Probabilistic Inductive Querying Using ProbLog

Luc De Raedt[1], Angelika Kimmig[1], Bernd Gutmann[1], Kristian Kersting[2], Vítor Santos Costa[3], and Hannu Toivonen[4]

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 one to represent and reason 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][1], 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

Department of Computer Science, Katholieke Universiteit Leuven, Belgium e-mail: {firstname.lastname}@cs.kuleuven.be · Fraunhofer IAIS, Sankt Augustin, Germany e-mail: kristian.kersting@iais.fraunhofer.de · Faculdade de Ciências, Universidade do Porto, Portugal e-mail: vsc@dcc.fc.up.pt · Department of Computer Science, University of Helsinki, Finland e-mail: hannu.toivonen@cs.helsinki.fi

[1] http://dtai.cs.kuleuven.be/problog/

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 supports.

A ProbLog program defines a probability 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, 5, 6, 7, 8]. The semantics of ProbLog is studied in Section 2. In Section 10, we discuss related work in statistical relational learning.

We now give a first overview of the types of queries ProbLog supports. Throughout the chapter, we use the graph in Figure 1(a) for illustration, inspired on the application in biological networks discussed in Section 9. 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. [9], as it returns the most likely cause for the query to succeed. This task will be discussed in Section 3.1.

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
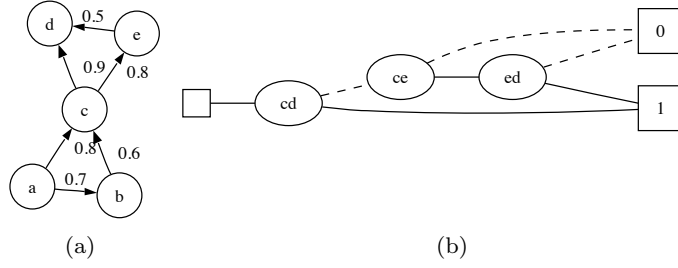
(a)   (b)

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 (cf. Sec. 4.3) 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 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 via Generalized Explanations
Which examples are most similar to a given example?
In explanation based learning the goal is to find a generalized explanation for a particular example in the light of a background theory. Within ProbLog, the traditional approach on explanation based learning is put into a new probabilistic perspective, as in a probabilistic background theory, choosing the most likely explanation provides a fundamental solution to the problem of multiple explanations, and furthermore, the found explanation can be used to retrieve and rank similar examples, that is, to reason by analogy. The most likely explanation thus 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. 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 that a direct connection is the most likely explanation, which can then be used to retrieve and rank other directly connected examples. This type of query is discussed in Section 5.

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 com-
monalities amongst the examples. Local pattern mining will be covered in
Section 6.

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 appli-
cation, 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 proba-
bilities to the negative examples. This task is analogous to a form of theory
revision [10, 11] 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. In the
example, assume 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 fewer edges) that
best matches these observations. Using a greedy approach, 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. Theory compression will be
discussed in Section 7.

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. We will discuss this type of query in Section 8.

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 [12]. 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 compu-
tational aspects and presents several algorithms (including approximation
and Monte Carlo algorithms) for computing probabilities of queries, while
the integration of ProbLog in the well-known implementation of YAP-Prolog
is discussed in Section 4. The following sections in turn consider each of the
inductive queries listed above. Finally, Section 9 provides a perspective on ap-

plying ProbLog on biological network mining, Section 10 positions ProbLog in the field of statistical relational learning, and Section 11 concludes.

2 ProbLog: Probabilistic Prolog

In this section, we present ProbLog and its semantics and then introduce two types of probabilistic queries: *probabilistic inference*, that is, computing the *success probability* of a query, and finding the *most likely explanation*, based on 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. To ensure a natural interpretation of these random variables, no two different facts c_i, c_j are allowed to unify, as otherwise, probabilities of ground facts would be higher than the individual probability given by different non-ground facts. The definite clauses allow the user to add arbitrary *background knowledge* (BK).[2] For ease of exposition, in the following we will assume all probabilistic facts to be ground.

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

$$0.8 :: \texttt{edge}(\texttt{a}, \texttt{c}). \qquad 0.7 :: \texttt{edge}(\texttt{a}, \texttt{b}). \qquad 0.8 :: \texttt{edge}(\texttt{c}, \texttt{e}).$$
$$0.6 :: \texttt{edge}(\texttt{b}, \texttt{c}). \qquad 0.9 :: \texttt{edge}(\texttt{c}, \texttt{d}). \qquad 0.5 :: \texttt{edge}(\texttt{e}, \texttt{d}).$$

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

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

We extend our example with the following background knowledge:

$$\texttt{path}(\texttt{X}, \texttt{Y}) :- \texttt{edge}(\texttt{X}, \texttt{Y}).$$
$$\texttt{path}(\texttt{X}, \texttt{Y}) :- \texttt{edge}(\texttt{X}, \texttt{Z}), \texttt{path}(\texttt{Z}, \texttt{Y}).$$

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

---

[2] While in early work on ProbLog [3] probabilities were attached to arbitrary definite clauses and all groundings of such a clause were treated as a single random event, we later on switched to a clear separation of logical and probabilistic part and random events corresponding to ground facts. This is often more natural and convenient, but can still be used to model the original type of clauses (by adding a corresponding probabilistic fact to the clause body) if desired.

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

$$P_s(q|T) = \sum_{L \subseteq L_T, \exists \theta : L \cup BK \models q\theta} P(L|T) . \tag{1}$$

In other words, the success probability of query q is the probability that the query q is *provable* in a randomly sampled logic program.

In our example, 40 of the 64 possible subprograms allow one to prove $path(c, d)$, namely all those that contain at least `edge(c,d)` (cd for short) or both `edge(c,e)` and `edge(e,d)`, so the success probability of that query is the sum of the probabilities of these programs: $P_s(path(c, d)|T) = P(\{ab, ac, bc, cd, ce, ed\}|T) + \ldots + P(\{cd\}|T) = 0.94$.

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

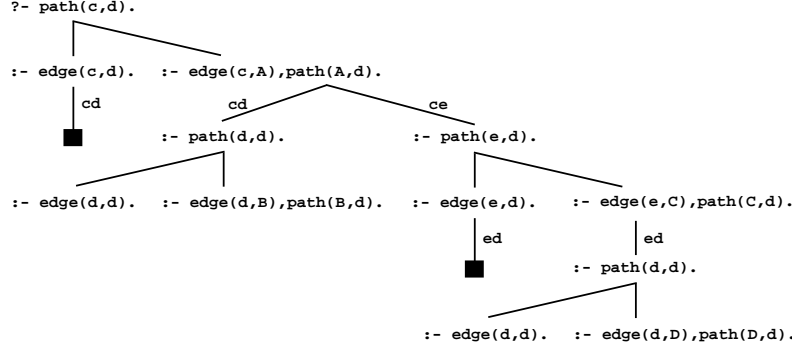$$P_x(q|T) = \max_{e \in E(q)} P(e|T) = \max_{e \in E(q)} \prod_{c_i \in e} p_i, \tag{2}$$

where $E(q)$ is the set of all explanations for query q [13].

