On the Efficient Execution of ProbLog Programs

Angelika Kimmig[1], Vítor Santos Costa[2], Ricardo Rocha[2], Bart Demoen[1], and Luc De Raedt[1]

[1] Departement Computerwetenschappen, K.U. Leuven
Celestijnenlaan 200A - bus 2402, B-3001 Heverlee, Belgium
{Angelika.Kimmig,Bart.Demoen,Luc.DeRaedt}@cs.kuleuven.be
[2] CRACS & Faculdade de Ciências, Universidade do Porto, Portugal
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
{vsc,ricroc}@dcc.fc.up.pt

Abstract. The past few years have seen a surge of interest in the field of probabilistic logic learning or 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 mutually independent probabilities that they 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.

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 [1], PRISM [2], SLPs [3], ProbView [4], CLP($\mathcal{BN}$) [5], CP-logic [6], Trio [7], probabilistic Datalog (pD) [8], and probabilistic databases [9]. 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 [10], 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 prediction techniques [11]. ProbLog is essentially an extension of Prolog where facts are labeled with the probability that they belong to a

randomly sampled program, and these probabilities are mutually independent. A ProbLog program thus specifies a probability distribution over all its possible non-probabilistic subprograms. 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: ProbLog programs define a distribution semantics [12]. This is a well-known semantics for probabilistic logics that has been (re)defined multiple times in the literature; see for instance the works of [13, 1, 8, 14, 9]. However, even though relying on the same semantics, in order to allow efficient inference, systems such as PRISM [12] and PHA [1] 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.

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) [15]. In addition to an iterative deepening algorithm that computes an approximation along the lines of [16], we further adapt the Monte Carlo approach suggested by [13] and used also by [11] in the context of biological network inference. These two approximation algorithms compute an upper and a lower bound on the success probability. Furthermore, we also contribute an approximation algorithm that computes a lower bound only using the k-most likely proofs.

The key contribution of this paper is the tight integration of these algorithms in the state-of-the-art implementation of the YAP-Prolog system. This integration includes several improvements over the initial implementation used in [10], which enable the use of ProbLog to effectively query Sevon's Biomine network [11] 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, where all probabilities are assumed mutually independent. The definite clauses allow to add arbitrary *background knowledge* (BK).

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

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

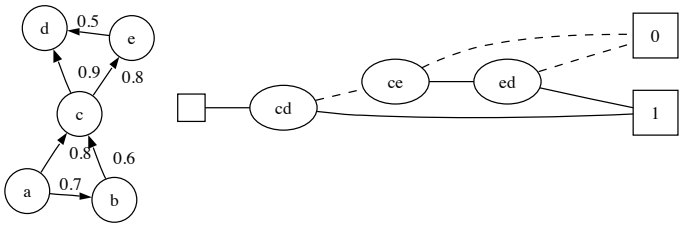

Fig. 1. (a) Example of a probabilistic graph: edge labels indicate the probability that the edge is part of the graph. (b) Binary Decision Diagram encoding the DNF formula $cd \vee (ce \wedge ed)$, corresponding to the two proofs of query *path(c,d)* in the graph. An internal node labeled xy represents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

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:

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

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 probability that a randomly sampled subgraph contains the edge from c to d, or the path from c to d via e (or both of these). Formally, the *success probability* $P_s(q|T)$ of a query q in a ProbLog program T is defined as

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

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.

As a consequence, the probability of a *specific* proof, also called *explanation*, is 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 [17].

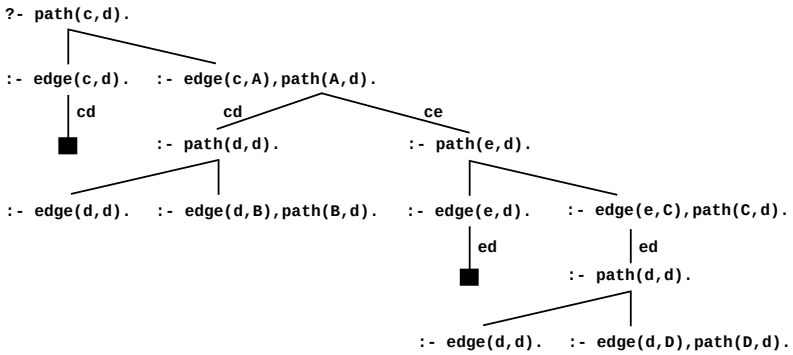

Fig. 2. SLD-tree for query $path(c, d)$.

