On the Implementation of the Probabilistic Logic Programming Language ProbLog

Angelika Kimmig, Bart Demoen and Luc De Raedt
Departement Computerwetenschappen, K.U. Leuven
Celestijnenlaan 200A - bus 2402, B-3001 Heverlee, Belgium
(*e-mail:* {Angelika.Kimmig,Bart.Demoen,Luc.DeRaedt}@cs.kuleuven.be)

Vítor Santos Costa and Ricardo Rocha
CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
(*e-mail:* {vsc,ricroc}@dcc.fc.up.pt)

submitted 19 June 2009; revised 14 October 2009; accepted 9 December 2009

Abstract

The past few years have seen a surge of interest in the field of probabilistic logic learning and statistical relational learning. In this endeavor, many probabilistic logics have been developed. ProbLog is a recent probabilistic extension of Prolog motivated by the mining of large biological networks. In ProbLog, facts can be labeled with probabilities. These facts are treated as mutually independent random variables that indicate whether these facts belong to a randomly sampled program. Different kinds of queries can be posed to ProbLog programs. We introduce algorithms that allow the efficient execution of these queries, discuss their implementation on top of the YAP-Prolog system, and evaluate their performance in the context of large networks of biological entities.
To appear in Theory and Practice of Logic Programming (TPLP)

1 Introduction

In the past few years, a multitude of different formalisms combining probabilistic reasoning with logics, databases, or logic programming has been developed. Prominent examples include PHA and ICL (Poole 1993b; Poole 2000), PRISM (Sato and Kameya 2001), SLPs (Muggleton 1995), ProbView (Lakshmanan et al. 1997), CLP($\mathcal{BN}$) (Santos Costa et al. 2003), CP-logic (Vennekens et al. 2004), Trio (Widom 2005), probabilistic Datalog (pD) (Fuhr 2000), and probabilistic databases (Dalvi and Suciu 2004). Although these logics have been traditionally studied in the knowledge representation and database communities, the focus is now often on a machine learning perspective, which imposes new requirements. First, these logics must be simple enough to be learnable and at the same time sufficiently expressive to support interesting probabilistic inferences. Second, because learning is computationally expensive and requires answering long sequences of possibly complex queries, inference

in such logics must be fast, although inference in even the simplest probabilistic logics is computationally hard.

In this paper, we study these problems in the context of a simple probabilistic logic, ProbLog (De Raedt et al. 2007), which has been used for learning in the context of large biological networks where edges are labeled with probabilities. Large and complex networks of biological concepts (genes, proteins, phenotypes, etc.) can be extracted from public databases, and probabilistic links between concepts can be obtained by various techniques (Sevon et al. 2006). ProbLog is essentially an extension of Prolog where a program defines a distribution over all its possible non-probabilistic subprograms. Facts are labeled with probabilities and treated as mutually independent random variables indicating whether or not the corresponding fact belongs to a randomly sampled program. The success probability of a query is defined as the probability that it succeeds in such a random subprogram. The semantics of ProbLog is not new: it is an instance of the distribution semantics (Sato 1995). This is a well-known semantics for probabilistic logics that has been (re)defined multiple times in the literature, often in a more limited database setting; cf. (Dantsin 1991; Poole 1993b; Fuhr 2000; Poole 2000; Dalvi and Suciu 2004). Sato has, however, shown that the semantics is also well-defined in the case of a countably infinite set of random variables and formalized it in his well-known distribution semantics (Sato 1995). However, even though relying on the same semantics, in order to allow efficient inference, systems such as PRISM (Sato and Kameya 2001) and PHA (Poole 1993b) additionally require all proofs of a query to be mutually exclusive. Thus, they cannot easily represent the type of network analysis tasks that motivated ProbLog. ICL (Poole 2000) extends PHA to the case where proofs need not be mutually exclusive. In contrast to the ProbLog implementation presented here, Poole's AILog2, an implementation of ICL, uses a meta-interpreter and is not tightly integrated with Prolog.

We contribute exact and approximate inference algorithms for ProbLog. We present algorithms for computing the success and explanation probabilities of a query, and show how they can be efficiently implemented combining Prolog inference with Binary Decision Diagrams (BDDs) (Bryant 1986). In addition to an iterative deepening algorithm that computes an approximation along the lines of (Poole 1993a), we further adapt the Monte Carlo approach used by (Sevon et al. 2006) in the context of biological network inference. These two approximation algorithms compute an upper and a lower bound on the success probability. We also contribute an additional approximation algorithm that computes a lower bound using only the k most likely proofs.

The key contribution of this paper is the tight integration of these algorithms in the state-of-the-art YAP-Prolog system. This integration includes several improvements over the initial implementation used in (De Raedt et al. 2007), which are needed to use ProbLog to effectively query Sevon's Biomine network (Sevon et al. 2006) containing about 1,000,000 nodes and 6,000,000 edges, as will be shown in the experiments.

This paper is organised as follows. After introducing ProbLog and its semantics in Section 2, we present several algorithms for exact and approximate inference in

Section 3. Section 4 then discusses how these algorithms are implemented in YAP-Prolog, and Section 5 reports on experiments that validate the approach. Finally, Section 6 concludes and touches upon related work.

2 ProbLog

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, that is, these facts correspond to random variables. We assume that these variables are mutually independent.[1] The definite clauses allow one to add arbitrary *background knowledge* (BK).

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

$$\begin{array}{lll} 0.8 :: \texttt{edge(a,c)}. & 0.7 :: \texttt{edge(a,b)}. & 0.8 :: \texttt{edge(c,e)}. \\ 0.6 :: \texttt{edge(b,c)}. & 0.9 :: \texttt{edge(c,d)}. & 0.5 :: \texttt{edge(e,d)}. \end{array} \tag{1}$$

Such a probabilistic graph can be used to sample subgraphs by tossing a coin for each edge. Given a ProbLog program $T = \{p_1 :: c_1, \cdots, p_n :: c_n\} \cup BK$ and a finite set of possible substitutions $\{\theta_{j1}, \ldots \theta_{ji_j}\}$ for each probabilistic fact $p_j :: c_j$, let L_T denote the maximal set of *logical* facts that can be added to BK, that is, $L_T = \{c_1\theta_{11}, \ldots, c_1\theta_{1i_1}, \cdots, c_n\theta_{n1}, \ldots, c_n\theta_{ni_n}\}$. As the random variables corresponding to facts in L_T are mutually independent, the ProbLog program defines a probability distribution over ground logic programs $L \subseteq L_T$:

$$P(L|\,T) = \prod_{c_i\theta_j \in L} p_i \prod_{c_i\theta_j \in L_T \setminus L} (1 - p_i). \tag{2}$$

Since the background knowledge BK is fixed and there is a one-to-one mapping between ground definite clause programs and Herbrand interpretations, a ProbLog program thus also defines a distribution over its Herbrand interpretations. Sato has shown how this semantics can be generalized to the countably infinite case; we refer to (Sato 1995) for details. For ease of readability, in the remainder of this paper we will restrict ourselves to the finite case and assume all probabilistic facts in a ProbLog program to be ground. We extend our example with the following background knowledge:

$$\begin{array}{lll} \texttt{path(X,Y)} & :- & \texttt{edge(X,Y)}. \\ \texttt{path(X,Y)} & :- & \texttt{edge(X,Z)}, \texttt{path(Z,Y)}. \end{array} \tag{3}$$

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

$$P_s(q|\,T) = \sum_{L \subseteq L_T} P(q|L) \cdot P(L|\,T)\ , \tag{4}$$

[1] If the program contains multiple instances of the same fact, they correspond to different random variables, i.e. $\{p :: c\}$ and $\{p :: c, p :: c\}$ are different ProbLog programs.

