_1, \dots, t_n)$ . It then constructs a refutation proof of the example using SLD-resolution. Explanation based learning will generalize this proof to obtain a generalized explanation. This is realized performing the same SLD-resolution steps as in the proof for the example, but starting from the variable goal, i.e.  $p(X_1, \dots, X_n)$  where the  $X_i$  are different variables. The only difference is that in the general proof atoms  $q(s_1, \dots, s_r)$  for operational predicates  $q$  in a goal  $?-g_1, \dots, g_i, q(s_1, \dots, s_r), g_{i+1}, \dots, g_n$  are not resolved away. Also, the proof procedure stops when the goal contains only atoms for operational predicates. The resulting goal provides a *generalized explanation* for the example. In terms of the SLD-resolution proof tree, explanation based learning cuts off branches below operational predicates. It is easy to implement the explanation based proof procedure as a meta-interpreter for Prolog [21, 20].

Reconsider the example of Figure 1(a), ignoring the probability labels for now. We define `edge/2` to be the only operational predicate, and use `path(c,d)` as training example. EBL proves this goal using one instance of the operational predicate, namely `edge(c,d)`, leading to the explanation `edge(X,Y)` for the generalized example `path(X,Y)`. The result can be represented as the clause `exp_path(X,Y) ← edge(X,Y)`. Using the second possible proof of `path(c,d)` instead, we would obtain `exp_path(X,Y) ← edge(X,Z), edge(Z,Y)`. We will call such clauses *explanation clauses*. To be able to identify the examples covered by this clause, we rename the predicate in the head of explanation clauses.

Probabilistic explanation based learning (PEBL) extends EBL to probabilistic logic representations, computing the generalized explanation from the most likely proof of an example as determined by the explanation probability  $P_x(q|T)$  (2). PEBL thus returns the first explanation clause in our example.

The probability of a given single proof is calculated simply as the product  $\prod_i p_i$  of probability labels of all ground facts  $c_i$  used (at least once) in that proof. This corresponds to the probability of randomly sampling a subprogram of the ProbLog program that contains all facts used in the proof and thus admits the proof. The set of facts  $C$  used in the proof of the example that is to be generalized can be partitioned into two sets. Indeed, define  $G = \{c|c \in C \text{ such that } c \text{ is used in the generalized proof}\}$ , and  $E = C - G$ , i.e.  $E$  contains the example-specific facts used to prove operational predicates. We then have that

$$\prod_{c_i \in C} p_i = \prod_{c_j \in G} p_j \prod_{c_k \in E} p_k.$$

Thus, the probability of the original proof is equal to the product of the probability of the generalized proof and the probability of the goals for the operational predicates in the example.

Computing the most likely proof for a given goal in ProbLog is straightforward: instead of traversing the SLD-tree in a left-to-right depth-first manner as in Prolog, nodes are expanded in order of the probability of the derivation leading to that node. This realizes a best-first search with the probability of the current proof as an evaluation function. We use iterative deepening in our implementation to avoid memory problems. The algorithm can also be used to return the  $k$  most probable structurally distinct explanations.

Probabilistic explanation based learning as incorporated in ProbLog offers natural solutions to two issues traditionally discussed in the context of explanation based learning [18, 22]. The first one is the *multiple explanation* problem, which is concerned with choosing the explanation to be generalized for examples having multiple proofs. The use of a sound probabilistic framework naturally deals with this issue by selecting the *most likely* proof. The second problem is that of *generalizing from multiple examples*, another issue that received considerable attention in traditional explanation based learning. To realize this in our setting, we modify the best-first search algorithm so that it searches for the most likely generalized explanation shared by the  $n$  examples  $e_1, \dots, e_n$ . Starting from the variabilized atom  $e$ , we compute  $n + 1$  SLD-resolution derivations in parallel. A resolution step resolving an atom for a non-operational predicate in the generalized proof for  $e$  is allowed only when the same resolution step can also be applied to each of the  $n$  parallel derivations. Atoms corresponding to operational predicates are – as sketched above – not resolved in the generalized proof, but it is nevertheless required that for each occurrence of these atoms in the  $n$  parallel derivations, there exists a resolution derivation.

Consider again our running example, and assume that we now want to construct a common explanation for  $\text{path}(c,d)$  and  $\text{path}(b,e)$ . We thus have to simultaneously prove both examples and the variabilized goal  $\text{path}(X,Y)$ .

After resolving all three goals with the first clause for `path/2`, we reach the first instance of the operational predicate `edge/2` and thus have to prove both `edge(c,d)` and `edge(b,e)`. As proving `edge(b,e)` fails, the last resolution step is rejected and the second clause for `path/2` used instead. Continuing this process finally leads to the explanation clause `exp_path(X,Y) ← edge(X,Z), edge(Z,Y)`.

Because PEBL generates a generalized explanation, which can be turned into an explanation clause, the technique can be employed to identify similar instances and to reason by analogy. Indeed, by asserting such an explanation clause and posing queries to the resulting predicate one obtains similar examples. Furthermore, the success probabilities of the examples can be used to rank them according to likelihood, and hence, similarity. As an example, using the explanation clause `exp_path(X,Y) ← edge(X,Z), edge(Z,Y)`, we obtain `exp_path(a,d)` ( $0.8 \cdot 0.9 = 0.72$ ), `exp_path(a,e)` ( $0.8 \cdot 0.8 = 0.64$ ), and so forth.

We refer to [12] for experiments in the context of biological networks.

## 5 Local Pattern Mining

In this section, we show how local pattern mining can be adapted towards probabilistic databases such as ProbLog. Even though local pattern mining is related to probabilistic explanation based learning, there are some important differences. Indeed, probabilistic explanation based learning typically employs a single positive example and a background theory to compute a generalized explanation of the example. Local pattern mining, on the other hand, does not rely on a background theory or declarations of operational predicates, uses a set of examples rather than a single one, and computes a set of patterns (or clauses) satisfying some conditions. As in probabilistic explanation based learning, the discovered patterns can be used to retrieve and rank further examples, realizing a kind of similarity based reasoning or reasoning by analogy.

Our approach to probabilistic local pattern mining [23] builds upon multi-relational query mining techniques [24], extending them towards probabilistic databases. We use ProbLog to represent databases and queries, abbreviating vectors of variables as  $\mathbf{X}$ . We assume a designated table  $ID$  containing the set of tuples to be characterized using queries, and restrict the language  $\mathcal{L}$  of patterns to the set of clauses of the form

$$r(\mathbf{X}) : -ID(\mathbf{X}), l_1, \dots, l_n \tag{5}$$

where the  $l_i$  are positive atoms or conjunctive conditions. Additional syntactic or semantic restrictions, called *bias*, can be imposed on the form of conjunctive queries by explicitly specifying the language  $\mathcal{L}$ , cf. [25, 26, 24].