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 an instance of the distribution semantics [14], where the basic distribution over ground facts is defined by treating each such fact as an independent random variable. 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 [14]. Similar instances of the distribution semantics have been used widely in the literature, e.g. [4, 5, 6, 7, 8]; see also Section 10.

3 Probabilistic Inference

In this section, we present various algorithms and techniques for performing probabilistic inference in ProbLog, that is computing the success probabilities and most likely explanations of queries. We will discuss the implementation of these methods in Section 4.

```
?- path(c,d).


:- edge(c,d).    :- edge(c,A),path(A,d).
        |  cd              cd              ce
        ■         :- path(d,d).      :- path(e,d).


  :- edge(d,d).   :- edge(d,B),path(B,d).  :- edge(e,d).    :- edge(e,C),path(C,d).
                                                   |  ed            |  ed
                                                   ■        :- path(d,d).


                                          :- edge(d,d).   :- edge(d,D),path(D,d).
```

Fig. 2 SLD-tree for query path(c,d).

3.1 Exact Inference

As computing the *success probability* of a query using Equation (1) directly
is infeasible for all but the tiniest programs, ProbLog uses a method involv-
ing two steps [3]. The first step computes the proofs of the query q in the
logical part of the theory T, that is, in $L_T \cup BK$. The result will be a DNF
formula. The second step employs Binary Decision Diagrams [15] to com-
pute the probability of this formula. Comparable first steps are performed
in pD [6], PRISM [8] and ICL [16], however, as we will see below, these sys-
tems differ in the method used to tackle the second step. Let us now explain
ProbLog's two steps in more detail.

The first step employs SLD-resolution [17], 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
$\{p_1 :: d_1, \cdots, p_k :: d_k\} \subseteq T$. 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 $\{p_{i_1} ::$
$d_{i_1}, \cdots, p_{i_k} :: d_{i_k}\} \subseteq T$ is then the probability of the conjunctive formula
$b_{i_1} \wedge \cdots \wedge b_{i_k}$. 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

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

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 $path(c,d)$ is $cd \vee (ce \wedge ed)$, where we use xy as Boolean variable representing $edge(x,y)$.

Computing the probability of DNF formulae is an #P-hard problem [18], 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, which is reported to scale to about ten proofs. PRISM [8] and PHA [7] avoid the disjoint-sum-problem by requiring proofs to be mutually exclusive, while ICL uses a symbolic disjoining technique with limited scalability [16]. As the type of application considered here often requires dealing with hundreds or thousands of proofs, the second step of our implementation employs Binary Decision Diagrams (BDDs) [15], an efficient graphical representation of a Boolean function over a set of variables which scales to tens of thousands of proofs; we will discuss the details in Section 4.3. Nevertheless, calculating the probability of a DNF formula remains a hard problem and can thus 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 for the success probability. We will come back to these methods from Section 3.2 onwards.

Compared to probabilistic inference, computing the most likely explanation is much easier. Indeed, calculating the *explanation probability P_x* corresponds to computing the probability of a conjunctive formula only, so that the disjoint-sum-problem does not arise. While one could imagine to use Viterbi-like dynamic programming techniques on the DNF to calculate the explanation probability, our approach avoids constructing the DNF – which requires examining a potentially high number of low-probability proofs – by using a best-first search, guided by the probability of the current partial proof. In terms of logic programming [17], the algorithm does not completely traverse the entire SLD-tree to find all proofs, but instead uses iterative deepening with a probability threshold α to find the most likely one. Algorithm 1 provides the details of this procedure, where *stop* is a minimum threshold to avoid exploring infinite SLD-trees without solution and *resolutionStep* performs the next possible resolution step on the goal and updates the probability p of the current derivation and its explanation *expl* accordingly; backtracking reverts these steps to explore alternative steps while at the same time keeping the current best solution $(max, best)$ and the current threshold α.

Algorithm 1 Calculating the most likely explanation by iterative deepening search in the SLD-tree.

---

   function BESTPROBABILITY(query q)
     $\alpha := 0.5$; $max = -1$; $best := false$; $expl := \emptyset$; $p = 1$; $goal = q$;
   while $\alpha > stop$ **do**
     repeat
       $(goal, p, expl) := resolutionStep(goal, p, expl)$
       if $p < \alpha$ **then**
         backtrack resolution
       end if
       if $goal = \emptyset$ **then**
         $max := p$; $best := expl$; $\alpha := p$; backtrack resolution
       end if
     until no further backtracking possible
     if $max > -1$ **then**
       return $(max, best)$
     else
       $\alpha := 0.5 \cdot \alpha$
     end if
   end while
   end function

3.2 Bounded Approximation

The first approximation algorithm for obtaining success probabilities, 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 [9] 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[3], 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 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 $d = cd \vee (ce \wedge ed)$.

---

[3] 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.

3.3 K-Best

Using a fixed number of proofs to approximate the success 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 8. [19] 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):

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

where $E_k(q) = \{e \in E(q) | P_x(e) \geq P_x(e_k)\}$ with e_k the kth 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 extending Algorithm 1; 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 $k \geq 4$.

3.4 Monte Carlo

As an alternative approximation technique without BDDs, [20] 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 δ. 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 [12].
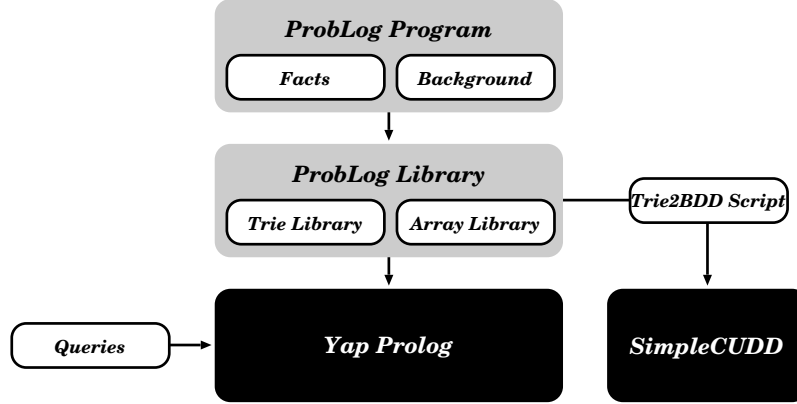
Fig. 3 ProbLog Implementation: A ProbLog program (top) requires the ProbLog library which in turn relies on functionality from the tries and array libraries. ProbLog queries (bottom-left) are sent to the YAP engine, and may require calling the BDD library CUDD via SimpleCUDD.

4 Implementation

This section discusses the main building blocks used to implement ProbLog on top of the YAP-Prolog system [21] as introduced in [20]. An overview is shown in Figure 3, with a typical ProbLog program, including ProbLog facts and background knowledge (BK), at the top.

The implementation requires ProbLog programs to use the `problog` module. Each program consists of a set of labeled facts and of unlabeled *background knowledge*, a generic Prolog program. Labeled facts are preprocessed as described below. Notice that the implementation requires all queries to non-ground probabilistic facts to be ground on calling.