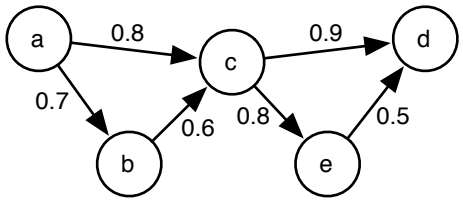

Figure 1. Example of a probabilistic graph: edge labels indicate the probability that the edge is part of the graph.

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

In our example, 40 of the 64 possible subprograms allow one to prove $path(c, d)$, namely all those that contain at least the edge from c to d or both the edge from c to e and from e to d, so the success probability of that query is the sum of the probabilities of these programs: $P_s(path(c, d)|\,T) = P(\{ab, ac, bc, cd, ce, ed\}|\,T) + \ldots + P(\{cd\}|\,T) = 0.94$, where xy is used as a shortcut for $edge(x, y)$ when listing elements of a subprogram. We will use this convention throughout the paper. Clearly, listing all subprograms is infeasible in practice; an alternative approach will be discussed in Section 3.1.

A ProbLog program also defines the probability of a *specific* proof E, also called *explanation*, of some query q, which is again a marginal of $P(L|\,T)$. Here, an *explanation* is a minimal subset of the probabilistic facts that together with the background knowledge entails $q\theta$ for some substitution θ. Thus, the probability of such an explanation E is that of sampling a logic program $L \cup E$ that contains at least all the probabilistic facts in E, that is, the marginal with respect to these facts:

$$P(E|\,T) = \sum\nolimits_{L \subseteq (L_T \setminus E)} P(L \cup E|\,T) = \prod\nolimits_{c_i \in E} p_i \tag{5}$$

The *explanation probability* $P_x(q|\,T)$ is then defined as the probability of the most likely explanation or proof of the query q

$$P_x(q|\,T) = \max\nolimits_{E \in E(q)} P(E|\,T) = \max\nolimits_{E \in E(q)} \prod_{c_i \in E} p_i, \tag{6}$$

where $E(q)$ is the set of all explanations for query q, i.e., all minimal sets $E \subseteq L_T$ of probabilistic facts such that $E \cup BK \models q$ (Kimmig et al. 2007).

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

The ProbLog semantics is essentially a distribution semantics (Sato 1995). 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 (allowing functors), that is, of the background knowledge BK together with a subprogram $L \subseteq L_T$; for further details we refer to (Sato 1995). The distribution semantics has been used widely in the literature, though often under other names or in a more

restricted setting; see e.g. (Dantsin 1991; Poole 1993b; Fuhr 2000; Poole 2000; Dalvi and Suciu 2004).

3 Inference in ProbLog

This section discusses algorithms for computing exactly or approximately the success and explanation probabilities of ProbLog queries. It additionally contributes a new algorithm for Monte Carlo approximation of success probabilities.

3.1 Exact Inference

Calculating the *success probability* of a query using Equation (4) directly is infeasible for all but the tiniest programs, as the number of subprograms to be checked is exponential in the number of probabilistic facts. However, as we have seen in our example in Section 2, we can describe all subprograms allowing for a specific proof by means of the facts that such a program has to contain, i.e., all the ground probabilistic facts used in that proof. As probabilistic facts correspond to random variables indicating the presence of facts in a sampled program, we alternatively denote proofs by conjunctions of such random variables. In our example, query *path(c,d)* has two proofs in the full program: {*edge(c,d)*} and {*edge(c,e)*,*edge(e,d)*}, or, using logical notation, cd and $ce \wedge ed$. The set of all subprograms containing *some* proof thus can be described by a disjunction over all possible proofs, in our case, $cd \vee (ce \wedge ed)$. This idea forms the basis for the inference method presented in (De Raedt et al. 2007), which uses two steps:

1. Compute the proofs of the query q in the logical part of the theory T, that is, in $BK \cup L_T$. The result will be a DNF formula.
2. Compute the probability of this formula.

Similar approaches are used for PRISM (Sato and Kameya 2001), ICL (Poole 2000) and pD (Fuhr 2000).

The probability of a single given proof, cf. Equation (5), is the marginal over all programs allowing for that proof, and thus equals the product of the probabilities of the facts used by that proof. However, we cannot directly sum the results for the different proofs to obtain the success probability, as a specific subprogram can allow several proofs and therefore contributes to the probability of each of these proofs. Indeed, in our example, all programs that are supersets of {*edge(c,e)*,*edge(e,d)*,*edge(c,d)*} contribute to the marginals of both proofs and would therefore be counted twice if summing the probabilities of the proofs. However, for mutually exclusive conjunctions, that is, conjunctions describing disjoint sets of subprograms, the probability is the sum of the individual probabilities. This situation can be achieved by adding *negated* random variables to a conjunction, thereby explicitly excluding subprograms covered by another part of the formula from the corresponding part of the sum. In the example, extending $ce \wedge ed$ to

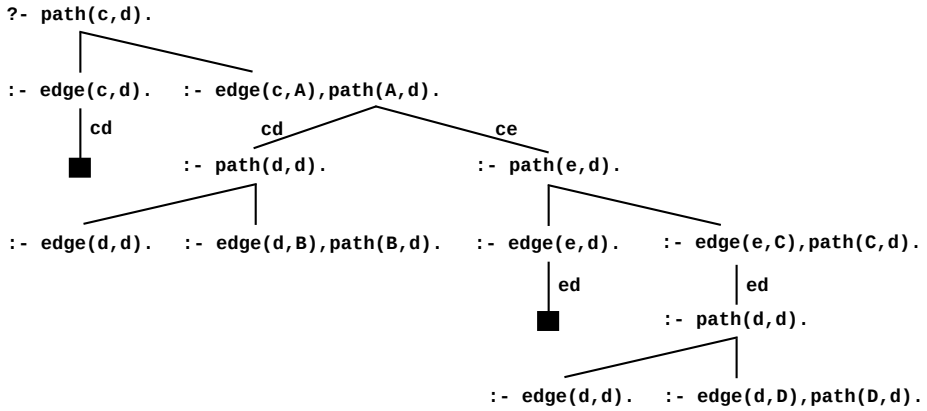

Figure 2. SLD-tree for query $path(c, d)$.

$ce \wedge ed \wedge \neg cd$ reduces the second part of the sum to those programs not covered by the first:

$$P_s(path(c,d)|\,T) = P(cd \vee (ce \wedge ed)|\,T)$$

$$= P(cd|\,T) + P(ce \wedge ed \wedge \neg cd|\,T)$$

$$= 0.9 + 0.8 \cdot 0.5 \cdot (1 - 0.9) = 0.94$$

However, as the number of proofs grows, disjoining them gets more involved. Consider for example the query *path(a,d)* which has four different but highly interconnected proofs. In general, this problem is known as the *disjoint-sum-problem* or the two-terminal network reliability problem, which is #P-complete (Valiant 1979).

Before returning to possible approaches to tackle the disjoint-sum-problem at the end of this section, we will now discuss the two steps of ProbLog's exact inference in more detail.