Two forms of local pattern mining are typically distinguished.

*Frequent Query Mining* (in the non-probabilistic case) aims at finding all queries satisfying a selection predicate  $\phi$ . It can be formulated as follows, cf. [24, 26]:

**Given** a language  $\mathcal{L}$  containing queries of the form (5), a database  $\mathcal{D}$  including the designated relation  $ID$ , and an anti-monotonic selection predicate  $\phi$

---

**Algorithm 2** AM mining

---

**Inputs:** language  $\mathcal{L}$ , refinement operator  $\rho$ , database  $\mathcal{D}$ , anti-monotonic selection predicate  $\phi$

**Outputs:**  $Th(\mathcal{L}, \mathcal{D}, \phi)$

$C_0 := \{q(\mathbf{X}) \leftarrow ID(\mathbf{X})\}$

$i := 0; F_0 := \emptyset; I_0 := \emptyset$

**while**  $C_i \neq \emptyset$  **do**

$F_i := \{h \in C_i \mid \phi(h, \mathcal{D}) = true\}$

$I_i := C_i - F_i$

$C_{i+1} := \{h \in \rho(h') \mid h' \in F_i, \neg \exists s \in \bigcup_j I_j : s \preceq h\}$

$i := i + 1$

**output**  $\cup_i F_i$

---

**Find** all queries  $q \in \mathcal{L}$  such that  $\phi(q, \mathcal{D}) = true$ .

A prominent example of an anti-monotonic selection predicate is minimum frequency, requiring a minimum number of tuples covered. Anti-monotonicity is based on a generality relation between patterns. We employ *OI*-subsumption [27], as the corresponding notion of subgraph isomorphism is favorable within the intended application in network mining.

*Correlated Pattern Mining* [28] uses both *positive* and *negative* examples, given as two designated relations  $ID^+$  and  $ID^-$  of the same arity, to find the top  $k$  patterns, that is, the  $k$  patterns scoring best w.r.t. a function  $\psi$ . The function  $\psi$  employed is convex, e.g. measuring a statistical significance criterion such as  $\chi^2$ , cf. [28], and measures the degree to which the pattern is statistically significant or unexpected. The task of correlated pattern mining is defined as:

**Given** a language  $\mathcal{L}$  containing queries of the form (5), a database  $\mathcal{D}$  including the designated relations  $ID^+$  and  $ID^-$ , and a correlation function  $\psi$

**Find**  $\arg_k \max_{q \in \mathcal{L}} \psi(q, \mathcal{D})$

*Multi-relational query miners* such as [24, 26] often follow a level-wise approach for frequent query mining [29]. In contrast to Apriori, instead of a “joining” operation, they employ a refinement operator  $\rho$  to compute more specific patterns. Additionally, they manage sets of both *frequent* and *infrequent* queries to take into account the specific language requirements imposed by  $\mathcal{L}$ .

A *refinement operator*  $\rho$  is a function  $\rho : \mathcal{L} \rightarrow 2^{\mathcal{L}}$  such that  $\forall q \in \mathcal{L} : \rho(q) \subseteq \{q' \in \mathcal{L} \mid q \preceq q'\}$ , where  $g \preceq s$  denotes that  $g$  subsumes  $s$  (or that  $s$  is a refinement of  $g$ ). For efficiency reasons it is important to use an *optimal* refinement operator, that is, an operator that ensures that there is exactly one path from the most general pattern to every other pattern in the search space. The key to developing an optimal refinement operator for conjunctive queries is to create a canonical form, see [26] for more details.

The standard algorithms for frequent and correlated pattern mining, adapted towards the use of the optimal operator, are shown in Figures 2 and 3.

---

**Algorithm 3** COR pattern mining

---

**Inputs:** language  $\mathcal{L}$ , refinement operator  $\rho$ , database  $\mathcal{D}$ , correlation measure  $\psi$  and upper bound function  $u$

**Outputs:**  $\arg_k \max_{q \in \mathcal{L}} \psi(q, \mathcal{D})$

$C_0 := \{q(\mathbf{X}) \leftarrow ID(\mathbf{X})\}$

$i := 0; F = \emptyset; I_0 := \emptyset;$

$t :=$  global lower bound on  $\psi$

**while**  $C_i \neq \emptyset$  **do**

$C_i := \{h \in C_i \mid \psi(h, \mathcal{D}) \geq t\}$

    Let  $q$  be the  $k$ -th best query on  $C_i \cup F$

$F := \{h \in C_i \cup F \mid \psi(h, \mathcal{D}) \geq \psi(q, \mathcal{D})\}$

$t := \psi(q, \mathcal{D})$

$I_i := C_i - F$

$C_{i+1} := \{h \in \rho(h') \mid h' \in C_i, u(h') \geq \psi(q, \mathcal{D}) \text{ and } \neg \exists s \in \bigcup_j I_j : s \preceq h\}$

$i := i + 1$

**output**  $F$

---

The first algorithm is an adaptation of the level-wise algorithm of [29] and the second one of Morishita and Sese's correlated pattern miner. This algorithm employs boundable correlation functions  $\psi$ , that is, that there exists a function  $u$  (different from a global maximum) such that  $\forall g, s \in \mathcal{L} : g \preceq s \rightarrow \psi(s) \leq u(g)$ . The algorithm then returns the top  $k$  patterns w.r.t. such a  $\psi$ .