In contrast to standard Prolog queries, where one is interested in answer substitutions, in ProbLog one is primarily interested in a probability. As discussed before, two common ProbLog queries ask for the most likely explanation and its probability, and the probability of whether a query would have an answer substitution. In Section 3, we have discussed two very different approaches to the problem:

- In exact inference (Section 3.1), k-best (Section 3.3) and bounded approximation (Section 3.2), 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 (Section 3.4), the probabilities of facts 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 programs execute from a top-level query and are driven through a ProbLog query. The inference algorithms discussed in Section 3 can be abstracted as follows:

- Initialize the inference algorithm;
- While probabilistic inference did not converge:

  – initialize a new query;
  – execute the query, instrumenting every ProbLog call in the current proof. Instrumentation is required for recording the ProbLog facts required by a proof, but may also be used by the inference algorithm to stop proofs (e.g., if the current probability is lower than a bound);
  – process success or exit substitution;

- Proceed to the next step of the algorithm: this may be trivial or may require calling an external solver, such as a BDD tool, to compute a probability.

Notice that the current ProbLog implementation relies on the Prolog engine to efficiently execute goals. On the other hand, and in contrast to most other probabilistic language 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.

From a Prolog implementation perspective, ProbLog poses a number of interesting challenges. First, labeled facts have to be efficiently compiled to allow mutual calls between the Prolog program and the ProbLog engine. Second, for exact inference, k-best and bounded approximation, 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.

4.1 Source-to-source transformation

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

$$0.715 :: \texttt{edge}('\texttt{PubMed\_2196878}','\texttt{MIM\_609065}').$$
$$0.659 :: \texttt{edge}('\texttt{PubMed\_8764571}','\texttt{HGNC\_5014}'). \tag{5}$$

would be compiled as:

$$\begin{aligned}
\texttt{edge(A,B)} : - \; &\texttt{problog\_edge(ID,A,B,LogProb)}, \\
&\texttt{grounding\_id(edge(A,B),ID,GroundID)}, \\
&\texttt{add\_to\_proof(GroundID,LogProb)}.
\end{aligned} \tag{6}$$

$$\texttt{problog\_edge(0,'PubMed\_2196878','MIM\_609065',-0.3348)}.$$
$$\texttt{problog\_edge(1,'PubMed\_8764571','HGNC\_5014',-0.4166)}.$$

Thus, the internal representation of each fact contains an identifier, the original arguments, and the logarithm of the probability[4]. The `grounding_id` procedure will create and store a grounding specific identifier for each new grounding of a non-ground probabilistic fact encountered during proving, and retrieve it on repeated use. For ground probabilistic facts, it simply returns the identifier itself. The `add_to_proof` procedure updates the data structure representing the current path through the search space, i.e., a queue of identifiers ordered by first use, together with its probability.

4.2 Tries

Manipulating proofs is critical in ProbLog. We represent each proof as a queue containing the identifier of each different ground probabilistic fact used in the proof, ordered by first use. The implementation requires calls to non-ground probabilistic facts to be ground, and during proving maintains a table of groundings used within the current query together with their identifiers. In our implementation, the queue is stored in a backtrackable global variable, which is updated by calling `add_to_proof` with an identifier for the current ProbLog fact. We thus exploit Prolog's backtracking mechanism to avoid recomputation of shared proof prefixes when exploring the space of proofs. Storing a proof is simply a question of adding the value of the variable to a store.

Storing and manipulating proofs is critical in ProbLog. 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* [22] 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 [23].

---

[4] We use the logarithm to avoid numerical problems when calculating the probability of a derivation, which is used to drive inference.

Algorithm 2 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
     $prob(h) :=$PROBABILITY(h)
     $prob(l) :=$PROBABILITY(l)
     return $p_n \cdot prob(h) + (1 - p_n) \cdot prob(l)$
   end function

4.3 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) [15]. Given a fixed variable ordering, a Boolean function f can be represented as a full Boolean decision tree, where each node on the ith level is labeled with the ith 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.

Figure 1(b) shows the BDD corresponding to $cd \vee (ce \wedge ed)$, the formula of the example query $path(c, d)$. Given a BDD, it is easy to compute the probability of the corresponding 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 2. In practice, memorization of intermediate results is used to avoid the recomputation at nodes that are shared between multiple paths, resulting in a time and space complexity linear in the number of nodes in the BDD.

We use SimpleCUDD [24][5] as a wrapper tool for the BDD package CUDD[6] to construct and evaluate BDDs. More precisely, the trie representation of the DNF is translated to a BDD generation script, which is processed by SimpleCUDD to build the BDD using CUDD primitives. It is executed via Prolog's shell utility, and results are reported via shared files.

During the generation of the code, it is crucial to exploit the structure sharing (prefixes and suffixes) already in the trie representation of a DNF formula, otherwise CUDD computation time becomes extremely long or memory overflows quickly. Since CUDD builds BDDs by joining smaller BDDs using logical operations, the trie is traversed bottom-up to successively generate

---

[5]   `http://people.cs.kuleuven.be/~theofrastos.mantadelis/tools/`
`simplecudd.html`

[6] `http://vlsi.colorado.edu/~fabio/CUDD`

code for all its subtrees. Two types of operations are used to combine nodes. The first creates conjunctions of leaf nodes and their parent if the leaf is a single child, the second creates disjunctions of all child nodes of a node if these child nodes are all leaves. In both cases, a subtree that occurs multiple times in the trie is translated only once, and the resulting BDD is used for all occurrences of that subtree. Because of the optimizations in CUDD, the resulting BDD can have a very different structure than the trie.

4.4 Monte Carlo

Monte Carlo execution is quite different from the approaches discussed before, as the two main steps are **(a)** generating a sample program and **(b)** performing standard refutation on the sample. Thus, instead of combining large numbers of proofs, we need 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 initialized 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. Note that as fact identifiers are used to access the array, the approach cannot directly be used for non-ground facts, whose identifiers are generated on demand. The current implementation of Monte Carlo therefore uses the internal database to store the result of sampling different groundings of such facts.

The tight integration of ProbLog's probabilistic inference algorithms in the state-of-the-art YAP-Prolog system discussed here includes several improvements over the initial implementation used in [3], thereby enabling the use of ProbLog to effectively query Sevon's Biomine network [12] containing about 1,000,000 nodes and 6,000,000 edges. For experimental results obtained using the various methods in the context of this network as well as for further implementation details, we refer to [25].

5 Probabilistic Explanation Based Learning

In this section, we address the question of finding examples that are similar or analogous to a given example. To this end, we combine two types of queries, namely finding the *most likely (generalized) explanation* for an example and *reasoning by analogy*, which is the process of finding (and possibly ranking) examples with a similar explanation. ProbLog's probabilistic explanation based learning technique (PEBL) [13] employs a background theory that allows to compute a most likely explanation for the example and to generalize that explanation. It thus extends the concept of explanation based learning (EBL) to a probabilistic framework. Probabilistic explanation based learning as introduced here is also related to probabilistic abduction, as studied by Poole [7]. The difference with Poole's work however is that we follow the deductive view of EBL to compute *generalized* explanations and also apply them for analogical reasoning.