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 [12]. 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, that is, of the background knowledge BK together with a subprogram $L \subseteq L_T$; for further details we refer to [12]. The distribution semantics has been used widely in the literature; see e.g. [13, 1, 8, 14, 9].

3 Inference in ProbLog

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

3.1 Exact Inference

Calculating the *success probability* of a query using Equation (1) directly is infeasible for all but the tiniest programs; [10] presents a method involving two steps. The first step computes the proofs of the query q in the logical part of the theory T, that is, in $BK \cup L_T$. This step is akin to that performed for pD by [8]. The result will be a DNF formula. The second component employs Binary Decision Diagrams [15] to compute the probability of this formula.

Following Prolog, we employ 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 facts $\{p_{i_1} :: c_{i_1}, \cdots, p_{i_k} :: c_{i_k}\} \subseteq T$. 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 clause $p_i :: c_i \in T$, 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 clauses $\{p_{i_1} :: c_{i_1}, \cdots, p_{i_k} :: c_{i_k}\} \subseteq T$ 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 NP-hard problem, as the different conjunctions need not be independent. Indeed, even under the assumption of independent variables used in ProbLog, the different conjunctions are not mutually exclusive and may overlap. Various algorithms have been developed to tackle this problem, which is known as the disjoint-sum-problem. The pD-engine HySpirit [8] uses the inclusion-exclusion principle, which is reported to scale to about ten proofs. For ICL, which extends PHA by allowing non-disjoint proofs, [14] proposes a symbolic disjoining algorithm, but does not report scalability results. 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; see Section 4 for more details.

Calculating the *explanation probability* P_x, however, can easily be realized using a best-first search, guided by the probability of the current derivation, through standard logic programming techniques based on the SLD-tree [18].

3.2 Approximative Inference

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

Bounded Approximation The first approximation algorithm, similar to the one proposed in [10], uses DNF formulae to obtain both an upper and a lower bound on the probability of a query. It is related to work by [16] 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[1], to obtain DNF formulae for the two bounds. The lower bound formula d_1 represents all proofs with a probability above the current threshold.

[1] Using a probability threshold instead of the depth bound of [10] has been found to speed up convergence, as upper bounds are tighter on initial levels.

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 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 $d = cd \vee (ce \wedge ed)$.

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. [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 (cf. also [1]).

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

Monte Carlo As an alternative approximation technique without BDDs, we propose a Monte Carlo method. In this algorithm, we repeatedly sample a logic program from the ProbLog program and check 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 our previous approximation algorithm, 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. Such an algorithm

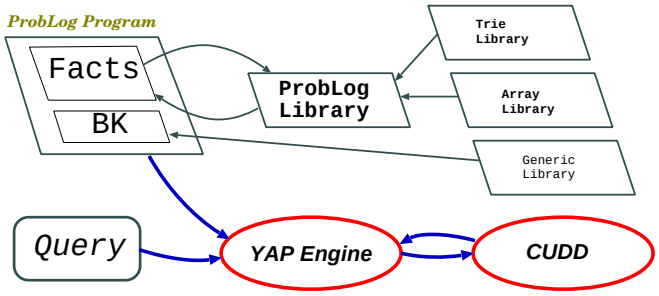

Fig. 3. ProbLog Implementation: A ProbLog program (top-left) requires the ProbLog library which in turn relies on functionality from the tries and array libraries. ProbLog queries (bottom-left) are sent to the YAP engine, and may require calling the BDD library CUDD.