We have adapted the multi-relational query mining algorithms to the use of a probabilistic databases. To this aim, we employ *probabilistic selection predicates* using either  $P_s(q(t)|\mathcal{D})$  (1) or  $P_x(q(t)|\mathcal{D})$  (2) as probabilistic membership function  $P(t|q, \mathcal{D})$ . Note that  $P$  is anti-monotonic: if  $q_1 \preceq q_2$  then  $P(t|q_1, \mathcal{D}) \geq P(t|q_2, \mathcal{D})$ . In correlated pattern mining, the aim is to find patterns with a high aggregated score on the positive examples and a low one on the negatives. As typical functions such as  $\chi^2$  rely on hard 0-1 decisions, they cannot easily be employed here. Instead, we consider class-sensitive aggregation functions: a *probabilistic frequency pf* also employed by [30] in the context of item-set mining, and a kind of *likelihood LL*:

$$pf(q, \mathcal{D}) = \sum_{t \in ID^+} P(t|q, \mathcal{D}) - \sum_{t \in ID^-} P(t|q, \mathcal{D}) \quad (6)$$

$$LL(q, \mathcal{D}) = \prod_{t \in ID^+} P(t|q, \mathcal{D}) \cdot \prod_{t \in ID^-} (1 - P(t|q, \mathcal{D})) \quad (7)$$

$$LL_n(q, \mathcal{D}) = \prod_{n(ID^+)} P(t|q, \mathcal{D}) \cdot \prod_{t \in ID^-} (1 - P(t|q, \mathcal{D})) \quad (8)$$

Here,  $n(ID^+)$  contains the  $n$  highest scoring tuples in  $ID^+$ . To work with unclassified instances, as in frequent pattern mining, we set  $ID^+ = ID$  and  $ID^- = \emptyset$ . We introduce  $LL_n$ , integrating a deterministic (anti-monotonic) selection predicate into a probabilistic correlation measure, as the likelihood (7) in combination with a non-zero threshold implies that *all* positive examples must be covered with

non-zero probability.  $LL_n$  requires that all queries return a minimum number of (positive) tuples. Note that whenever the membership function  $P$  is monotone, combining one of these functions with a minimum threshold leads to monotone selection predicates w.r.t.  $OI$ -subsumption. In correlated pattern mining, omitting the scores of negative examples provides an upper bound in all cases.

To illustrate this, consider again our small example graph from Figure 1(a). Assume that node  $c$  is a positive example, whereas node  $e$  is a negative one. Using the pattern language of connected graphs with a designated node matching the example node and probabilistic frequency with  $P_x$ , the best pattern obtained is  $ex(X) : -edge(X, Y), edge(X, Z)$  with score  $0.8 \cdot 0.9 - 0 = 0.72$ , as  $e$  is not covered due to the use of  $OI$ -subsumption, which forces  $Y$  and  $Z$  to be mapped to different nodes.

## 6 Theory Compression

In this section, we investigate how to obtain a small compressed database that contains the essential links w.r.t. a given set of positive and negative examples. This is useful for scientists trying to understand and analyze large biological networks as it allows them to identify the most relevant components of the theory.

The technique on which we build is that of theory compression [31], where the goal is to remove as many edges, i.e., facts as possible from the theory while still explaining the (positive) examples. The examples, as usual, take the form of relationships that are either interesting or uninteresting to the scientist. The resulting theory should contain the essential facts, assign high probabilities to the positive and low probabilities to the negative examples, and it should be a lot smaller and hence easier to understand and to employ by the scientists than the original theory.

As an illustrative example, consider again the graph in Figure 1(a) together with the definition of the path predicate given earlier. Assume now that we just confirmed that  $path(a, d)$  is of interest and that  $path(a, e)$  is not. We can then try to find a small graph (containing  $k$  or less edges) that best matches these observations. In our example, we would first remove the edges connecting  $e$  to the rest of the graph, as they strongly contribute to proving the negative example, while the positive example still has likely proofs in the resulting graph.

Before introducing the ProbLog theory compression problem, it is helpful to consider the corresponding problem in a purely logical setting, i.e., ProbLog programs where all facts are part of the background knowledge. In this case, the theory compression task coincides with a form of theory revision<sup>2</sup> [9, 10] where the only operation allowed is the deletion of rules or facts: **given** a set of positive and negative examples in the form of true and false facts, **find** a theory that best explains the examples, i.e., one that scores best w.r.t. a function such as accuracy. At the same time, the theory should be *small*, that is it should contain less than

<sup>2</sup> The analogy explains why we have formalized the theory compression task within the ProbLog framework.

$k$  facts. So, *logical* theory compression aims at finding a small theory that best explains the examples. As a result the compressed theory should be a better fit w.r.t. the data but should also be much easier to understand and to interpret. This holds in particular when starting with large networks containing thousands of nodes and edges and then obtaining a small compressed graph that consists of say 20 edges only. In biological databases such as the ones considered in this chapter, scientists can easily analyse the interactions in such small networks but have a very hard time with the large networks. The *ProbLog Theory Compression Problem* is now an adaptation of the traditional theory revision (or compression) problem towards probabilistic Prolog programs. Intuitively, we are interested in finding a small number of facts (at most  $k$  many) that maximizes the likelihood of the examples. More formally:

**Given** a ProbLog theory  $S$ , sets  $P$  and  $N$  of positive and negative examples in the form of independent and identically-distributed (iid) ground facts, and a constant  $k \in \mathbb{N}$ ,

**Find** a theory  $T \subseteq S$  of size at most  $k$  ( $|T| \leq k$ ) that has a maximum likelihood  $\mathcal{L}$  w.r.t. the examples  $E = P \cup N$ , i.e.,  $T = \arg \max_{T \subseteq S \wedge |T| \leq k} \mathcal{L}(E|T)$ , where

$$\mathcal{L}(E|T) = \prod_{e \in P} P(e|T) \cdot \prod_{e \in N} (1 - P(e|T)) \quad (9)$$

In other words, we use a ProbLog theory  $T$  to specify the conditional class distribution, i.e., the probability  $P(e|T)$  that any given example  $e$  is positive<sup>3</sup>. Because the examples are assumed to be iid the total likelihood is obtained as a simple product.

Despite its intuitive appeal, using the likelihood as defined in Eq. (9) has some subtle downsides. For an optimal ProbLog theory  $T$ , the probability of the positives is as close to 1 as possible, and for the negatives as close to 0 as possible. In general, however, we want to allow for misclassifications (with a high cost in order to avoid overfitting) to effectively handle noisy data and to obtain smaller theories. Furthermore, the likelihood function can become 0, e.g., when a positive example is not covered by the theory at all. To overcome these problems, we slightly redefine  $P(e|T)$  in Eq. (9) as

$$\hat{P}(e|T) = \max(\min[1 - \epsilon, P(e|T)], \epsilon) \quad (10)$$

for some constant  $\epsilon > 0$  specified by the user.

The compression approach can efficiently be implemented following a two-steps strategy as shown in Algorithm 4. In a first step, we compute the BDDs for all given examples. Then, we use these BDDs in a second step to greedily remove facts. This compression approach is efficient since the (expensive) construction of the BDDs is performed only once per example.