Following Prolog, the first step employs SLD-resolution 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 ground probabilistic facts $\{p_1 :: c_1, \cdots, p_k :: c_k\} \subseteq T$. 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 ground probabilistic fact $p_i :: c_i \in T$, indicating whether c_i is in a sampled logic program, that is, b_i has probability p_i of being true.[2] A particular proof of query q involving ground facts $\{p_1 :: c_1, \cdots, p_k :: c_k\} \subseteq T$ is thus represented by the conjunctive formula $b_1 \wedge \cdots \wedge b_k$, which at the same time represents the set of all subprograms containing these facts. Furthermore, using $E(q)$ to denote the set of proofs or explanations of the goal q, the set of all subprograms containing *some* proof of q can be denoted

[2] For better readability, we do not write substitutions explicitly here.

by $\bigvee_{e \in E(q)} \bigwedge_{c_i \in e} b_i$, as the following derivation shows:

$$\begin{aligned}
\bigvee_{e \in E(q)} \bigwedge_{c_i \in e} b_i &= \bigvee_{e \in E(q)} \left(\bigwedge_{c_i \in e} b_i \wedge \bigwedge_{c_i \in L_T \setminus e} (b_i \vee \neg b_i) \right) \\
&= \bigvee_{e \in E(q)} \bigvee_{L \subseteq L_T \setminus e} \left(\bigwedge_{c_i \in e} b_i \wedge \left(\bigwedge_{c_i \in L} b_i \wedge \bigwedge_{c_i \in L_T \setminus (L \cup e)} \neg b_i \right) \right) \\
&= \bigvee_{e \in E(q), L \subseteq L_T \setminus e} \left(\bigwedge_{c_i \in L \cup e} b_i \wedge \bigwedge_{c_i \in L_T \setminus (L \cup e)} \neg b_i \right) \\
&= \bigvee_{L \subseteq L_T, \exists \theta L \cup BK \models q\theta} \left(\bigwedge_{c_i \in L} b_i \wedge \bigwedge_{c_i \in L_T \setminus L} \neg b_i \right)
\end{aligned}$$

We first add all possible ways of extending a proof e to a full sampled program by considering each fact not in e in turn. We then note that the disjunction of these fact-wise extensions can be written on the basis of sets. Finally, we rewrite the condition of the disjunction in the terms of Equation (4). This is possible as each subprogram that is an extension of an explanation of q entails some ground instance of q, and vice versa, each subprogram entailing q is an extension of some explanation of q. As the DNF now contains conjunctions representing fully specified programs, its probability is a sum of products, which directly corresponds to Equation (4):

$$\begin{aligned}
&P\Big(\bigvee_{L \subseteq L_T, \exists \theta L \cup BK \models q\theta} \left(\bigwedge_{c_i \in L} b_i \wedge \bigwedge_{c_i \in L_T \setminus L} \neg b_i \right) \Big) \\
&= \sum_{L \subseteq L_T, \exists \theta L \cup BK \models q\theta} \left(\prod_{c_i \in L} p_i \cdot \prod_{c_i \in L_T \setminus L} (1 - p_i) \right) \\
&= \sum_{L \subseteq L_T, \exists \theta L \cup BK \models q\theta} P(L|T)
\end{aligned}$$

We thus obtain the following alternative characterisation of the success probability:

$$P_s(q|T) = P\left(\bigvee_{e \in E(q)} \bigwedge_{c_i \in e} b_i \right) \tag{7}$$

where $E(q)$ denotes the set of proofs or explanations of the goal q and b_i denotes the Boolean variable corresponding to ground probabilistic fact $p_i :: c_i$. 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.

However, as argued above, due to overlap between different conjunctions, the proof-based DNF of Equation (7) cannot directly be transformed into a sum of products. Computing the probability of DNF formulae thus involves solving the disjoint-sum-problem, and therefore is itself a #P-hard problem. Various algorithms have been developed to tackle this problem. The pD-engine HySpirit (Fuhr 2000)

uses the inclusion-exclusion principle, which is reported to scale to about ten proofs. For ICL, which extends PHA by allowing non-disjoint proofs, (Poole 2000) proposes a symbolic disjoining algorithm, but does not report scalability results. Our implementation of ProbLog employs Binary Decision Diagrams (BDDs) (Bryant 1986), an efficient graphical representation of a Boolean function over a set of variables, which scales to tens of thousands of proofs; see Section 4.4 for more details. PRISM (Sato and Kameya 2001) and PHA (Poole 1993b) differ from the systems mentioned above in that they avoid the disjoint-sum-problem by requiring the user to write programs such that proofs are guaranteed to be disjoint.

On the other hand, as the *explanation probability* P_x exclusively depends on the probabilistic facts used in one proof, it can be calculated using a simple branch-and-bound approach based on the SLD-tree, where partial proofs are discarded if their probability drops below that of the best proof found so far.

3.2 Approximative Inference

As the size of the DNF formula grows with the number of proofs, its evaluation can become quite expensive, and ultimately 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.

3.2.1 Bounded Approximation

The first approximation algorithm, a slight variant of the one proposed in (De Raedt et al. 2007), uses DNF formulae to obtain both an upper and a lower bound on the probability of a query. It is closely related to work by (Poole 1993a) in the context of PHA, but adapted towards ProbLog. The method relies on two observations.

First, we remark that the DNF formula describing sets of proofs is *monotone*, meaning that adding more proofs will never decrease the probability of the formula being true. Thus, formulae describing subsets of the full set of proofs of a query will always give a lower bound on the query's success probability. In our example, the lower bound obtained from the shorter proof would be $P(cd|\,T) = 0.9$, while that from the longer one would be $P(ce \wedge ed|\,T) = 0.4$.

Our second observation is that the probability of a proof $b_1 \wedge \ldots \wedge b_n$ will always be at most the probability of an arbitrary prefix $b_1 \wedge \ldots \wedge b_i$, $i \leq n$. In our example, the probability of the second proof will be at most the probability of its first edge from c to e, i.e., $P(ce|\,T) = 0.8 \geq 0.4$. As disjoining sets of proofs, i.e., including information on facts that are *not* elements of the subprograms described by a certain proof, can only decrease the contribution of single proofs, this upper bound carries over to a set of proofs or partial proofs, as long as prefixes for all possible proofs are included. Such sets can be obtained from an incomplete SLD-tree, i.e., an SLD-tree where branches are only extended up to a certain point.

This motivates ProbLog's *bounded approximation algorithm*. The algorithm relies on a probability threshold γ to stop growing the SLD-tree and thus obtain

Algorithm 1 Bounded approximation using iterative deepening with probability thresholds.

function BOUNDS(interval width δ_p, initial threshold γ, constant $\beta \in (0,1)$)
  $d_1 =$ FALSE; $d_2 =$ FALSE; $P(d_1|T) = 0$; $P(d_2|T) = 1$;
  while $P(d_2|T) - P(d_1|T) > \delta_p$ **do**
    $p =$TRUE;
    repeat
      Expand current proof p
    until either p:
        (a) Fails, in this case backtrack to the remaining choice points;
        (b) Succeeds, in this case set $d_1 := d_1 \vee p$ and $d_2 := d_2 \vee p$;
        (c) $P(p|T) < \gamma$, in this case set $d_2 := d_2 \vee p$
    if $d_2 ==$ FALSE **then**
      set $d_2 =$ TRUE
    Compute $P(d_1|T)$ and $P(d_2|T)$
    $\gamma := \gamma \cdot \beta$
  return $[P(d_1|T), P(d_2|T)]$

DNF formulae for the two bounds[3]. 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. Our goal is therefore to grow d_1 and d_2 in order to decrease $P(d_2|T) - P(d_1|T)$.

Given an acceptance threshold δ_p, an initial probability threshold γ, and a shrinking factor $\beta \in (0,1)$, the algorithm proceeds in an iterative-deepening manner as outlined in Algorithm 1. Initially, both d_1 and d_2 are set to FALSE, the neutral element with respect to disjunction, and the probability bounds are 0 and 1, as we have no full proofs yet, and the empty partial proof holds in any model.

It should be clear that $P(d_1|T)$ monotonically increases, as the number of proofs never decreases. On the other hand, as explained above, if d_2 changes from one iteration to the next, this is always because a partial proof p is either removed from d_2 and therefore no longer contributes to the probability, or it is replaced by proofs $p_1, \ldots, p_n$, such that $p_i = p \wedge s_i$, hence $P(p_1 \vee \ldots \vee p_n|T) = P(p \wedge s_1 \vee \ldots \vee p \wedge s_n|T) = P(p \wedge (s_1 \vee \ldots \vee s_n)|T)$. As proofs are subsets of the probabilistic facts in the ProbLog program, each literal's random variable appears at most once in the conjunction representing a proof, even if the corresponding subgoal is called multiple times when constructing the proof. We therefore know that the literals in the prefix p cannot be in any suffix s_i, hence, given ProbLog's independence assumption, $P(p \wedge (s_1 \vee \ldots \vee s_n)|T) = P(p|T)P(s_1 \vee \ldots \vee s_n|T) \leq P(p|T)$. Therefore, $P(d_2)$ monotonically decreases.