(without the use of confidence intervals) was suggested already by Dantsin [13], although he does not report on an implementation. It was also used in the context of networks (not Prolog programs) by [11].

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. On the top-left corner we show a typical ProbLog program, including ProbLog facts and background knowledge (BK).

The implementation requires ProbLog programs to use the `problog` module. Each program consists of a set of labeled ground facts and of unlabeled *background knowledge*, a generic Prolog program. Labeled ground facts are preprocessed as described below. Notice that the implementation currently only supports labeled *ground facts*.

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

- In k best and bounded approximation, the engine explicitly reasons about probabilities of proofs. The challenge is how to compute the probability of each individual proof, store a large number of proofs, and compute the probability of sets of proofs.
- In Monte Carlo, 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 ProbLog top-level query and proceed as follows:

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

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

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

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

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

would be compiled as:

$$\texttt{edge}(\texttt{A},\texttt{B}) :- \texttt{problog\_edge}(C, A, B, D),\ \texttt{add\_to\_proof}(C, D).$$

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

Thus, the internal representation of each fact contains an identifier, the original arguments, and the logarithm of the probability. The `add_to_proof` procedure updates the data structure representing the current path through the search space and its probability. Compared to the original meta-interpreter based implementation of [10], the main benefit of source-to-source transformation is faster execution time, which in turn improves scalability.

Tries Manipulating proofs is critical in ProbLog. We represent each proof as a list containing the identifier of each different ground probabilistic fact used in the proof, ordered by first use. When manipulating proofs, the key operation is often *insertion*: we would like to add a proof to an existing set of proofs. Some

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

Binary Decision Diagrams To efficiently compute the probability of a DNF formula representing a set of proofs, our implementation represents this formula as a Binary Decision Diagram (BDD) [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 for the existence of a path between c and d in our earlier example.

Our implementation uses the C++ interface of the BDD package CUDD[2] to construct and evaluate BDDs. More precisely, the trie representation of the DNF is translated to C++ code that uses the CUDD primitives for building BDDs. The program is executed via Prolog's shell utility, and results are reported via shared files. We currently work on a tighter integration of BDDs into Prolog.

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. Our translation starts by creating the C++ code for each single variable. 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. Two types of operations are used to combine nodes. First, all the children of a node are combined as a disjunction resulting in a new child node. This child node is then combined with the parent node as a conjunction. 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.

After CUDD has generated the BDD, the probability of a formula is calculated (also in C++) 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. Intermediate results are cached, and the algorithm has a time and space complexity linear in the size of the BDD.

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

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

Generating complete samples and checking for a proof does not scale to large databases, even if proofs are cached in a trie to skip inference on a new sample by checking first whether a subsample is in the proof cache. In fact, already representing and generating the whole sample is a challenge for large databases. Within YAP, the efficient implementation of arrays offers the most compact way of representing large numbers of nodes. On the other hand, quite often proofs are local, i.e. we only need to verify whether facts from a small fragment of the database are in the sample. We take advantage of independence between facts to generate the sample *lazily*: we verify whether a fact is in the sample only when we need it for a proof. Samples are thus represented as a three-valued array: 0 means sampling was not asked yet, 1 means in sample, 2 means not in sample.

5 Experiments

We experiment our implementation of ProbLog in the context of the biological network obtained from the Biomine project [11]. We use 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 are 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 use SMALL for a first comparison of our algorithms on a small scale network where success probabilities can be calculated exactly. Scalability is evaluated using both MEDIUM and the entire BIOMINE network with roughly 1,000,000 nodes and 6,000,000 edges. In all experiments, we query for the probability that two of the gene nodes mentioned above are connected, that is, we use queries such as `path('HGNC_983','HGNC_620',Path)`. We use 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).
```

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).
```