More precisely, the algorithm starts by calling the approximation algorithm sketched earlier, which computes the DNFs and BDDs for lower and upper

<sup>3</sup> Note that this is slightly different from specifying a distribution over (positive) examples.

---

**Algorithm 4** ProbLog theory compression

---

```
COMPRESS(input:  $S = \{p_1 :: c_1, \dots, p_n :: c_n\}, E, k, \epsilon)$ 
1  for all  $e \in E$ 
2      do call APPROXIMATE( $e, S, \delta$ ) to get  $DNF(low, e)$  and  $BDD(e)$ 
           where  $DNF(low, e)$  is the lower bound DNF formula for  $e$ 
           and  $BDD(e)$  is the BDD corresponding to  $DNF(low, e)$ 
3   $R := \{p_i :: c_i \mid b_i \text{ (indicator for fact } i \text{) occurs in a } DNF(low, e)\}$ 
4   $BDD(E) := \bigcup_{e \in E} \{BDD(e)\}$ 
5  improves := true
6  while ( $|R| > k$  or improves) and  $R \neq \emptyset$ 
7      do  $ll := \text{LIKELIHOOD}(R, BDD(E), \epsilon)$ 
8           $i := \arg \max_{i \in R} \text{LIKELIHOOD}(R - \{i\}, BDD(E), \epsilon)$ 
9          improves := ( $ll \leq \text{LIKELIHOOD}(R - \{i\}, BDD(E), \epsilon)$ )
10         if improves or  $|R| > k$ 
11             then  $R := R - \{i\}$ 
12  return  $R$ 
```

---

bounds (lines 1-4). In the second step, only the lower bound DNFs and BDDs are employed because they are simpler and, hence, more efficient to use. All facts used in at least one proof occurring in the (lower bound) BDD of some example constitute the set  $R$  of possible revision points. All other facts do not occur in any proof contributing to probability computation and hence can immediately be removed.

After the set  $R$  of revision points has been determined and the other facts removed the ProbLog theory compression algorithm performs a *greedy* search in the space of subsets of  $R$  (lines 7-11). At each step, the algorithm finds that fact whose deletion results in the best likelihood score, and then deletes it. As explained in more details in [31], this can efficiently be done using the BDDs computed in the preprocessing step: *set the probability of the node corresponding to the fact to 0 and recompute the probability of the BDD*. This process is continued until both  $|R| \leq k$  and deleting further facts does not improve the likelihood (lines 5-6).

For more details including experiments showing that ProbLog compression is not only of theoretical interest but is also applicable to various realistic problems in a biological link discovery domain we refer to [31].

## 7 Parameter Estimation

In this section, we address the question as to how to set the parameters of the ProbLog facts in the light of a set of examples. These examples consist of ground queries together with the desired probabilities, which implies that we are dealing with weighted examples such as  $0.6 : \text{locatedIn}(a, b)$  and  $0.7 : \text{interacting}(a, c)$

as used by Gupta and Sarawagi [32] and Chen *et al.* [33]. The parameter estimation technique should then determine the best values for the parameters. Our approach as implemented in LeProbLog [16] (Least Square Parameter Estimation for ProbLog) performs a gradient-based search to minimize the error on the given training data.

The problem tackled can be formalized as follows.

**Given** a ProbLog database  $T$  with unknown parameters and a set of training examples  $\{(q_i, \tilde{p}_i)\}_{i=1}^M$ ,  $M > 0$ , where each  $q_i \in \mathcal{H}$  is a query or proof and  $\tilde{p}_i$  is the  $k$ -probability of  $q_i$ ,

**Find** the parameters of the database  $T$  that minimize the mean squared error:

$$MSE(T) = \frac{1}{M} \sum_{1 \leq i \leq M} (P_k(q_i|T) - \tilde{p}_i)^2. \quad (11)$$

It is easy to see that minimizing the squared error corresponds to finding a maximum likelihood hypothesis, provided that each training example  $(q_i, \tilde{p}_i)$  is disturbed by a Gaussian error  $\tilde{p}_i$ , i.e.  $\tilde{p}_i = p(q_i) + e_i$ , with  $p(q_i)$  the actual probability of query  $q_i$  and  $e_i$  drawn independently from a zero-mean Gaussian. See [34, Chapter 6.4] for a detailed derivation.

Gradient descent is a standard way of minimizing a given error function. The tunable parameters are initialized randomly. Then, as long as the error did not converge, the gradient of the error function is calculated, scaled by the learning rate  $\eta$ , and subtracted from the current parameters. To get the gradient of the MSE, we apply the sum and chain rule to Eq. (11). This yields the partial derivative

$$\frac{\partial MSE(T)}{\partial p_j} = \frac{2}{M} \sum_{1 \leq i \leq M} \underbrace{(P_k(q_i|T) - \tilde{p}_i)}_{\text{Part 1}} \cdot \underbrace{\frac{\partial P_k(q_i|T)}{\partial p_j}}_{\text{Part 2}}. \quad (12)$$

where part 1 can be calculated by a ProbLog inference call computing (4). It does not depend on  $j$  and has to be calculated only once in every iteration of a gradient descent algorithm. Part 2 can be calculated as following

$$\frac{\partial P_k(q_i|T)}{\partial p_j} = \sum_{\substack{S \subseteq L_T \\ S \models q_i}} \delta_{jS} \prod_{\substack{c_x \in S \\ x \neq j}} p_x \prod_{\substack{c_x \in L_T \setminus S \\ x \neq j}} (1 - p_x), \quad (13)$$

where  $\delta_{jS} := 1$  if  $c_j \in S$  and  $\delta_{jS} := -1$  if  $c_j \in L_T \setminus S$ . It is derived by first deriving the gradient  $\partial P(S|T)/\partial p_j$  for a fixed subset  $S \subseteq L_T$  of facts, which is straight-forward, and then summing over all subsets  $S$  where  $q_i$  can be proven.

To ensure that all  $p_j$  stay probabilities during gradient descent, we reparameterize the search space and express each  $p_j \in ]0, 1[$  in terms of the sigmoid function  $p_j = \sigma(a_j) := 1/(1 + \exp(-a_j))$  applied to  $a_j \in \mathbb{R}$ . This technique has been used for Bayesian networks and in particular for sigmoid belief networks [35].

---

**Algorithm 5** Evaluating the gradient of a query efficiently by traversing the corresponding BDD, calculating partial sums, and adding only relevant ones.