As an illustration, consider a probability threshold $\gamma = 0.9$ for the SLD-tree in

[3] Using a probability threshold instead of the depth bound of (De Raedt et al. 2007) has been found to speed up convergence, as upper bounds have been found to be tighter on initial levels.

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 $d = cd \vee (ce \wedge ed)$. The lower bound thus is 0.9, the upper bound (obtained by disjoining d_2 to $cd \vee (ce \wedge \neg cd)$) is 0.98, whereas the true probability is 0.94.

Notice that in order to implement this algorithm we need to compute the probability of a set of proofs. This task will be described in detail in Section 4.

3.2.2 K-Best

Using a fixed number of proofs to approximate the probability allows 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. (Gutmann et al. 2008) therefore introduces the k-probability $P_k(q|T)$, which approximates the success probability by using the k-best (that is, the k most likely) explanations instead of all proofs when building the DNF formula used in Equation (7):

$$P_k(q|T) = P\left(\bigvee_{e \in E_k(q)} \bigwedge_{b_i \in var(e)} b_i\right) \tag{8}$$

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 $k = \infty$ leads to the success probability, whereas $k = 1$ corresponds to the explanation probability provided that there is a single best proof. The branch-and-bound approach used to calculate the explanation probability can directly be generalized to finding the k-best proofs; cf. also (Poole 1993b).

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 $k = 2$, the 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 $k \geq 4$.

3.2.3 Monte Carlo

As an alternative approximation technique, we propose a Monte Carlo method, where we proceed as follows.
Execute until convergence:

1. Sample a logic program from the ProbLog program
2. Check for the existence of some proof of the query of interest

3. Estimate the query probability P as the fraction of samples where the query is provable

We estimate convergence by computing the 95% confidence interval at each m samples. Given a large number N of samples, we can use the standard normal approximation interval to the binomial distribution:

$$\delta \approx 2 \times \sqrt{\frac{P \cdot (P-1)}{N}}$$

Notice that confidence intervals do not directly correspond to the exact bounds used in our previous approximation algorithm. Still, we employ the same stopping criterion, that is, we run the Monte Carlo simulation until the width of the confidence interval is at most δ_p.

A similar algorithm (without the use of confidence intervals) was also used in the context of biological networks (not represented as Prolog programs) by (Sevon et al. 2006). The use of a Monte Carlo method for probabilistic logic programs was suggested already by (Dantsin 1991), although he neither provides details nor reports on an implementation. Our approach differs from the MCMC method for Stochastic Logic Programs (SLPs) introduced by (Cussens 2000) in that we do not use a Markov chain, but restart from scratch for each sample. Furthermore, SLPs are different in that they directly define a distribution over all proofs of a query. Investigating similar probabilistic backtracking approaches for ProbLog is a promising future research direction.

4 Implementation

This section discusses the main building blocks used to implement ProbLog on top of the YAP-Prolog system. 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. We have discussed two very different approaches to the problem:

- In exact inference, k-best and bounded approximation, the engine explicitly reasons about probabilities of proofs. The challenge is how to compute the probability of each individual proof, store a large number of proofs, and compute the probability of sets of proofs.

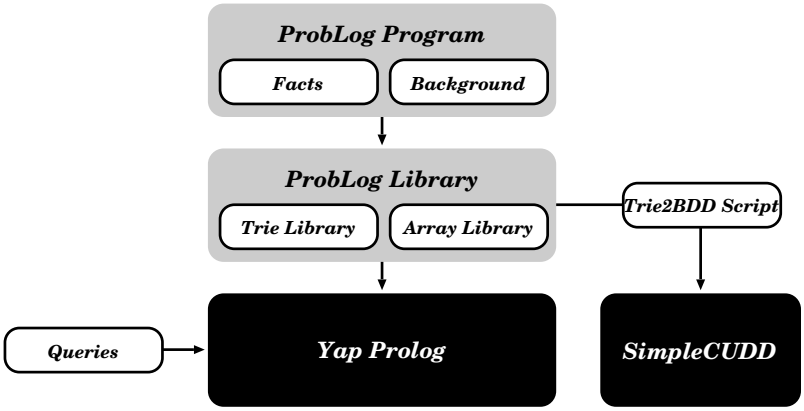

Figure 3. ProbLog Implementation: A ProbLog program (top) requires the ProbLog library which in turn relies on functionality from the tries and array libraries. ProbLog queries (bottom-left) are sent to the YAP engine, and may require calling the BDD library CUDD via SimpleCUDD.

- In Monte Carlo, 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 above can be abstracted as follows:

- Initialise the inference algorithm;
- While probabilistic inference did not converge:
  - initialise 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 :: edge('PubMed_2196878','MIM_609065').
0.659 :: edge('PubMed_8764571','HGNC_5014').
```
(9)

would be compiled as:

```
edge(A,B) :- problog_edge(ID,A,B,LogProb),
             grounding_id(edge(A,B),ID,GroundID),
             add_to_proof(GroundID,LogProb).