The central idea of explanation based learning [26, 27] 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 θ such that $head(c)\theta = e$ and $body(c)\theta$ is the most likely explanation for e given T.

Following the work by [28, 29], 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 $p(t_1, ..., t_n)$. 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 variabelized goal, i.e. $p(X_1, ..., X_n)$ where the X_i are different variables. The only difference is that in the general proof atoms $q(s_1, ..., s_r)$ for operational predicates q in a goal $? - g_1, ..., g_i, q(s_1, ..., s_r), g_{i+1}, ..., g_n$ 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 [28, 29].

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)`. To be able to identify the examples covered by such an explanation, we represent it as so-called *explanation clause*, where the generalized explanation forms the body and the predicate in the head is renamed to distinguish the clause from those for the original predicate. In our example, we thus get the explanation 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)`.

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). It thus returns the first explanation clause in our example.

As we have explained in Section 3.1, 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 PEBL algorithm thus modifies Algorithm 1 to return the generalized explanation based on the most likely proof, which, as in standard EBL, is generated using the same sequence of resolution steps on the variabelized goal. As for the k-probability (Section 3.3), a variant of the algorithm can be used to return the k most probable structurally distinct explanations.

The probabilistic view on explanation based learning adopted in ProbLog offers natural solutions to two issues traditionally discussed in the context of explanation based learning [26, 30]. 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 $e_1, ..., e_n$. Including the variabelized 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 `path(c,d)` and `path(b,e)`. We

thus have to simultaneously prove both examples and the variabelized goal
`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)`.

At the beginning of this section, we posed the question of finding ex-
amples that are similar or analogous to a given example. The explanation
clause constructed by PEBL provides a concrete measure for analogy or sim-
ilarity based reasoning: examples are considered similar if they can be ex-
plained using the general pattern that best explains the given example, that
is, if they can be proven using the explanation clause. In our example, us-
ing the clause `exp_path(X,Y)← edge(X,Y)` obtained from `path(c,d)`,
five additional instances of `exp_path(X,Y)` can be proven, corresponding
to the other edges of the graph. Furthermore, such similar examples can
naturally be ranked according to their probability, that is, in our example,
`exp_path(a,c)` and `exp_path(c,e)` would be considered most similar to
`path(c,d)`, as they have the highest probability.

We refer to [13] for more details as well as experiments in the context of
biological networks.

6 Local Pattern Mining

In this section, we address the question of finding queries that are likely to
succeed on a given set of examples. 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 – possibly including negative
ones – rather than a single one, and computes a set of patterns (or clauses)
satisfying certain conditions. As in probabilistic explanation based learning,
the discovered patterns can be used to retrieve and rank further examples,
again realizing a kind of similarity based reasoning or reasoning by analogy.

Our approach to probabilistic local pattern mining [31] builds upon multi-
relational query mining techniques [32], extending them towards probabilistic
databases. We use ProbLog to represent databases and queries, abbreviating
vectors of variables as $\mathbf{X}$. We assume a designated relation *key* containing
the set of tuples to be characterized using queries, and restrict the language
$\mathcal{L}$ of patterns to the set of conjunctive queries $r(\mathbf{X})$ defined as

$$r(\mathbf{X}) : -key(\mathbf{X}), l_1, ..., l_n \tag{7}$$

where the l_i are positive atoms. Additional syntactic or semantic restrictions, called *bias*, can be imposed on the form of queries by explicitly specifying the language $\mathcal{L}$, cf. [33, 34, 32]. *Query Mining* aims at finding all queries satisfying a selection predicate ϕ. It can be formulated as follows, cf. [32, 34]:

Given   a language $\mathcal{L}$ containing queries of the form (7), a database $\mathcal{D}$ including the designated relation key, and a selection predicate ϕ

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

The most prominent selection predicate is minimum frequency, an anti-monotonic predicate, requiring a minimum number of tuples covered. Anti-monotonicity is based on a generality relation between patterns. We employ OI-subsumption [35], as the corresponding notion of subgraph isomorphism is favorable within the intended application in network mining.

   Correlated Pattern Mining [36] uses both *positive* and *negative* examples, given as two designated relations key^+ and key^- of the same arity, to find the top k patterns, that is, the k patterns scoring best w.r.t. a function ψ. The function ψ employed is convex, e.g. measuring a statistical significance criterion such as χ^2, cf. [36], and measures the degree to which the pattern is statistically significant or unexpected. Thus correlated pattern mining corresponds to the setting

$$\phi(q, \mathcal{D}) = q \in \arg_k \max_{q \in \mathcal{L}} \psi(q, \mathcal{D}) \ . \tag{8}$$

Consider the database corresponding to the graph in Figure 1(a) (ignoring probability labels) with $key^+ = \{a, c\}$ and $key^- = \{d, e\}$. A simple correlation function is $\psi(q, \mathcal{D}) = \mathsf{COUNT}(\mathsf{q}^+(*)) - \mathsf{COUNT}(\mathsf{q}^-(*))$, where $\mathsf{COUNT}(\mathsf{q}(*))$ is the number of different provable ground instances of q and q^x denotes query q restricted to key^x. We obtain $\psi(Q1, \mathcal{D}) = 2 - 0 = 2$ and $\psi(Q2, \mathcal{D}) = 1 - 1 = 0$ for queries

$(Q1)\ \ q(X) : -key(X), edge(X, Y), edge(Y, Z).$
$(Q2)\ \ q(X) : -key(X), edge(X, d).$

   Multi-relational query miners such as [32, 34] often follow a level-wise approach for frequent query mining [37], where at each level new candidate queries are generated from the frequent queries found on the previous level. In contrast to Apriori, instead of a "joining" operation, they employ a refinement operator ρ to compute more specific queries, and also manage a set of infrequent queries to take into account the specific language requirements imposed by $\mathcal{L}$. To search for all solutions, it is essential that the refinement operator is optimal w.r.t. $\mathcal{L}$, i.e. ensures that there is exactly one path from the most general query to every query in the search space. This can be achieved

by restricting the refinement operator to generate queries in a canonical form, cf. [34].