---

```

function GRADIENT(BDD  $b$ , fact to derive for  $n_j$ )
   $(val, seen) = \text{GRADIENTEVAL}(\text{root}(b), n_j)$ 
  If  $seen = 1$  return  $val \cdot \sigma(a_j) \cdot (1 - \sigma(a_j))$ 
  Else return 0
function GRADIENTEVAL(node  $n$ , target node  $n_j$ )
  If  $n$  is the 1-terminal return  $(1, 0)$ 
  If  $n$  is the 0-terminal return  $(0, 0)$ 
  Let  $h$  and  $l$  be the high and low children of  $n$ 
   $(val(h), seen(h)) = \text{GRADIENTEVAL}(h, n_j)$ 
   $(val(l), seen(l)) = \text{GRADIENTEVAL}(l, n_j)$ 
  If  $n = n_j$  return  $(val(h) - val(l), 1)$ 
  ElseIf  $seen(h) = seen(l)$  return  $(\sigma(a_n) \cdot val(h) + (1 - \sigma(a_n)) \cdot val(l), seen(h))$ 
  ElseIf  $seen(h) = 1$  return  $(\sigma(a_n) \cdot val(h), 1)$ 
  ElseIf  $seen(l) = 1$  return  $((1 - \sigma(a_n)) \cdot val(l), 1)$ 

```

---

We derive the partial derivative  $\partial P_k(q_i|T)/\partial a_j$  in the same way as (13) but we have to apply the chain rule one more time due to the  $\sigma$  function

$$\sigma(a_j) \cdot (1 - \sigma(a_j)) \cdot \sum_{\substack{S \subseteq L_T \\ L \models q_i}} \delta_{jS} \prod_{\substack{c_x \in S \\ x \neq j}} \sigma(a_x) \prod_{\substack{c_x \in L_T \setminus S \\ x \neq j}} (1 - \sigma(a_x)).$$

We also have to replace every  $p_j$  by  $\sigma(p_j)$  when calculating the success probability. We employ the BDD-based algorithm to compute probabilities as outlined in Algorithm 1. In the following section we update this towards the gradient and introduce LeProbLog, the gradient descent algorithm for ProbLog.

As described in Section 3, BDDs can be used to efficiently calculate the success probability of a query, solving the disjoint-sum problem arising at summing over probabilities in an elegant way. Algorithm 1 can be modified straightforwardly such that it calculates the value of the gradient (13) of a success probability. Algorithm 5 shows the pseudocode. Both algorithms have a time and space complexity of  $O(\text{number of nodes in the BDD})$  when intermediate results are cached.

To see why this algorithm calculates the correct output let us first consider a full decision tree instead of a BDD. Each branch in the tree represents a product  $n_1 \cdot n_2 \cdot \dots \cdot n_i$ , where the  $n_i$  are the probabilities associated to the corresponding variable assignment of nodes on the branch. The gradient of such a branch  $b$  with respect to  $n_j$  is  $g_b = n_1 \cdot n_2 \cdot \dots \cdot n_{j-1} \cdot n_{j+1} \cdot \dots \cdot n_i$  if  $n_j$  is true, and  $-g_b$  if  $n_j$  is false in  $b$ . As all branches in a full decision tree are mutually exclusive, the gradient w.r.t.  $n_j$  can be obtained by simply summing the gradients of all branches ending in a leaf labeled 1. In BDDs however, isomorphic sub-parts are merged, and obsolete parts are left out. This implies that some paths from the root to the 1-terminal may not contain  $n_j$ , therefore having a gradient of 0. So, when calculating the gradient on the BDD, we have to keep track of whether  $n_j$

appeared on a path or not. Given that the variable order is the same on all paths, we can easily propagate this information in our bottom-up algorithm. This is exactly what is described in Algorithm 5. Specifically, `GRADIENTEVAL( $n, n_j$ )` calculates the gradient w.r.t.  $n_j$  in the sub-BDD rooted at  $n$ . It returns two values: the gradient on the sub-BDD and a Boolean indicating whether or not the target node  $n_j$  appears in the sub-BDD. When at some node  $n$  the indicator values for the two children differ, we know that  $n_j$  does not appear above the current node, and we can drop the partial result from the child with indicator 0. The indicator variable is also used on the top level: `GRADIENT` returns the value calculated by the bottom-up algorithm if  $n_j$  occurred in the BDD and 0 otherwise.

LeProbLog combines the BDD-based gradient calculation with a standard gradient descent search. Starting from parameters  $\mathbf{a} = a_1, \dots, a_n$  initialized randomly, the gradient  $\Delta\mathbf{a} = \Delta a_1, \dots, \Delta a_n$  is calculated, parameters are updated by subtracting the gradient, and updating is repeated until convergence. When using the  $k$ -probability with finite  $k$ , the set of  $k$  best proofs may change due to parameter updates. After each update, we therefore recompute the set of proofs and the corresponding BDD.

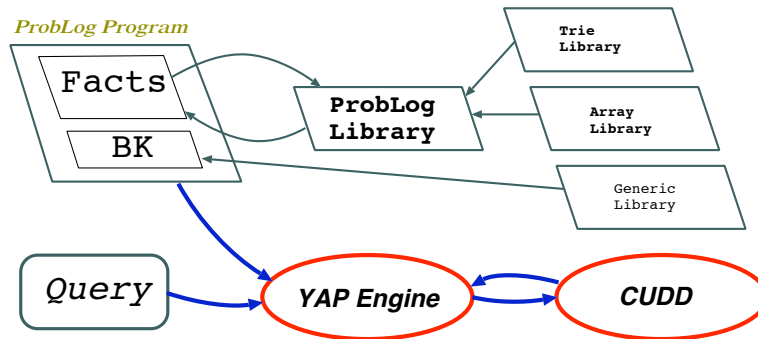
One nice side effect of the use of ProbLog is that it naturally combines *learning from entailment* and *learning from proofs*, two learning settings that so far have been considered separately. So far, we have assumed that the examples were ground facts together with their target probability. It turns out that the sketched technique also works when the examples are proofs, which correspond to conjunctions of probabilistic facts, and which can be seen as a conjunction of queries. Therefore, LeProbLog can use examples of both forms, (atomic) queries and proofs, at the same time.

## 8 Implementation Issues

Next, we discuss the main building blocks used to implement ProbLog on top of the YAP Prolog system [36], shown in Figure 3. On the top-left corner we show a typical ProbLog program, including ProbLog facts and background knowledge (BK).