All experiments were performed on Core 2 Duo 3 GHz machines running Linux. All times reported are in `msec` and do not include the time to load the graph into Prolog. The latter takes 32, 192 and 66772 `msec` for SMALL, MEDIUM

path	$983-620$			$983-582$			$620-582$		
k	T_p	T_B	P	T_p	T_B	P	T_p	T_B	P
1	16	-	0.07	4	-	0.03	4	-	0.42
2	0	1613	0.08	0	1686	0.05	4	1511	0.66
4	4	1758	0.10	0	1519	0.06	4	1676	0.86
8	0	1590	0.11	0	1643	0.06	4	1778	0.92
16	4	1744	0.11	4	1536	0.06	4	1719	0.92
32	8	1839	0.11	12	1676	0.07	4	1681	0.96
64	24	1891	0.11	20	1665	0.09	12	1590	0.99
128	52	2054	0.11	32	2130	0.10	48	2286	1.00
256	212	2141	0.11	128	2039	0.10	76	1942	1.00
512	436	13731	0.11	209	2280	0.11	300	2245	1.00
1024	1837	3349	0.11	1372	2195	0.11	581	4080	1.00
exact	641	8343	0.11	5629	2716	0.11	496	2288	1.00

Table 1. k-probability on SMALL.

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 compile and 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 compute k-probability for $k = 1, 2, \ldots, 1024$. Note that no BDDs are used for $k = 1$. In the bounding algorithms, we range the error interval between 10% and 1%. Monte Carlo recalculates confidence intervals after $m = 1000$ samples. We also report the number S of samples used.

Small Sized Sample We first compare 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 obtain 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.

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 [10], 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.

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
0.1	0	5051	3	[0.07,0.12]	0	4994	3	[0.06,0.12]	12	1690	1	[0.99,1.00]
0.05	0	6504	4	[0.07,0.12]	40	10907	6	[0.06,0.11]	12	1751	1	[0.99,1.00]
0.01	8	9897	6	[0.10,0.11]	68	12684	7	[0.10,0.11]	12	1968	1	[0.99,1.00]

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
0.1	1000	19	0.10	1000	21	0.10	1000	63	1.00
0.05	1000	19	0.10	1000	23	0.11	1000	59	1.00
0.01	16000	898	0.11	16000	1418	0.11	1000	59	1.00

Table 3. Monte Carlo Inference on SMALL.

Medium Sized Sample For graph MEDIUM with around 11000 edges we impose a limit of one hour on running times. On this graph, 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 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 very loose bounds within the one hour timelimit, e.g. $[0.33, 0.90]$ for nodes 983 and 620. We found that this is due to the fact that BDDs representing upper bounds get very complex easily.

The Monte Carlo estimator using the standard definition of `path/3` on MEDIUM did not converge within the time limit. 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 restrict our experiments to the approximations given by k-probability and Monte Carlo. Given the results on MEDIUM, we directly use `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 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.

Altogether, the experiments confirm that our implementation provides good approximations of ProbLog probabilities and is able to deal with large graphs.

path	$983-620$			$983-582$			$620-582$		
k	T_p	T_B	P	T_p	T_B	P	T_p	T_B	P
1	208	-	0.07	737	-	0.03	45	-	0.42
2	172	1591	0.11	725	1560	0.03	44	1599	0.47
4	200	1681	0.16	757	1738	0.05	60	1464	0.72
8	217	1691	0.25	744	1538	0.06	80	1778	0.92
16	284	1756	0.33	725	1508	0.10	100	1825	0.99
32	628	1855	0.38	753	1570	0.15	144	1578	1.00
64	717	1653	0.41	809	1684	0.23	200	1801	1.00
128	749	1715	0.42	933	1890	0.30	296	1734	1.00
256	849	1600	0.55	1044	1513	0.49	405	1904	1.00
512	2352	1696	0.64	2880	1598	0.53	576	2496	1.00
1024	6208	1849	0.70	5032	1728	0.56	2549	52250	1.00

Table 4. k-probability on MEDIUM.

memo	$983-620$			$983-582$			$620-582$		
δ	S	T_p	P	S	T_p	P	S	T_p	P
0.1	1000	1319	0.77	1000	2364	0.76	1000	1878	1.00
0.05	2000	2682	0.76	2000	4766	0.76	1000	1805	1.00
0.01	29000	39687	0.76	29000	70183	0.77	1000	1970	1.00