Morishita and Sese [36] adapt Apriori for finding the top k patterns w.r.t. a boundable function ψ, i.e. for the case where there exists a function u (different from a global maximum) such that $\forall g, s \in \mathcal{L} : g \preceq s \rightarrow \psi(s) \leq u(g)$. Again, at each level candidate queries are obtained from those queries generated at the previous level that qualify for refinement, which now means they either belong to the current k best queries, or are still promising as their upper-bound is higher than the value of the current k-th best query. The function $\psi(q, \mathcal{D}) = \mathsf{COUNT}(\mathsf{q}^+(*)) - \mathsf{COUNT}(\mathsf{q}^-(*))$ used in the example above is upper-boundable using $u(q, \mathcal{D}) = \mathsf{COUNT}(\mathsf{q}^+(*))$. For any $g \preceq s$, $\psi(s) \leq \mathsf{COUNT}(\mathsf{s}^+(*)) \leq \mathsf{COUNT}(\mathsf{g}^+(*))$, as $\mathsf{COUNT}(\mathsf{s}^-(*)) \geq 0$ and COUNT is anti-monotonic. To illustrate this, assume we mine for the 3 best correlated queries in our graph database. Table 1 shows counts on key^+ and key^- and ψ-values obtained during the first level of mining. The highest score achieved is 1. Queries 1, 3, 4, 6, 8, 9 are the current best queries and will thus be refined on the next level. Queries 5 and 7 have lower scores, but upper bound $c^+ = 1$, implying that their refinements may still belong to the best queries and have to be considered on the next level as well. The remaining queries are pruned, as they all have an upper bound $c^+ = 0 < 1$, i.e. all their refinements are already known to score lower than the current best queries.

	query	c^+	c^-	ψ
1	key(X),edge(X,Y)	**2**	1	**1**
2	key(X),edge(X,a)	0	0	0
3	key(X),edge(X,b)	**1**	0	**1**
4	key(X),edge(X,c)	**1**	0	**1**
5	key(X),edge(X,d)	**1**	1	0
6	key(X),edge(X,e)	**1**	0	**1**
7	key(X),edge(Y,X)	**1**	2	- 1
8	key(X),edge(a,X)	**1**	0	**1**
9	key(X),edge(b,X)	**1**	0	**1**
10	key(X),edge(c,X)	0	2	-2
11	key(X),edge(d,X)	0	0	0
12	key(X),edge(e,X)	0	1	-1

Table 1 Counts on key^+ and key^- and ψ-values obtained during the first level of mining in the graph of Figure 1(a). The current minimal score for best queries is 1, i.e. only queries with $\psi \geq 1$ or $c^+ \geq 1$ will be refined on the next level.

The framework for query mining as outlined above can directly be adapted towards *probabilistic* databases. The key changes involved are 1) that the database $\mathcal{D}$ is *probabilistic*, and 2) that the selection predicate ϕ or the correlation measure ψ is based on the probabilities of queries. In other words, we employ a probabilistic membership function. In non-probabilistic frequent query mining, every tuple in the relation key either satisfies the query or not. So, for a conjunctive query q and a 0-1 membership function $M(t|q, \mathcal{D})$, we

can explicitly write the counting function underlying frequency as a sum:

$$freq(q, \mathcal{D}) = \sum_{t \in key} M(t|q, \mathcal{D})$$

On a more general level, this type of function can be seen as *aggregate* of the membership function $M(t|q, \mathcal{D})$.

To apply the algorithms sketched above with a probabilistic database $\mathcal{D}$, it suffices to replace the deterministic membership function $M(t|q, \mathcal{D})$ with a probabilistic variant. Possible choices for such a probabilistic membership function $P(t|q, \mathcal{D})$ include the success probability $P_s(q(t)|\mathcal{D})$ or the explanation probability $P_x(q(t)|\mathcal{D})$ as introduced for ProbLog in Equations (1) and (2). Note that using such query probabilities as probabilistic membership function is anti-monotonic, that is, if $q_1 \preceq q_2$ then $P(t|q_1, \mathcal{D}) \geq P(t|q_2, \mathcal{D})$. Again, a natural choice of selection predicate ϕ is the combination of a minimum threshold with an aggregated probabilistic membership function:

$$agg(q, \mathcal{D}) = \mathbf{AGG}_{t \in key} \ P(t|q, \mathcal{D}). \tag{9}$$

Here, $\mathbf{AGG}$ denotes an aggregate function such as $\sum$, min, max or $\prod$, which is to be taken over all tuples t in the relation *key*. Choosing $\sum$ with a deterministic membership relation corresponds to the traditional frequency function, whereas $\prod$ computes a kind of *likelihood* of the data. Note that whenever the membership function P is anti-monotone, selection predicates of the form $agg(q, \mathcal{D}) > c$ (with $agg \in \{\sum, \min, \max, \prod\}$) are anti-monotonic with regard to OI-subsumption, which is crucial to enable pruning.

When working with both positive and negative examples, the main focus lies on finding queries with a high aggregated score on the positives and a low aggregated score on the negatives. Note that using unclassified instances *key* corresponds to the special case where $key^+ = key$ and $key^- = \emptyset$. In the following, we will therefore consider instances of the selection function (9) for the case of classified examples key^+ and key^- only. Choosing sum as aggregation function results in a *probabilistic frequency pf* (10) also employed by [38] in the context of item-set mining, whereas product defines a kind of *likelihood LL* (11). Notice that using the product in combination with a non-zero threshold implies that *all* positive examples must be covered with non-zero probability. We therefore introduce a softened version LL_n (12) of the likelihood, where $n < |key^+|$ examples have to be covered with non-zero probability. This is achieved by restricting the set of tuples in the product to the n highest scoring tuples in key^+, thus integrating a deterministic (anti-monotonic) selection predicate into the probabilistic one. More formally, the three functions used are defined as follows:

$$pf(q, \mathcal{D}) = \sum_{t \in key^+} P(t|q, \mathcal{D}) - \sum_{t \in key^-} P(t|q, \mathcal{D}) \qquad (10)$$

$$LL(q, \mathcal{D}) = \prod_{t \in key^+} P(t|q, \mathcal{D}) \cdot \prod_{t \in key^-} (1 - P(t|q, \mathcal{D})) \qquad (11)$$

$$LL_n(q, \mathcal{D}) = \prod_{t \in key_n^+} P(t|q, \mathcal{D}) \cdot \prod_{t \in key^-} (1 - P(t|q, \mathcal{D})) \qquad (12)$$

Here, key_n^+ contains the n highest scoring tuples in key^+. In correlated query mining, we obtain an upper bound on each of these functions by omitting the scores of negative examples, i.e. the aggregation over key^-.

Consider again our graph database, now with probabilities. Using P_x as probabilistic membership function, the query $q(X) : -key(X), edge(X, Y)$ gets probabilistic frequency $pf(q, \mathcal{D}) = P_x(a|q, \mathcal{D}) + P_x(c|q, \mathcal{D}) - (P_x(d|q, \mathcal{D}) + P_x(e|q, \mathcal{D})) = 0.8 + 0.9 - (0 + 0.5) = 1.2$ (with upper bound $0.8 + 0.9 = 1.7$), likelihood $LL(q, \mathcal{D}) = 0.8 \cdot 0.9 \cdot (1 - 0) \cdot (1 - 0.5) = 0.36$ (with upper bound $0.8 \cdot 0.9 = 0.72$), and softened likelihood $LL_1(q, \mathcal{D}) = 0.9 \cdot (1 - 0) \cdot (1 - 0.5) = 0.9$ (with upper bound 0.9).

For further details and experiments in the context of the biological network of Section 9, we refer to [31].

7 Theory Compression

In this section, we investigate how to obtain a small compressed probabilistic 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 networks of uncertain relationships between biological entities as it allows them to identify the most relevant components of the theory.