In the current implementation, ProbLog programs are required to initially load the `problog` module. Each program consists of a set of labeled ground facts and of unlabeled *background knowledge*, a generic Prolog program. Labeled ground facts are preprocessed as described below. Unlabeled fragments of the program are generic Prolog programs. Notice that the implementation currently only supports labeled *ground facts*.

In contrast to standard Prolog queries, where one is interested in answer substitutions, in ProbLog one is interested in a probability. As discussed before, two common ProbLog queries are the most likely explanation and its probability, and the probability of whether a query would have a success substitution. We have discussed two very different approaches to the problem:



**Fig. 3.** ProbLog Implementation: A ProbLog program (top-left) requires the ProbLog library which in turn relies on functionality from the tries and array library. ProbLog queries (bottom-left) are sent to the YAP engine, and may require calling the BDD library.

- In  $k$  best and bounded approximation, the engine explicitly reasons about probabilities of proofs. The challenge is how to compute the probability of each individual proof, store a large number of proofs, and compute the probability of sets of proofs.
- In Monte Carlo, probabilities are used to sample from ProbLog programs. The challenge is how to compute a sample quickly, in a way that inference can be as efficient as possible.

ProbLog execution receives a ProbLog top-level query and proceeds as follows:

- Initialise a new ProbLog query;
- While probabilistic inference did not converge:
  - set environment for new query;
  - call Prolog query;
  - instrument every ProbLog call in the current proof: for example, a proof may be pruned immediately if its probability is lower than some bound;
  - process success or exit substitution;
- Call external solver, if required;

Notice that the current ProbLog implementation relies on Prolog’s backtracking to explore the search space. On the other hand, and in contrast to most other probabilistic logic implementations, in ProbLog there is no clear separation between logical and probabilistic inference: in a fashion similar to constraint logic programming, probabilistic inference can drive logical inference.

Implementing ProbLog poses a number of interesting challenges. First, labeled facts have to be efficiently compiled to allow mutual calls between the Prolog BK and the ProbLog engine. Second, for  $k$  best and bounded inference, sets of proofs have to be manipulated and transformed into BDDs. Finally, Monte Carlo simulation requires representing and manipulating samples. We discuss these issues next.

*Source-to-source transformation* We use the Prolog `term_expansion` mechanism to allow Prolog calls to labeled facts, and for labeled facts to call the ProbLog engine. As an example, the program:

```
0.715 :: drc('PubMed_2196878','MIM_609065').
0.659 :: drc('PubMed_8764571','HGNC_5014').
```

would be compiled as:

```
drc(A,B) :-
    problog_drc(C,A,B,D),
    add_to_proof(C,D).

problog_drc(0,'PubMed_2196878','MIM_609065',-0.3348).
problog_drc(1,'PubMed_8764571','HGNC_5014',-0.4166).
```

Thus, the internal representation of each fact contains an identifier, the original arguments, and the logarithm of the probability. The `add_to_proof` procedure receives the identifier of the ProbLog fact for the current proof and manipulates the proof according to the current query.

*Tries* Manipulating proofs is critical in ProbLog. We represent each proof as a list containing the identifier of each different ground probabilistic fact used in the proof, ordered by first use.

When manipulating proofs, the key operation is often *insertion*: we would like to add a proof to an existing set of proofs. Some algorithms, such as exact inference or Monte Carlo, only manipulate complete proofs. Others, such as bounded approximation, require adding partial derivations too. The nature of the SLD-tree means that proofs tend to share both a prefix and a suffix. Partial proofs tend to share prefixes only. This suggests using *tries* to maintain the set of proofs. We use the YAP implementation of tries for this task, based itself on XSB Prolog's work on tries of terms.

*Binary Decision Diagrams* To efficiently compute the probability of a DNF formula representing a set of proofs, our implementation represents this formula as a Binary Decision Diagram (BDD) [14]. Given a fixed variable ordering, a Boolean function  $f$  can be represented as a full Boolean decision tree, where each node on the  $i$ th level is labeled with the  $i$ th variable and has two children called low and high. Leaves are labeled by the outcome of  $f$  for the variable assignment corresponding to the path to the leaf, where in each node labeled  $x$ , the branch to the low (high) child is taken if variable  $x$  is assigned 0 (1). Starting from such a tree, one obtains a BDD by merging isomorphic subgraphs and deleting redundant nodes until no further reduction is possible. A node is redundant if the subgraphs rooted at its children are isomorphic. See Figure 1(b) for an example of a BDD.

Our implementation uses the C++ interface of the BDD package CUDD<sup>4</sup> to construct and evaluate BDDs. More precisely, a C++ program is created and

<sup>4</sup> <http://vlsi.colorado.edu/~fabio/CUDD>

then executed via Prolog’s shell utility, and results are communicated back via shared files. We currently work on a tighter integration of BDDs into Prolog.

When writing the program file, it is crucial to exploit the structure sharing already given by the trie representation of a DNF formula. CUDD builds BDDs by joining smaller BDDs using logical operations. The program therefore starts by creating a BDD for each variable. Afterwards, we traverse the trie bottom-up to successively build BDDs for all its subtrees. We use two types of operations. If we have BDDs for each child of an internal node, we construct a new BDD representing the disjunction of the children. In the next step, we build the conjunction of this new child BDD and the BDD representing the variable of the parent node. Subtrees occurring multiple times are only translated once, and the resulting BDD is used for all instances of the subtree.

As already outlined in Algorithm 1, the probability of a formula is then calculated by traversing the BDD, in each node summing the probability of the high and low child, weighted by the probability of the node’s variable being assigned true and false respectively. When intermediate results are cached, the algorithm has a time and space complexity linear in the number of nodes in the BDD.

*Monte Carlo* Monte Carlo execution is rather different from the approaches discussed so far. Instead of combining large numbers of proofs, we now need to be able to manipulate large numbers of different programs or samples.

One naive approach would be to generate a complete sample, and to check for a proof within the sample. Unfortunately, the approach does not scale to large databases, even if we try to reuse previous proofs: just generating a sample can be fairly expensive, as one would need to visit every ProbLog fact at every sample. In fact, in our experience, just representing and generating the whole sample can be a challenge for large databases. To address this first problem, we rely on YAP’s efficient implementation of arrays as the most compact way of representing large numbers of nodes. Moreover, we take advantage of the observation that often proofs are local, i.e. we only need to verify whether facts from a small fragment of the database are in the sample, to generate the sample *lazily*. In other words, we verify if a fact is in the sample only when we need it for a proof. Samples are thus represented as a three-valued array, originally initialised to 0, that means sampling was not asked yet; 1 means that the fact is in the sampled program, and 2 means not in sample.