problog_edge(0,'PubMed_2196878','MIM_609065',-0.3348).
problog_edge(1,'PubMed_8764571','HGNC_5014',-0.4166).
```
(10)

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. Compared to the original meta-interpreter based implementation of (De Raedt et al. 2007), the main benefit of source-to-source transformation is better scalability, namely by having a compact representation of the facts for the YAP engine (Santos Costa 2007) and by allowing access to the YAP indexing mechanism (Santos Costa et al. 2007).

4.2 Proof Manipulation

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. Grounding identifiers are based on the fact's identifier extended with a grounding number, i.e. 5_1 and 5_2 would refer to different groundings of the non-ground fact with identifier 5. 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.

As we have discussed above, the actual number of proofs can grow very quickly.

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

ProbLog compactly represents a proof as a list of numbers. We would further like to have a scalable implementation of *sets* of proofs, such that we can compute the joint *probability* of large sets of proofs efficiently. Our representation for sets of proofs and our algorithm for computing the probability of such a set are discussed next.

4.3 Sets of Proofs

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* 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 (Ramakrishnan et al. 1999), which we briefly summarize here.

Tries (Fredkin 1962) were originally invented to index dictionaries, and have since been generalised to index recursive data structures such as terms. Please refer to (Bachmair et al. 1993; Graf 1996; Ramakrishnan et al. 1999) for the use of tries in automated theorem proving, term rewriting and tabled logic programs. An essential property of the trie data structure is that common prefixes are stored only once. A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term described by the tokens labelling the nodes traversed. For example, the tokenized form of the term $f(g(a), 1)$ is the sequence of 4 tokens: $f/2$, $g/1$, a and 1. Two terms with common prefixes will branch off from each other at the first distinguishing token.

Trie's internal nodes are four field data structures, storing the node's token, a pointer to the node's first child, a pointer to the node's parent and a pointer to the node's next sibling, respectively. Each internal node's outgoing transitions may be determined by following the child pointer to the first child node and, from there, continuing sequentially through the list of sibling pointers. When a list of sibling nodes becomes larger than a threshold value (8 in our implementation), we dynamically index the nodes through a hash table to provide direct node access and therefore optimise the search. Further hash collisions are reduced by dynamically expanding the hash tables. Inserting a term requires in the worst case allocating as many nodes as necessary to represent its complete path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node.

In order to minimize the number of nodes when storing proofs in a trie, we use Prolog lists to represent proofs. For example, a ProbLog proof $[3, 5\_1, 7, 5\_2]$ uses ground fact 3, a first grounding of fact 5, ground fact 7 and another grounding of fact 5, that is, list elements in proofs are always either integers or two integers with an underscore in between.

Figure 4 presents an example of a trie storing three proofs. Initially, the trie

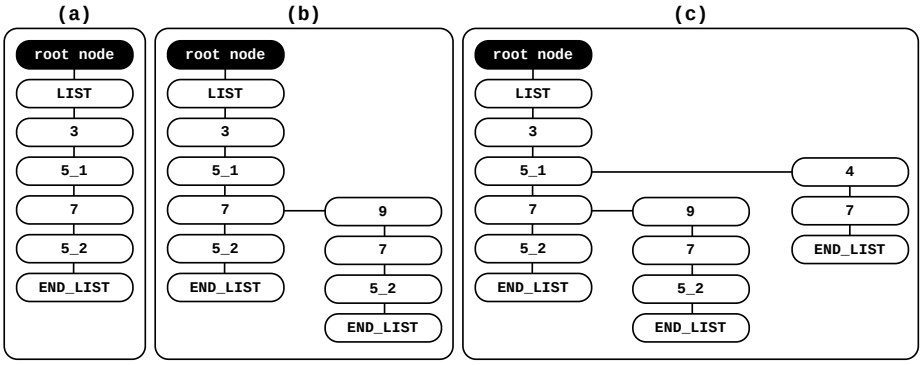

Figure 4. Using tries to store proofs. Initially, the trie contains the root node only. Next, we store the proofs: (a) $[3, 5\_1, 7, 5\_2]$; (b) $[3, 5\_1, 9, 7, 5\_2]$; and (c) $[3, 4, 7]$.

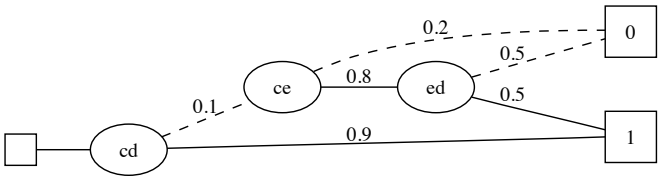

Figure 5. Binary Decision Diagram encoding the DNF formula $cd \vee (ce \wedge ed)$, corresponding to the two proofs of query *path(c,d)* in the example graph. An internal node labeled xy represents the Boolean variable for the edge between x and y, solid/dashed edges correspond to values true/false and are labeled with the probability that the variable takes this value.

contains the root node only. Next, we store the proof $[3, 5\_1, 7, 5\_2]$ and six nodes (corresponding to six tokens) are added to represent it (Figure 4(a)). The proof $[3, 5\_1, 9, 7, 5\_2]$ is then stored which requires seven nodes. As it shares a common prefix with the previous proof, we save the three initial nodes common to both representations (Figure 4(b)). The proof $[3, 4, 7]$ is stored next and we save again the two initial nodes common to all proofs (Figure 4(c)).

4.4 Binary Decision Diagrams

To efficiently compute the probability of a DNF formula representing a set of proofs, our implementation represents this formula as a reduced ordered Binary Decision Diagram (BDD) (Bryant 1986), which can be viewed as a compact encoding of a Boolean decision tree. 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

Algorithm 2 Translating a trie T representing a DNF to a BDD generation script. $\textsc{Replace}(T, C, n_i)$ replaces each occurence of C in T by n_i.

function TRANSLATE(trie T)
  $i := 1$
  while $\neg leaf(T)$ **do**
    $S_\wedge := \{(C, P) | C$ leaf in T and single child of its parent $P\}$
    for all $(C, P) \in S_\wedge$ **do**
      write $n_i = P \wedge C$
      $T := \textsc{Replace}(T, (C, P), n_i)$
      $i := i + 1$
    $S_\vee := \{[C_1, \ldots, C_n] ||$ leaves C_j are all the children of some parent P in $T\}$
    for all $[C_1, \ldots, C_n] \in S_\vee$ **do**
      write $n_i = C_1 \vee \ldots \vee C_n$
      $T := \textsc{Replace}(T, [C_1, \ldots, C_n], n_i)$
      $i := i + 1$
  write $top = n_{i-1}$

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 5 shows the BDD for the existence of a path between c and d in our earlier example.

We use SimpleCUDD[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 code for all its subtrees. Algorithm 2 gives the details of this procedure. 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

[5] `http://www.cs.kuleuven.be/~theo/tools/simplecudd.html`
[6] `http://vlsi.colorado.edu/~fabio/CUDD`

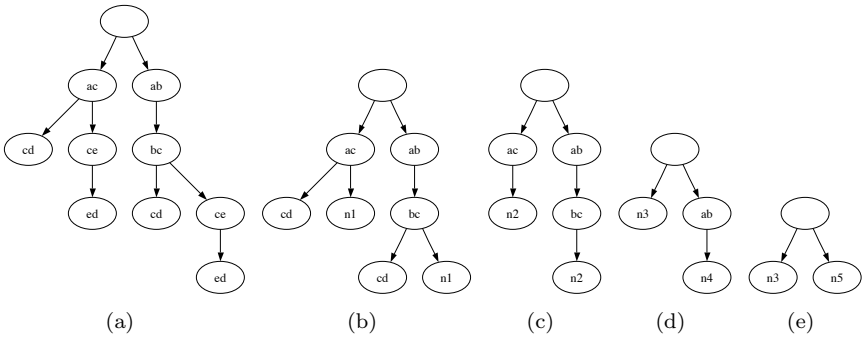

Figure 6. Translating the DNF for *path(a,d)*.

Algorithm 3 Calculating the probability of a BDD.

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

a very different structure than the trie. The translation for query *path(a,d)* in our example graph is illustrated in Figure 6, it results in the following script:

$$\begin{aligned} n1 &= ce \wedge ed \\ n2 &= cd \vee n1 \\ n3 &= ac \wedge n2 \\ n4 &= bc \wedge n2 \\ n5 &= ab \wedge n4 \\ n6 &= n3 \vee n5 \\ top &= n6 \end{aligned}$$

After CUDD has generated the BDD, the probability of a formula is calculated by traversing the BDD, in each node summing the probability of the high and low child, weighted by the probability of the node's variable being assigned true and false respectively, cf. Algorithm 3. Intermediate results are cached, and the algorithm has a time and space complexity linear in the size of the BDD. For illustration, consider again Figure 5. The algorithm starts by assigning probabilities 0 and 1 to the 0- and 1-leaf respectively. The node labeled *ed* has probability $0.5 \cdot 1 + 0.5 \cdot 0 = 0.5$, node *ce* has probability $0.8 \cdot 0.5 + 0.2 \cdot 0 = 0.4$; finally, node *cd*, and thus the entire formula, has probability $0.9 \cdot 1 + 0.1 \cdot 0.4 = 0.94$.

Algorithm 4 Monte Carlo Inference.

function MONTECARLO(query q, interval width δ_p, constant m)
  $c = 0$; $i = 0$; $p = 0$; $\delta = 1$;
  while $\delta > \delta_p$ **do**
    Generate a sample P';
    if $P' \models q$ **then**
      $c := c + 1$;
    $i := i + 1$;
    if i mod $m == 0$ **then**
      $p := c/i$
      $\delta := 2 \times \sqrt{\frac{p \cdot (p-1)}{i}}$
  return p

4.5 Monte Carlo

The Monte Carlo implementation is shown in Algorithm 4. It receives a query q, an acceptance threshold δ_p and a constant m determining the number of samples generated per iteration. At the end of each iteration, it estimates the probability p as the fraction of programs sampled over all previous iterations that entailed the query, and the confidence interval width to be used in the stopping criterion as explained in Section 3.2.3. 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.

Our first approach was to generate a complete sample and to check for a proof. In order to accelerate the process, proofs were cached in a trie to skip inference on a new sample. If no proofs exist on a cache, we call the standard Prolog refutation procedure. Although this approach works rather well for small databases, it does not scale to larger databases where just generating a new sample requires walking through millions of facts.

We observed that even in large programs proofs are often quite short, i.e., we only need to verify whether facts from a small fragment of the database are in the sample. This suggests that it may be a good idea to take advantage of the independence between facts and generate the sample *lazily*: we verify whether a fact is in the sample only when we need it for a proof. YAP represents samples compactly as a three-valued array with one field for each fact, where 0 means the fact was not yet sampled, 1 it was already sampled and belongs to the sample, 2 it was already sampled and does not belong to the sample. In this implementation:

1. New samples are generated by resetting the sampling array.
2. At every call to `add_to_proof`, given the current ProbLog literal f:
   (a) if $s[f] == 0$, $s[f] = sample(f)$;
   (b) if $s[f] == 1$, succeed;
   (c) if $s[f] == 2$, fail;

Note that as fact identifiers are used to access the array, the approach cannot directly be used for non-ground facts. The current implementation of Monte Carlo therefore uses the internal database to store the result of sampling different groundings of such facts.

5 Experiments

We performed experiments with our implementation of ProbLog in the context of the biological network obtained from the Biomine project (Sevon et al. 2006). We used two subgraphs extracted around three genes known to be connected to the Alzheimer disease (HGNC numbers 983, 620 and 582) as well as the full network. The smaller graphs were obtained querying Biomine for best paths of length 2 (resulting in graph SMALL) or all paths of length 3 (resulting in graph MEDIUM) starting at one of the three genes. SMALL contains 79 nodes and 144 edges, MEDIUM 5220 nodes and 11532 edges. We used SMALL for a first comparison of our algorithms on a small scale network where success probabilities can be calculated exactly. Scalability was evaluated using both MEDIUM and the entire BIOMINE network with roughly 1,000,000 nodes and 6,000,000 edges. In all experiments, we queried for the probability that two of the gene nodes mentioned above are connected, that is, we used queries such as `path('HGNC_983','HGNC_620',Path)`. We used the following definition of an acyclic path in our background knowledge:

```
path(X,Y,A)      :-  path(X,Y,[X],A),
path(X,X,A,A).
path(X,Y,A,R)    :-  X \== Y,
                     edge(X,Z),
                     absent(Z,A),
                     path(Z,Y,[Z|A],R).
```
(11)

As list operations to check for the absence of a node get expensive for long paths, we consider an alternative definition for use in Monte Carlo. It provides cheaper testing by using the internal database of YAP to store nodes on the current path under key `visited`:

```
memopath(X,Y,A)      :-  eraseall(visited),
                         memopath(X,Y,[X],A).
memopath(X,X,A,A).
memopath(X,Y,A,R)    :-  X \== Y,
                         edge(X,Z),
                         recordzifnot(visited,Z,_),
                         memopath(Z,Y,[Z|A],R).
```
(12)

path **k**	983 − 620 T_P	T_B	P	983 − 582 T_P	T_B	P	620 − 582 T_P	T_B	P
1	0	13	0.07	0	7	0.05	0	26	0.66
2	0	12	0.08	0	6	0.05	0	6	0.66
4	0	12	0.10	10	6	0.06	0	6	0.86
8	10	12	0.11	0	6	0.06	0	6	0.92
16	0	12	0.11	10	6	0.06	0	6	0.92
32	20	34	0.11	10	17	0.07	0	7	0.96
64	20	74	0.11	10	46	0.09	10	38	0.99
128	50	121	0.11	40	161	0.10	20	257	1.00
256	140	104	0.11	80	215	0.10	90	246	1.00
512	450	118	0.11	370	455	0.11	230	345	1.00
1024	1310	537	0.11	950	494	0.11	920	237	1.00
exact	670	450	0.11	8060	659	0.11	630	721	1.00

Table 1. k-probability on SMALL.

Finally, to assess performance on the full network for queries with smaller probabilities, we use the following definition of paths with limited length:

```
lenpath(N,X,Y,Path)  :-  lenpath(N,X,Y,[X],Path).
lenpath(N,X,X,A,A)   :-  N ≥ 0.
lenpath(N,X,Y,A,P)   :-  X\ == Y,
                         N > 0,
                         edge(X,Z),
                         absent(Z,A),
                         NN is N − 1,
                         lenpath(NN,Z,Y,[Z|A],P).
```
(13)

All experiments were performed on a Core 2 Duo 2.4 GHz 4 GB machine running Linux. All times reported are in `msec` and do not include the time to load the graph into Prolog. The latter takes 20, 200 and 78140 `msec` for SMALL, MEDIUM and BIOMINE respectively. Furthermore, as YAP indexes the database at query time, we query for the explanation probability of `path('HGNC_620','HGNC_582',Path)` before starting runtime measurements. This takes 0, 50 and 25900 `msec` for SMALL, MEDIUM and BIOMINE respectively. We report T_P, the time spent by ProbLog to search for proofs, as well as T_B, the time spent to execute BDD programs (whenever meaningful). We also report the estimated probability P. For approximate inference using bounds, we report exact intervals for P, and also include the number n of BDDs constructed. We set both the initial threshold and the shrinking factor to 0.5. We computed k-probability for $k = 1, 2, \ldots, 1024$. In the bounding algorithms, the error interval ranged between 10% and 1%. Monte Carlo recalculates confidence intervals after $m = 1000$ samples. We also report the number S of samples used.

path	$983-620$				$983-582$				$620-582$			
δ	T_P	T_B	n	P	T_P	T_B	n	P	T_P	T_B	n	P
0.10	0	48	4	[0.07,0.12]	10	74	6	[0.06,0.11]	0	25	2	[0.91,1.00]
0.05	0	71	6	[0.07,0.11]	0	75	6	[0.06,0.11]	0	486	4	[0.98,1.00]
0.01	0	83	7	[0.11,0.11]	140	3364	10	[0.10,0.11]	60	1886	6	[1.00,1.00]

Table 2. Inference using bounds on SMALL.

path	$983-620$			$983-582$			$620-582$		
δ	S	T_P	P	S	T_P	P	S	T_P	P
0.10	1000	10	0.11	1000	10	0.11	1000	30	1.00
0.05	1000	10	0.11	1000	10	0.10	1000	20	1.00
0.01	16000	130	0.11	16000	170	0.11	1000	30	1.00

Table 3. Monte Carlo Inference on SMALL.

Small Sized Sample We first compared our algorithms on SMALL. Table 1 shows the results for k-probability and exact inference. Note that nodes 620 and 582 are close to each other, whereas node 983 is farther apart. Therefore, connections involving the latter are less likely. In this graph, we obtained good approximations using a small fraction of proofs (the queries have 13136, 155695 and 16048 proofs respectively). Our results also show a significant increase in running times as ProbLog explores more paths in the graph, both within the Prolog code and within the BDD code. The BDD running times can vary widely, we may actually have large running times for smaller BDDs, depending on BDD structure. However, using SimpleCUDD instead of the C++ interface used in (Kimmig et al. 2008) typically decreases BDD time by at least one or two orders of magnitude.

Table 2 gives corresponding results for bounded approximation. The algorithm converges quickly, as few proofs are needed and BDDs remain small. Note however that exact inference is competitive for this problem size. Moreover, we observe large speedups compared to the implementation with meta-interpreters used in (De Raedt et al. 2007), where total runtimes to reach $\delta = 0.01$ for these queries were 46234, 206400 and 307966 `msec` respectively. Table 3 shows the performance of the Monte Carlo estimator. On SMALL, Monte Carlo is the fastest approach. Already within the first 1000 samples a good approximation is obtained.

The experiments on SMALL thus confirm that the implementation on top of YAP-Prolog enables efficient probabilistic inference on small sized graphs.

Medium Sized Sample For graph MEDIUM with around 11000 edges, exact inference is no longer feasible. Table 4 again shows results for the k-probability. Comparing these results with the corresponding values from Table 1, we observe that the estimated probability is higher now: this is natural, as the graph has both more

path **k**	983 − 620 T_P	T_B	P	983 − 582 T_P	T_B	P	620 − 582 T_P	T_B	P
1	180	6	0.33	1620	6	0.30	10	6	0.92
2	180	6	0.33	1620	6	0.30	20	6	0.92
4	180	6	0.33	1630	6	0.30	10	6	0.92
8	220	6	0.33	1630	6	0.30	20	6	0.92
16	260	6	0.33	1660	6	0.30	30	6	0.99
32	710	6	0.40	1710	7	0.30	110	6	1.00
64	1540	7	0.42	1910	6	0.30	200	6	1.00
128	1680	6	0.42	2230	6	0.30	240	9	1.00
256	2190	7	0.55	2720	6	0.49	290	196	1.00
512	2650	7	0.64	3730	7	0.53	1310	327	1.00
1024	8100	41	0.70	5080	8	0.56	3070	1357	1.00

Table 4. k-probability on MEDIUM.

memo δ	983 − 620 S	T_P	P	983 − 582 S	T_P	P	620 − 582 S	T_P	P
0.10	1000	1180	0.78	1000	2130	0.76	1000	1640	1.00
0.05	2000	2320	0.77	2000	4230	0.74	1000	1640	1.00
0.01	29000	33220	0.77	29000	61140	0.77	1000	1670	1.00

Table 5. Monte Carlo Inference using `memopath/3` on MEDIUM.

nodes and is more connected, therefore leading to many more possible explanations. This also explains the increase in running times. Approximate inference using bounds only reached loose bounds (with differences > 0.2) on queries involving node `'HGNC_983'`, as upper bound formulae with more than 10 million conjunctions were encountered, which could not be processed.

The Monte Carlo estimator using the standard definition of `path/3` on MEDIUM did not complete the first 1000 samples within one hour. A detailed analysis shows that this is caused by some queries backtracking too heavily. Table 5 therefore reports results using the memorising version `memopath/3`. With this improved definition, Monte Carlo performs well: it obtains a good approximation in a few seconds. Requiring tighter bounds however can increase runtimes significantly.

Biomine Database The Biomine Database covers hundreds of thousands of entities and millions of links. On BIOMINE, we therefore restricted our experiments to the approximations given by k-probability and Monte Carlo. Given the results on MEDIUM, we directly used `memopath/3` for Monte Carlo. Tables 6 and 7 show the results on the large network. We observe that on this large graph, the number of possible paths is tremendous, which implies success probabilities practically equal to 1. Still, we observe that ProbLog's branch-and-bound search to find the best

path **k**	983 − 620 T_P	T_B	P	983 − 582 T_P	T_B	P	620 − 582 T_P	T_B	P
1	5,760	49	0.16	8,910	48	0.11	10	48	0.59
2	5,800	48	0.16	10,340	48	0.17	180	48	0.63
4	6,200	48	0.16	13,640	48	0.28	360	48	0.65
8	7,480	48	0.16	15,550	49	0.38	500	48	0.66
16	11,470	49	0.50	58,050	49	0.53	630	48	0.92
32	15,100	49	0.57	106,300	49	0.56	2,220	167	0.95
64	53,760	84	0.80	146,380	101	0.65	3,690	167	0.95
128	71,560	126	0.88	230,290	354	0.76	7,360	369	0.98
256	138,300	277	0.95	336,410	520	0.85	13,520	1,106	1.00
512	242,210	730	0.98	501,870	2,744	0.88	23,910	3,444	1.00
1024	364,490	10,597	0.99	1,809,680	100,468	0.93	146,890	10,675	1.00

Table 6. k-probability on BIOMINE.

memo δ	983 − 620 S	T_P	P	983 − 582 S	T_P	P	620 − 582 S	T_P	P
0.10	1000	100,700	1.00	1000	1,656,660	1.00	1000	1,696,420	1.00
0.05	1000	100,230	1.00	1000	1,671,880	1.00	1000	1,690,830	1.00
0.01	1000	93,120	1.00	1000	1,710,200	1.00	1000	1,637,320	1.00

Table 7. Monte Carlo Inference using `memopath/3` on BIOMINE.

solutions performs reasonably also on this size of network. However, runtimes for obtaining tight confidence intervals with Monte Carlo explode quickly even with the improved path definition. Given that sampling a program that does not entail the query is extremely unlikely for the setting considered so far, we performed an additional experiment on BIOMINE, where we restrict the number of edges on the path connecting two nodes to a maximum of 2 or 3. Results are reported in Table 8. As none of the resulting queries have more than 50 proofs, exact inference is much faster than Monte Carlo, which needs a higher number of samples to reliably estimate probabilities that are not close to 1.

Altogether, the experiments confirm that our implementation provides efficient inference algorithms for ProbLog that scale to large databases. Furthermore, compared to the original implementation of (De Raedt et al. 2007), we obtain large speedups in both the Prolog and the BDD part, thereby opening new perspectives for applications of ProbLog.

6 Conclusions

ProbLog is a simple but elegant probabilistic logic programming language that allows one to explicitly represent uncertainty by means of probabilistic facts denoting

len	983 − 620			983 − 582			620 − 582		
δ	S	T	P	S	T	P	S	T	P
0.10	1000	21,400	0.04	1000	18,720	0.11	1000	19,150	0.58
0.05	1000	19,770	0.05	1000	20,980	0.10	2000	35,100	0.55
0.01	6000	112,740	0.04	16000	307,520	0.11	40000	764,700	0.55
exact	-	477	0.04	-	456	0.11	-	581	0.55
0.10	1000	106,730	0.14	1000	105,350	0.33	1000	45,400	0.96
0.05	1000	107,920	0.14	2000	198,930	0.34	1000	49,950	0.96
0.01	19000	2,065,030	0.14	37000	3,828,520	0.35	6000	282,400	0.96
exact	-	9,413	0.14	-	9,485	0.35	-	15,806	0.96

Table 8. Monte Carlo inference for different values of δ and exact inference using `lenpath/4` with length at most 2 (top) or 3 (bottom) on BIOMINE. For exact inference, runtimes include both Prolog and BDD time.

independent random variables. The language is a simple and natural extension of the logic programming language Prolog. We presented an efficient implementation of the ProbLog language on top of the YAP-Prolog system that is designed to scale to large sized problems. We showed that ProbLog can be used to obtain both explanation and (approximations of) success probabilities for queries on a large database. To the best of our knowledge, ProbLog is the first example of a probabilistic logic programming system that can execute queries on such large databases. Due to the use of BDDs for addressing the disjoint-sum-problem, the initial implementation of ProbLog used in (De Raedt et al. 2007) already scaled up much better than alternative implementations such as Fuhr's pD engine HySpirit (Fuhr 2000). The tight integration in YAP-Prolog presented here leads to further speedups in runtime of several orders of magnitude.