The technique on which we build is that of theory compression [39], where the goal is to remove as many edges, i.e., probabilistic 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 fewer edges) that best matches these observations. Using a greedy approach, 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 [10, 11] 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 at most 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 analyze 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

$$\mathcal{L}(E|T) = \prod_{e \in P} P(e|T) \cdot \prod_{e \in N} (1 - P(e|T)) \qquad (13)$$

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[7]. 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. (13) 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

---

[7] Note that this is slightly different from specifying a distribution over (positive) examples.

Algorithm 3 ProbLog theory compression

---

   function Compress($S = \{p_1 :: c_1, \ldots, p_n :: c_n\}$, E, k, ϵ)

     for $e \in E$ **do**

       Call Approximate(e, S, δ) 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 $DNF(low, e)$

     end for

     $R := \{p_i :: c_i \mid b_i$ (indicator for fact i) occurs in a $DNF(low, e)\}$

     $BDD(E) := \bigcup_{e \in E} \{BDD(e)\}$

     improves := true

     while ($|R| > k$ or improves) and $R \neq \emptyset$ **do**

       $ll :=$ Likelihood($R, BDD(E), \epsilon$)

       $i := \arg\max_{i \in R}$ Likelihood($R - \{i\}, BDD(E), \epsilon$)

       improves := ($ll \leq$ Likelihood($R - \{i\}, BDD(E), \epsilon$))

       if improves or $|R| > k$ **then**

         $R := R - \{i\}$

       end if

     end while

     Return R

   end function

---

these problems, we slightly redefine $P(e|T)$ in Eq. (13) as

$$\hat{P}(e|T) = \max\big(\min[1 - \epsilon, P(e|T)], \epsilon\big) \qquad (14)$$

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 3. 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 in Section 3.2, which computes the DNFs and BDDs for lower and upper bounds (for-loop). 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 (while-loop). 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 [39], 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.

Theory compression as introduced here bears some relationships to the PTR approach by [40], where weights or probabilities are used as a kind of bias during the process of revising a logical theory. ProbLog compression is also somewhat related to Zelle and Mooney's work on Chill [41] in that it specializes an overly general theory but differs again in the use of a probabilistic framework. In the context of probabilistic logic languages, PFORTE [42] is a theory revision system using BLPs [43] that follows a hill-climbing approach similar to the one used here, but with a wider choice of revision operators.

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 [39].

8 Parameter Estimation

In this section, we address the question of 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 $0.6 : locatedIn(a, b)$ and $0.7 : interacting(a, c)$ as used by Gupta and Sarawagi [44] and Chen *et al.* [45]. The parameter estimation technique should then determine the best values for the parameters. Our approach as implemented in LeProbLog [19, 46] (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 regression task 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:

$$MSE(T) = \frac{1}{M} \sum_{1 \leq i \leq M} \left(P_k(q_i|T) - \tilde{p}_i \right)^2 . \tag{15}$$

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 η, and subtracted from the current parameters. To get the gradient of the MSE, we apply the sum and chain rule to Eq. (15). This yields the partial derivative

$$\frac{\partial MSE(T)}{\partial p_j} = \frac{2}{M} \sum_{1 \leq i \leq M} \underbrace{\left(P_k(q_i|T) - \tilde{p}_i \right)}_{\text{Part 1}} \cdot \underbrace{\frac{\partial \, P_k(q_i|T)}{\partial p_j}}_{\text{Part 2}} . \tag{16}$$

Algorithm 4 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) = $ GRADIENTEVAL($root(b), n_j$)
  If $seen = 1$ return $val \cdot \sigma(a_j) \cdot (1 - \sigma(a_j))$
  Else return 0
end function
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)) = $ GRADIENTEVAL(h, n_j)
  $(val(l), seen(l)) = $ 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 $((1 - \sigma(a_n)) \cdot val(l), 1)$
end function

---

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

$$\frac{\partial P_k(q_i|T)}{\partial p_j} = \sum_{\substack{S \subseteq L_T \\ S \models q_i}} \delta_{jS} \prod_{\substack{c_x \in S \\ x \neq j}} p_x \prod_{\substack{c_x \in L_T \setminus S \\ x \neq j}} (1 - p_x) \,, \tag{17}$$

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 straightforward, 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 [47]. We derive the partial derivative $\partial P_k(q_i|T)/\partial a_j$ in the same way as (17) but we have to apply the chain rule one more time due to the σ 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 2. In the following, we update this towards the gradient and introduce LeProbLog, the gradient descent algorithm for ProbLog.
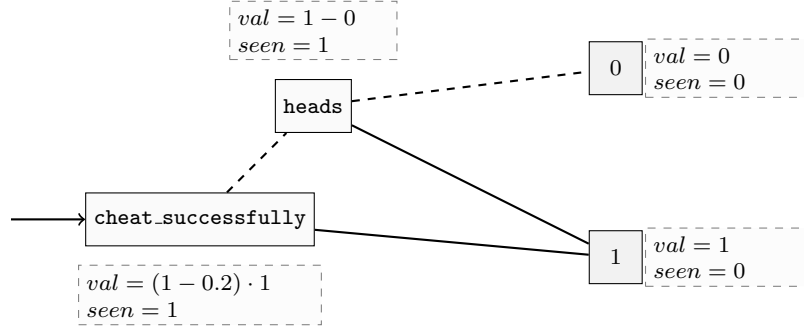
Fig. 4 Intermediate results when calculating the gradient $\partial P(\texttt{win})/\partial\texttt{heads}$ using Algorithm 4. The result is read off at the root node of the BDD.

The following example illustrates the gradient calculation on a simple query.

Example 1 (Gradient of a query). Consider a simple coin toss game: One can either win by getting heads or by cheating as described by the following theory:

$$?? :: \texttt{heads.} \qquad\qquad ?? :: \texttt{cheat\_succesfully.}$$
$$\texttt{win} : -\texttt{cheat\_successfully.}$$
$$\texttt{win} : -\texttt{heads.}$$

Suppose we want to estimate unknown fact probabilities (indicated by the symbol ??) from the training example $P(\texttt{win}) = 0.3$.

As a first step the fact probabilities get initialized with some random probabilities:

$$0.6 :: \texttt{heads.} \qquad\qquad 0.2 :: \texttt{cheat\_succesfully.}$$
$$\texttt{win} : -\texttt{cheat\_successfully.}$$
$$\texttt{win} : -\texttt{heads.}$$

In order to calculate the gradient of the MSE (cf. Equation (16)), the algorithm evaluates the partial derivative for every probabilistic fact and every training example. Figure 4 illustrates the calculation of the partial derivate $\partial P(\texttt{win})/\partial\texttt{heads}$ using Algorithm 4.

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 2 can be modified straightforwardly such that it calculates the value of the gradient (17) of a success probability. Algorithm 4 shows the pseudocode. Both algorithms have a time and space complexity of O(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 $n_1 \cdot n_2 \cdot \ldots \cdot n_i$, 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 $g_b = n_1 \cdot n_2 \cdot \ldots n_{j-1} \cdot n_{j+1} \cdot \ldots \cdot n_i$ 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 4. 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 $\mathbf{a} = a_1, \ldots, a_n$ initialized randomly, the gradient $\Delta \mathbf{a} = \Delta a_1, \ldots, \Delta a_n$ 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. For further details and experimental results in the context of the biological network application, we refer to [19, 46].