## 9 Application

As an application of ProbLog, consider link mining in large networks of biological entities, such as genes, proteins, tissues, organisms, biological processes, and molecular functions. Life scientist utilize such data to identify and analyze relationships between entities, for instance between a protein and a disease.

Molecular biological data is available from public sources, such as Ensembl<sup>5</sup>, NCBI Entrez<sup>6</sup>, and many others. They contain information about various types of objects, such as the ones mentioned above, and many more. Information about known or predicted relationships between entities is also available, e.g., that gene A of organism B codes for protein C, which is expressed in tissue D, or that genes E and F are likely to be related since they co-occur often in scientific articles. Mining such data has been identified as an important and challenging task (cf. [37]).

A collection of interlinked heterogeneous biological data can be conveniently seen as a weighted graph or network of biological concepts, where the weight of an edge corresponds to the probability that the corresponding nodes are related [11]. A ProbLog representation of such a graph can simply consist of probabilistic `edge/2` facts, though finer grained representations using relations such as `codes/2`, `expresses/2` are also possible.

We have used the Biomine dataset [11] in our applications. It is an integrated index of a number of public biological databases, consisting of about 1 million objects and about 7 million relations. We next outline different ways of using ProbLog to query the Biomine dataset. We only assume probabilistic `edge/3` facts, where the third term indicates the edge type, and a simple background theory that contains the type of individual nodes as `node/2` facts and specifies an indirected (symmetric) `path/2` relation.

*Probabilistic inference* Assume a life scientist has hypothesized that ROBO1 gene is related to Alzheimer disease (AD). The probability that they are related is computed by ProbLog query `?- path('ROBO1', 'AD')`. The results is 0.70, indicating that—under all the assumptions made by ProbLog, Biomine and the source databases—they might be related. Assuming the life scientist has 100 candidate genes for Alzheimer disease, ProbLog can easily be used to rank the genes by their likelihood of being relevant for AD.

*Most likely explanation* Obviously, our life scientist would not be happy with the answer 0.70 alone. Knowing *what* the possible relation is is much more interesting, and could potentially lead to novel insight.

The best (most likely) proof obtained by ProbLog consists of edges `edge('ROBO1', 'SLIT1', interacts-with)`, `edge('SLIT1', 'GenomicContext:hsa10q23.3-q24', is-located-in)`, `edge('GenomicContext:hsa10q23.3-q24', 'GenomicContext:hsa10q24', contains)`, `edge('GenomicContext:hsa10q24', 'AD', is-related-to)`. In other words, ROBO1 interacts with SLIT1, which is located in a genomic area related to AD. This proof has probability 0.14.

Explanations obtained by probabilistic explanation based learning within ProbLog are on a more general level. By defining predicates related to node and edge types as operational, we obtain explanation `exp_path(A, B) ← node(A, gene), edge(A, C, interacts-with), node(C, gene), edge(C, D, is-located-in)`,

<sup>5</sup> [www.ensembl.org](http://www.ensembl.org)

<sup>6</sup> [www.ncbi.nlm.nih.gov/Entrez/](http://www.ncbi.nlm.nih.gov/Entrez/)

```

e_path(A,B) ← node(A, gene), edge(A,C, belongs_to), node(C, homologgroup),
              edge(B,C, refers_to), node(B, phenotype), nodes_distinct([B,C,A]).
e_path(A,B) ← node(A, gene), edge(A,C, codes_for), node(C, protein),
              edge(D,C, subsumes), node(D, protein), edge(D,E, interacts_with),
              node(E, protein), edge(B,E, refers_to), node(B, phenotype),
              nodes_distinct([B,E,D,C,A])
e_path(A,B) ← node(A, gene), edge(A,C, participates_in), node(C, pathway),
              edge(D,C, participates_in), node(D, gene), edge(D,E, codes_for),
              node(E, protein), edge(B,E, refers_to), node(B, phenotype),
              nodes_distinct([B,E,D,C,A])
e_path(A,B) ← node(A, gene), edge(A,C, is_found_in),
              node(C, cellularcomponent), edge(D,C, is_found_in), node(D, protein),
              edge(B,D, refers_to), node(B, phenotype), nodes_distinct([B,D,C,A])

```

**Fig. 4.** Some explanation clauses for  $\text{path}(A,B)$ , connecting gene  $A$  to phenotype  $B$ .

```

node(D, genomic-context), edge(D, E, contains), node(E, genomic-context),
edge(E, B, is-related-to), node(B, phenotype).

```

Figure 4 contains four explanations obtained for relationships between a gene (such as *ROBO1*) and a phenotype (such as *AD*). These explanations are all semantically meaningful. For instance, the first one indicates that gene  $A$  is related to phenotype  $B$  if  $A$  belongs to a group of homologous (i.e., evolutionarily related) genes that relate to  $B$ . The three other explanations are based on interaction of proteins: either an explicit one, by participation in the same pathway, or by being found in the same cellular component.

Such an explanation can then be used to query the database for a list of other genes connected to *AD* by the same type of pattern, and to rank them according to the probability of that connection, which may help the scientist to further examine the information obtained.

*Theory compression* The most likely explanation for  $\text{path}('ROBO1', 'AD')$  is just a single proof and does not capture alternative proofs, not to mention the whole network of related and potentially relevant objects. Theory compression can be used here to automatically extract a suitable subgraph for illustration. By definition, the extracted subgraph aims at maximizing the probability of  $\text{path}('ROBO1', 'AD')$ , i.e., it contains the most relevant nodes and edges.

Looking at a small graph of, say 12 nodes, helps to give an overview of the most relevant connections between *ROBO1* and *AD*. Such a look actually indicates that the association of *AD* to genomic context *hsa10q24* is possibly due to the *PLAU* gene, which is suspected to be associated with late-onset Alzheimer disease. The life scientist could now add  $\text{path}('ROBO1', 'Genomic-Context:hsa10q24')$  as a negative example, in order to remove connections using the genomic context from the extracted graph.

*Local pattern mining* Given a number of genes he considers relevant for the problem at hand, our life scientist could now be interested in relationships these

genes take part in with high probability. Local pattern mining offers a way to query ProbLog for such patterns or subgraphs of relationships without relying on predefined specific connections such as path.