Although we focused on connectivity queries and Biomine in this work, similar problems are found across many domains; we believe that the techniques presented apply to a wide variety of queries and databases because ProbLog provides a clean separation between background knowledge and what is specific to the engine. As shown for Monte Carlo inference, such an interface can be very useful to improve performance as it allows incremental refinement of background knowledge, e.g., graph procedures. Initial experiments with Dijkstra's algorithm for finding the explanation probability are very promising.

ProbLog is closely related to some alternative formalisms such as PHA and ICL (Poole 1993b; Poole 2000), pD (Fuhr 2000) and PRISM (Sato and Kameya 2001) 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 (Sato 1995) 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 (Poole 2000). 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 (Sato and Kameya 2001). 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 (Riguzzi 2007; Ishihata et al. 2008).

Furthermore, as compared to SLPs (Muggleton 1995), CLP($\mathcal{BN}$) (Santos Costa et al. 2003), and BLPs (Kersting and De Raedt 2008), 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 (Getoor and Taskar 2007) and probabilistic programming (De Raedt et al. 2008). 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 (Vennekens et al. 2004), a recent elegant probabilistic knowledge representation language based on a probabilistic extension of clausal logic, can be compiled into ProbLog (Riguzzi 2007) and it is well-known that SLPs (Muggleton 1995) can be compiled into Sato's PRISM, which is closely related to ProbLog. Further evidence is provided in (De Raedt et al. 2008).

Another, related use of ProbLog is as a vehicle for developing learning and mining algorithms and tools (Kimmig et al. 2007; De Raedt et al. 2008; Gutmann et al. 2008; Kimmig and De Raedt 2009; De Raedt et al. 2009). In the context of probabilistic representations (Getoor and Taskar 2007; De Raedt et al. 2008), 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. (Gutmann et al. 2008) introduces a gradient descent approach to parameter learning for ProbLog that extends the BDD-based methods discussed here. 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. The limiting factor in statistical relational learning and probabilistic logic learning is often the efficiency of inference, as learning requires repeated computation of the probabilities of many queries. Therefore,

improvements on inference in probabilistic programming implementations have an immediate effect on learning. The above compilation approach also raises the interesting and largely open question whether not only inference problems for alternative formalisms can be compiled into ProbLog but whether it is also possible to compile learning problems for these logics into learning problems for ProbLog.

Finally, as ProbLog, unlike PRISM and PHA, deals with the disjoint-sum-problem, it is interesting to study how program transformation and analysis techniques could be used to optimize ProbLog programs, by detecting and taking into account situations where some conjunctions are disjoint. At the same time, we currently investigate how tabling, one of the keys to PRISM's efficiency, can be incorporated in ProbLog (Mantadelis and Janssens 2009; Kimmig et al. 2009).

Acknowledgements

We would like to thank Hannu Toivonen, Bernd Gutmann and Kristian Kersting for their many contributions to ProbLog, the Biomine team for the application, and Theofrastos Mantadelis for the development of SimpleCUDD. This work is partially supported by the GOA project 2008/08 Probabilistic Logic Learning. Angelika Kimmig is supported by the Research Foundation-Flanders (FWO-Vlaanderen). Vítor Santos Costa and Ricardo Rocha are partially supported by the research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/ EIA/66924/2006) and by Fundação para a Ciência e Tecnologia.

References

Bachmair, L., Chen, T., and Ramakrishnan, I. V. 1993. Associative Commutative Discrimination Nets. In *International Joint Conference on Theory and Practice of Software Development*. LNCS, vol. 668. Springer, 61–74.

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

Cussens, J. 2000. Stochastic logic programs: Sampling, inference and applications. In *Uncertainty in Artificial Intelligence*, C. Boutilier and M. Goldszmidt, Eds. Morgan Kaufmann, 115–122.

Dalvi, N. N. and Suciu, D. 2004. Efficient query evaluation on probabilistic databases. In *International Conference on Very Large Databases*, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, Eds. Morgan Kaufmann, 864–875.

Dantsin, E. 1991. Probabilistic logic programs and their semantics. In *Russian Conference on Logic Programming*, A. Voronkov, Ed. LNCS, vol. 592. Springer, 152–164.

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., and Vennekens, J. 2008. Towards Digesting the Alphabet-Soup of Statistical Relational Learning. In *NIPS Workshop on Probabilistic Programming*.

De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S., Eds. 2008. *Probabilistic Inductive Logic Programming - Theory and Applications*. LNCS, vol. 4911.

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

De Raedt, L., Kimmig, A., Gutmann, B., Kersting, K., Santos Costa, V., and Toivonen, H. 2009. Probabilistic inductive querying using ProbLog. Tech. Rep. CW 552, Department of Computer Science, Katholieke Universiteit Leuven.

De Raedt, L., Kimmig, A., and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*, M. M. Veloso, Ed. 2462–2467.

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

Fuhr, N. 2000. Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science (JASIS) 51,* 2, 95–110.

Getoor, L. and Taskar, B., Eds. 2007. *Statistical Relational Learning*. The MIT press.

Graf, P. 1996. *Term Indexing*. LNAI, vol. 1053. Springer.

Gutmann, B., Kimmig, A., Kersting, K., and De Raedt, L. 2008. Parameter learning in probabilistic databases: A least squares approach. In *European Conference on Machine Learning*, W. Daelemans, B. Goethals, and K. Morik, Eds. LNCS, vol. 5211. Springer, 473–488.

Ishihata, M., Kameya, Y., Sato, T., and ichi Minato, S. 2008. Propositionalizing the EM algorithm by BDDs. In *Proceedings of Inductive Logic Programming (ILP 2008), Late Breaking Papers*, F. Železný and N. Lavrač, Eds. Prague, Czech Republic, 44–49.

Kersting, K. and De Raedt, L. 2008. Basic principles of learning bayesian logic programs. In *Probabilistic Inductive Logic Programming*, L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, Eds. LNCS, vol. 4911. Springer, 189–221.

Kimmig, A. and De Raedt, L. 2009. Local query mining in a probabilistic Prolog. In *International Joint Conference on Artificial Intelligence*, C. Boutilier, Ed. 1095–1100.

Kimmig, A., De Raedt, L., and Toivonen, H. 2007. Probabilistic explanation based learning. In *European Conference on Machine Learning*, J. N. Kok, J. Koronacki, R. L. de Mántaras, S. Matwin, D. Mladenic, and A. Skowron, Eds. LNCS, vol. 4701. Springer, 176–187.

Kimmig, A., Gutmann, B., and Santos Costa, V. 2009. Trading memory for answers: Towards tabling ProbLog. In *International Workshop on Statistical Relational Learning*.

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

Lakshmanan, L. V. S., Leone, N., Ross, R. B., and Subrahmanian, V. S. 1997. ProbView: A flexible probabilistic database system. *ACM Transactions on Database Systems 22,* 3, 419–469.

Mantadelis, T. and Janssens, G. 2009. Tabling relevant parts of SLD proofs for ground goals in a probabilistic setting. In *International Colloquium on Implementation of Constraint and LOgic Programming Systems*.

Muggleton, S. 1995. Stochastic logic programs. In *Advances in ILP*, L. De Raedt, Ed.

Poole, D. 1993a. Logic programming, abduction and probability. *New Generation Computing 11*, 377–400.

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

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

Ramakrishnan, I. V., Rao, P., Sagonas, K., Swift, T., and Warren, D. S. 1999.

Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming 38,* 1, 31–54.

Riguzzi, F. 2007. A top down interpreter for LPAD and CP-logic. In *Congress of the Italian Association for Artificial Intelligence (AI*IA)*, R. Basili and M. T. Pazienza, Eds. LNCS, vol. 4733. Springer, 109–120.

Santos Costa, V. 2007. Prolog performance on larger datasets. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007.*, M. Hanus, Ed. LNCS, vol. 4354. Springer, 185–199.

Santos Costa, V., Page, D., Qazi, M., and Cussens, J. 2003. CLP(BN): constraint logic programming for probabilistic knowledge. In *Conference on Uncertainty in Artificial Intelligence*, C. Meek and U. Kjærulff, Eds. Morgan Kaufmann, 517–524.

Santos Costa, V., Sagonas, K., and Lopes, R. 2007. Demand-driven indexing of prolog clauses. In *International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. LNCS, vol. 4670. Springer, 305–409.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, 715–729.

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

Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., and Toivonen, H. 2006. Link discovery in graphs derived from biological databases. In *Data Integration in the Life Sciences*, U. Leser, F. Naumann, and B. A. Eckman, Eds. LNCS, vol. 4075. Springer, 35–49.

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

Vennekens, J., Verbaeten, S., and Bruynooghe, M. 2004. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, B. Demoen and V. Lifschitz, Eds. LNCS, vol. 3132. Springer, 431–445.

Widom, J. 2005. Trio: A system for integrated management of data, accuracy, and lineage. In *Conference on Innovative Data Systems Research*. 262–276.