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[8], NCBI Entrez[9], 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. [48]).

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 [12]. 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 [12] 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. In this dataset, weights are associated to edges, indicating the probability that the corresponding nodes are related[10].

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 acyclic, indirected (symmetric) `path/2` relation.

Probabilistic inference (Section 3)

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 can-

---

[8] http://www.ensembl.org

[9] http://www.ncbi.nlm.nih.gov/Entrez/

[10] [12] view this strength or probability as the product of three factors, indicating the *reliability*, the *relevance* as well as the *rarity* (specificity) of the information.

didate genes for Alzheimer disease, ProbLog can easily be used to rank the genes by their likelihood of being relevant for AD.

Most likely explanation (Section 3.1)

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

When including node type information in the definition of a path between two nodes, the best (most likely) proof of `path('ROBO1','AD')` obtained by ProbLog is

```
node('ROBO1', gene),
edge('ROBO1', 'SLIT1',  interacts-with),
node('SLIT1', gene),
edge('SLIT1', 'hsa10q23.3-q24', is-located-in),
node('hsa10q23.3-q24', genomic-context),
edge('hsa10q23.3-q24', 'hsa10q24', contains),
node('hsa10q24', genomic-context),
edge('hsa10q24', 'AD', is-related-to),
node('AD', phenotype).
```

In other words, ROBO1 interacts with SLIT1, which is located in a genomic area related to AD. This proof has probability 0.14.

Most likely generalized explanation (Section 5)

Explanations obtained by probabilistic explanation based learning within ProbLog are on a more general level, that is, they replace constants occurring in a concrete proof by variables. By defining predicates related to node and edge types as operational, the proof above is generalized to explanation `exp_path(A, B)` ←

```
node(A, gene), edge(A, C, interacts-with),
node(C, gene), edge(C, D, is-located-in),
node(D, genomic-context), edge(D, E, contains),
node(E, genomic-context),
edge(E, B, is-related-to), node(B, phenotype).
```