*Parameter estimation* Imagine our life scientist got information on new entities and links between them, for example performing experiments or using information extraction techniques on a collection of texts. However, he does not know all the probabilities that should be attached to these new links, but only the probabilities of some of the links, of some specific paths, and of some pairs of entities being connected by some path. He could now use this knowledge as training examples for LeProbLog to automatically adjust the parameters of the new network to fit the available information.

## 10 Conclusions

In this chapter, we provided a survey of the developments around ProbLog, an extension of Prolog supporting both inductive and probabilistic querying. ProbLog has been motivated by the need to develop intelligent tools for supporting life scientists analyzing large biological networks involving uncertain data. The techniques presented here, i.e. probabilistic inference, similarity based reasoning, local pattern mining, theory compression and parameter estimation have all been evaluated in the context of such a biological network; we refer to [3, 12, 23, 31, 16] for details.

## Acknowledgements

We would like to thank our co-authors Kate Revoredo, Bart Demoen and Ricardo Rocha for their contributions to ProbLog. This work is partially supported by IQ (European Union Project IST-FET FP6-516169) and the GOA project 2008/08 Probabilistic Logic Learning. Angelika Kimmig and Bernd Gutmann are supported by the Research Foundation-Flanders (FWO-Vlaanderen).

## References

1. Suciu, D.: Probabilistic databases. *SIGACT News* **39**(2) (2008) 111–124
2. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. *Commun. ACM* **39**(11) (1996) 58–64
3. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In Veloso, M., ed.: *IJCAI*. (2007) 2462–2467
4. Dantsin, E.: Probabilistic logic programs and their semantics. In Voronkov, A., ed.: *Proc. 1st Russian Conf. on Logic Programming*. Volume 592 of LNCS. (1992)
5. Dalvi, N.N., Suciu, D.: Efficient query evaluation on probabilistic databases. In: *VLDB*. (2004) 864–875
6. Fuhr, N.: Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science* **51**(2) (2000) 95–110

7. Poole, D.: Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence* **64** (1993) 81–129
8. Poole, D.: Logic programming, abduction and probability. *New Generation Computing* **11** (1993) 377–400
9. Zelle, J., Mooney, R.: Inducing deterministic prolog parsers from treebanks: A machine learning approach. In: *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*. (1994) 748–753
10. Wrobel, S.: First order theory refinement. In De Raedt, L., ed.: *Advances in Inductive Logic Programming*. IOS Press, Amsterdam (1996) 14 – 33
11. Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., Toivonen, H.: Link discovery in graphs derived from biological databases. In: *DILS*. (2006) 35–49
12. Kimmig, A., De Raedt, L., Toivonen, H.: Probabilistic explanation based learning. In: *ECML*. (2007) 176–187
13. Sato, T.: A statistical learning method for logic programs with distribution semantics. In Sterling, L., ed.: *ICLP*, MIT Press (1995) 715–729
14. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers* **35**(8) (1986) 677–691
15. Lloyd, J.W.: *Foundations of Logic Programming*. 2. edn. Springer, Berlin (1989)
16. Gutmann, B., Kimmig, A., De Raedt, L., Kersting, K.: Parameter learning in probabilistic databases: A least squares approach. In Daelemans, W., Goethals, B., Morik, K., eds.: *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2008)*, Antwerp, Belgium, Springer (September 2008) 473–488
17. Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., De Raedt, L.: On the efficient execution of ProbLog programs. In: *ICLP*. (2008)
18. Mitchell, T.M., Keller, R.M., Kedar-Cabelli, S.T.: Explanation-based generalization: A unifying view. *Machine Learning* **1**(1) (1986) 47–80
19. DeJong, G., Mooney, R.J.: Explanation-based learning: An alternative view. *Machine Learning* **1**(2) (1986) 145–176
20. Hirsh, H.: Explanation-based generalization in a logic-programming environment. In: *IJCAI*. (1987) 221–227
21. van Harmelen, F., Bundy, A.: Explanation-based generalisation = partial evaluation. *Artif. Intell.* **36**(3) (1988) 401–412
22. Langley, P.: Unifying themes in empirical and explanation-based learning. In: *ML*. (1989) 2–4
23. Kimmig, A., De Raedt, L.: Probabilistic local pattern mining. In: *ILP*. (2008)
24. Dehaspe, L., Toivonen, H., King, R.D.: Finding frequent substructures in chemical compounds. In Agrawal, R., Stolorz, P., Piatetsky-Shapiro, G., eds.: *Proceedings of the 4th ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining*, AAAI Press (1998) 30–36
25. Tsur, S., Ullman, J.D., Abiteboul, S., Clifton, C., Motwani, R., Nestorov, S., Rosenthal, A.: Query flocks: A generalization of association-rule mining. In: *SIGMOD Conference*. (1998) 1–12
26. De Raedt, L., Ramon, J.: Condensed representations for inductive logic programming. In Dubois, D., Welty, C.A., Williams, M.A., eds.: *Proceedings of the 9th International Conference on Principles and Practice of Knowledge Representation*. AAAI Press (2004) 438–446
27. Esposito, F., Fanizzi, N., Ferilli, S., Semeraro, G.: Ideal refinement under object identity. In Langley, P., ed.: *Proceedings of the 17th International Conference on Machine Learning*, Morgan Kaufmann (August 2000) 263–270

28. Morishita, S., Sese, J.: Traversing itemset lattice with statistical metric pruning. In: Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM Press (2000) 226–236
29. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery* **1**(3) (1997) 241–258
30. Chui, C.K., Kao, B., Hung, E.: Mining frequent itemsets from uncertain data. In Zhou, Z.H., Li, H., Yang, Q., eds.: PAKDD. Volume 4426 of Lecture Notes in Computer Science., Springer (2007) 47–58
31. De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic prolog programs. *Machine Learning* **70**(2-3) (2008) 151–168
32. Gupta, R., Sarawagi, S.: Creating probabilistic databases from information extraction models. In: VLDB. (2006) 965–976
33. Chen, J., Muggleton, S., Santos, J.: Learning probabilistic logic models from probabilistic examples (extended abstract). In: ILP. (2007) 22–23
34. Mitchell, T.M.: *Machine Learning*. McGraw-Hill, New York (1997)
35. Saul, L., Jaakkola, T., Jordan, M.: Mean field theory for sigmoid belief networks. *JAIR* **4** (1996) 61–76
36. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: YAP User’s Manual. (2002) <http://www.ncc.up.pt/~vsc/Yap>.
37. Perez-Iratxeta, C., Bork, P., Andrade, M.: Association of genes to genetically inherited diseases using data mining. *Nature Genetics* **31** (2002) 316–319