Figure 5 shows four other 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.

```
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. 5 Additional explanation clauses for `path(A,B)`, connecting gene `A` to phenotype `B`, obtained from different examples.

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.

While the linear explanation used for illustration here could also be obtained using standard shortest-path algorithms, PEBL offers a more general framework for finding explanations where the structure is defined by background knowledge in the form of an arbitrary logic program.

Theory compression (Section 7)

The most likely explanation for *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 *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 *path('ROBO1', 'hsa10q24')* as a negative example, in order to remove connections using the genomic context from the extracted graph.

Local pattern mining (Section 6)

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 (Section 8)

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 Related Work in Statistical Relational Learning

In this section, we position ProbLog in the field of statistical relational learning [49] and probabilistic inductive logic programming [50]. In this context, its distinguishing features are that it is a probabilistic logic programming language based on Sato's distribution semantics [14], that it also can serve as a target language into which many of the other statistical relational learning formalisms can be compiled [51] and that several further approaches for learning ProbLog are being developed. Let us now discuss each of these aspects in turn.

First, ProbLog is closely related to some alternative formalisms such as PHA and ICL [7, 16], pD [6] and PRISM [8] as their semantics are all based on Sato's distribution semantics even though there exist also some subtle differences. However, ProbLog is – to the best of the authors' knowledge – the first implementation that tightly integrates Sato's original distribution semantics [14] in a state-of-the-art Prolog system without making additional restrictions (such as the exclusive explanation assumption made in PHA and PRISM). As ProbLog, both PRISM and the ICL implementation AILog2 use a two-step approach to inference, where proofs are collected in the first phase, and probabilities are calculated once all proofs are known. AILog2 is a meta-interpreter implemented in SWI-Prolog for didactical purposes, where the disjoint-sum-problem is tackled using a symbolic disjoining technique [16]. PRISM, built on top of B-Prolog, requires programs to be written such that

alternative explanations for queries are mutually exclusive. PRISM uses a meta-interpreter to collect proofs in a hierarchical datastructure called explanation graph. As proofs are mutually exclusive, the explanation graph directly mirrors the sum-of-products structure of probability calculation [8]. ProbLog is the first probabilistic logic programming system using BDDs as a basic datastructure for probability calculation, a principle that receives increased interest in the probabilistic logic learning community, cf. for instance [52, 53].

Furthermore, as compared to SLPs [54], CLP($\mathcal{BN}$) [55], and BLPs [43], ProbLog is a much simpler and in a sense more primitive probabilistic programming language. Therefore, the relationship between probabilistic logic programming and ProbLog is, in a sense, analogous to that between logic programming and Prolog. From this perspective, it is our hope and goal to further develop ProbLog so that it can be used as a general purpose programming language with an efficient implementation for use in statistical relational learning [49] and probabilistic programming [50]. One important use of such a probabilistic programming language is as a target language in which other formalisms can be efficiently compiled. For instance, it has already been shown that CP-logic [56], a recent elegant probabilistic knowledge representation language based on a probabilistic extension of clausal logic, can be compiled into ProbLog [52] and it is well-known that SLPs [54] can be compiled into Sato's PRISM, which is closely related to ProbLog. Further evidence is provided in [51].

Another, important use of ProbLog is as a vehicle for developing learning and mining algorithms and tools [13, 39, 19, 31], an aspect that we have also discussed in the present paper. In the context of probabilistic representations [49, 50], one typically distinguishes two types of learning: parameter estimation and structure learning. In parameter estimation in the context of ProbLog and PRISM, one starts from a set of queries and the logical part of the program and the problem is to find good estimates of the parameter values, that is, the probabilities of the probabilistic facts in the program. In the present paper and [19], we have discussed a gradient descent approach to parameter learning for ProbLog in which the examples are ground facts together with their target probability. In [57], an approach to learning from interpretations based on an EM algorithm is introduced. There, each example specifies a possible world, that is, a set of ground facts together with their truth value. This setting closely corresponds to the standard setting for learning in statistical relational learning systems such as Markov Logic [58] and probabilistic relational models [59]. In structure learning, one also starts from queries but has to find the logical part of the program as well. Structure learning is therefore closely related to inductive logic programming. An initial approach to learning the structure, that is, the rules of a ProbLog program has recently been introduced in [60].

11 Conclusions

In this chapter, we provided a survey of the developments around ProbLog, a simple probabilistic extension of Prolog based on the distribution semantics. This combination of definite clause logic and probabilities leads to an expressive general framework supporting both inductive and probabilistic querying. Indeed, probabilistic explanation based learning, local pattern mining, theory compression and parameter estimation as presented in this chapter all share a common core: they all use the probabilistic inference techniques offered by ProbLog to score queries or examples. ProbLog has been motivated by the need to develop intelligent tools for supporting life scientists analyzing large biological networks involving uncertain data. All techniques presented here have been evaluated in the context of such a biological network; we refer to [3, 13, 31, 39, 19] for details.

Acknowledgements

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) 152–164
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. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. J. Artif. Intell. Res. (JAIR) **15** (2001) 391–454

9. Poole, D.: Logic programming, abduction and probability. New Generation Computing **11** (1993) 377–400

10. Wrobel, S.: First order theory refinement. In De Raedt, L., ed.: Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996) 14 – 33

11. Richards, B.L., Mooney, R.J.: Automated refinement of first-order horn-clause domain theories. Machine Learning **19**(2) (1995) 95–131

12. Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., Toivonen, H.: Link discovery in graphs derived from biological databases. In: DILS. Volume 4075 of LNCS., Springer (2006) 35–49

13. Kimmig, A., De Raedt, L., Toivonen, H.: Probabilistic explanation based learning. In Kok, J.N., Koronacki, J., de Mantaras, R.L., Matwin, S., Mladenic, D., Skowron, A., eds.: 18th European Conference on Machine Learning (ECML). Volume 4701 of LNCS., Springer (2007) 176–187

14. Sato, T.: A statistical learning method for logic programs with distribution semantics. In Sterling, L., ed.: ICLP, MIT Press (1995) 715–729

15. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8) (1986) 677–691

16. Poole, D.: Abducing through negation as failure: stable models within the independent choice logic. Journal of Logic Programming **44**(1-3) (2000) 5–35

17. Lloyd, J.W.: Foundations of Logic Programming. 2. edn. Springer, Berlin (1989)

18. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM Journal on Computing **8**(3) (1979) 410–421

19. 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), Part I. Volume 5211 of LNCS (Lecture Notes In Computer Science)., Antwerp, Belgium, Springer Berlin/Heidelberg (September 2008) 473–488

20. Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., De Raedt, L.: On the Efficient Execution of ProbLog Programs. In de la Banda, M.G., Pontelli, E., eds.: International Conference on Logic Programming. Number 5366 in LNCS, Springer (December 2008) 175–189

21. Santos Costa, V.: The life of a logic programming system. In de la Banda, M.G., Pontelli, E., eds.: Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings. Volume 5366 of Lecture Notes in Computer Science., Springer (2008) 1–6

22. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499

23. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (January 1999) 31–54

24. Mantadelis, T., Demoen, B., Janssens, G.: A simplified fast interface for the use of CUDD for binary decision diagrams (2008) `http://people.cs.kuleuven.be/~theofrastos.mantadelis/tools/simplecudd.html`.

25. Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming (TPLP) (2010) to appear; `https://lirias.kuleuven.be/handle/123456789/259607`.

26. Mitchell, T.M., Keller, R.M., Kedar-Cabelli, S.T.: Explanation-based generalization: A unifying view. Machine Learning **1**(1) (1986) 47–80

27. DeJong, G., Mooney, R.J.: Explanation-based learning: An alternative view. Machine Learning **1**(2) (1986) 145–176

28. Hirsh, H.: Explanation-based generalization in a logic-programming environment. In: IJCAI'87: Proceedings of the 10th international joint conference on Artificial intelligence, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1987) 221–227

29. Van Harmelen, F., Bundy, A.: Explanation-based generalisation = partial evaluation. Artificial Intelligence **36**(3) (1988) 401–412

30. Langley, P.: Unifying themes in empirical and explanation-based learning. In: Proceedings of the sixth international workshop on Machine learning, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1989) 2–4

31. Kimmig, A., De Raedt, L.: Local query mining in a probabilistic Prolog. In Boutilier, C., ed.: International Joint Conference on Artificial Intelligence. (2009) 1095–1100

32. 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

33. 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

34. 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

35. 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 (2000) 263–270

36. 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

37. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery **1**(3) (1997) 241–258

38. 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

39. De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic Prolog programs. Machine Learning **70**(2-3) (2008) 151–168

40. Koppel, M., Feldman, R., Segre, A.M.: Bias-driven revision of logical domain theories. J. Artif. Intell. Res. (JAIR) **1** (1994) 159–208

41. 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

42. Paes, A., Revoredo, K., Zaverucha, G., Santos Costa, V.: Probabilistic first-order theory revision from examples. In Kramer, S., Pfahringer, B., eds.: ILP. Volume 3625 of Lecture Notes in Computer Science., Springer (2005) 295–311

43. Kersting, K., De Raedt, L.: Basic principles of learning bayesian logic programs. [50] 189–221

44. Gupta, R., Sarawagi, S.: Creating probabilistic databases from information extraction models. In: VLDB. (2006) 965–976

45. Chen, J., Muggleton, S., Santos, J.: Learning probabilistic logic models from probabilistic examples (extended abstract). In: ILP. (2007) 22–23

46. Gutmann, B., Kimmig, A., Kersting, K., De Raedt, L.: Parameter estimation in ProbLog from annotated queries. Technical Report CW 583, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (April 2010)

47. Saul, L., Jaakkola, T., Jordan, M.: Mean field theory for sigmoid belief networks. JAIR **4** (1996) 61–76

48. Perez-Iratxeta, C., Bork, P., Andrade, M.: Association of genes to genetically inherited diseases using data mining. Nature Genetics **31** (2002) 316–319

49. Getoor, L., Taskar, B., eds.: Statistical Relational Learning. The MIT press (2007)

50. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S., eds.: Probabilistic Inductive Logic Programming — Theory and Applications. Volume 4911 of Lecture Notes in Artificial Intelligence. Springer (2008)
51. 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.: Towards digesting the alphabet-soup of statistical relational learning. In Roy, D., Winn, J., McAllester, D., Mansinghka, V., Tenenbaum, J., eds.: Proceedings of the 1st Workshop on Probabilistic Programming: Universal Languages, Systems and Applications, Whistler, Canada (December 2008)
52. Riguzzi, F.: A top down interpreter for LPAD and CP-logic. In: AI*IA 2007: Artificial Intelligence and Human-Oriented Computing. Volume 4733 of LNCS. (2007)
53. Ishihata, M., Kameya, Y., Sato, T., ichi Minato, S.: Propositionalizing the EM algorithm by BDDs. In Železný, F., Lavrač, N., eds.: Proceedings of Inductive Logic Programming (ILP 2008), Late Breaking Papers, Prague, Czech Republic (September 2008) 44–49
54. Muggleton, S.: Stochastic logic programs. In De Raedt, L., ed.: ILP. (1995)
55. Santos Costa, V., Page, D., Cussens, J.: Clp(bn): Constraint logic programming for probabilistic knowledge. In: In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI03, Morgan Kaufmann (2003) 517–524
56. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In Demoen, B., Lifschitz, V., eds.: ICLP. Volume 3132 of LNCS., Springer, Heidelberg (2004) 431–445
57. Gutmann, B., Thon, I., De Raedt, L.: Learning the parameters of probabilistic logic programs from interpretations. Technical Report CW 584, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (April 2010)
58. Domingos, P., Lowd, D.: Markov Logic: an interface layer for AI. Morgan & Claypool (2009)
59. Getoor, L., Friedman, N., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In Džeroski, S., Lavrač, N., eds.: Relational Data Mining. Springer (2001) 307–335
60. De Raedt, L., Thon, I.: Probabilistic rule learning. Technical Report CW 580, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (April 2010)