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

6 Conclusions

ProbLog is an elegant probabilistic logic language that addresses the problem of representing uncertain knowledge by explicitely encoding uncertainty about the truth of facts. The language naturally extends Logic Programming languages such as Prolog. We present an implementation of the ProbLog language on top of the YAP Prolog system that is designed to scale for large sized problems. We show that ProbLog can indeed be used to obtain both explanation and (approximations of) success probabilities for queries on a large database. To the best of our knowledge, this is the first example of a probabilistic logic programming system that can execute queries on such large databases. Furthermore, compared to the initial implementation of ProbLog used in [10], the tight integration in YAP-Prolog leads to speedups in runtime of several orders of magnitude.

Although we focussed on connectivity queries and Biomine in this work, similar problems are found across many domains; we believe that the techniques presented so far apply to a variety of queries and databases. This is largely possible 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 incrementally refining background knowledge, e.g. graph procedures. Initial experiments with Dijkstra's algorithm for finding the explanation probability are very promising.

Compared to alternative formalisms such as PHA [1], PRISM [2], SLPs [3], CLP($\mathcal{BN}$) [5], and CP-logic [6], ProbLog is an extremely simple probabilistic

path	$983-620$			$983-582$			$620-582$		
k	T_p	T_B	P	T_p	T_B	P	T_p	T_B	P
1	5,445	-	0.09	1,248	-	0.11	10,189	-	0.59
2	5,472	1,611	0.12	1,313	1,563	0.17	2,288	1,570	0.63
4	5,989	1,735	0.13	13,729	1,986	0.28	600	1,545	0.65
8	7,016	1,656	0.16	19,885	1,878	0.38	929	1,792	0.66
16	10,012	1,980	0.50	30,338	1,816	0.53	1,557	1,644	0.92
32	14,857	1,872	0.57	35,134	1,657	0.56	2,484	1,922	0.95
64	19,770	1,642	0.80	36,995	1,737	0.65	4,425	1,925	0.95
128	23,165	1,892	0.88	163,242	1,835	0.76	8,472	2,117	0.98
256	35,395	2,149	0.95	292,054	1,463	0.85	16,390	4,935	1.00
512	170,438	3,148	0.98	489,254	15,410	0.88	29,525	7,693	1.00
1024	346,742	609,700	0.99	767,968	97,818	0.93	49,952	102,366	1.00

Table 6. k-probability on BIOMINE.

memo	$983-620$			$983-582$			$582-620$		
δ	S	T_p	P	S	T_p	P	S	T_p	P
0.1	1000	2,714,781	1.00	1000	4,887,260	0.97	1000	4,709,921	0.99
0.05	1000	2,807,927	1.00	1000	4,769,216	0.98	1000	4,823,262	0.99
0.01	1000	2,686,881	1.00	4000	19,187,318	0.98	2000	9,406,026	0.99

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

logic. Yet, it has proven to be natural and convenient for modeling biological networks and as a vehicle for developing mining and machine learning approaches [17, 20, 19, 21]. The efficiency of the probabilistic logic implementation is the most important factor determining the success and the performance of the learning approaches. Therefore, we expect the efficiency gains to open new possibilities for learning, and to increase the use of probabilistic logics in practical applications. Another possible use of a simple probabilistic logic, such as ProbLog, is as a target language in which other, possibly more complex, formalisms can be compiled. For instance, [22] shows how CP-logic [6] can be compiled into ProbLog, and SLPs [3] can be compiled in Sato's PRISM, which is closely related to 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.

Acknowledgements We would like to thank Hannu Toivonen for his many contributions to ProbLog and the Biomine team for the application. 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

1. Poole, D.: Probabilistic Horn abduction and Bayesian networks. Artificial Intelligence **64** (1993) 81–129
2. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. J. Artif. Intell. Res. (JAIR) **15** (2001) 391–454
3. Muggleton, S.: Stochastic logic programs. In De Raedt, L., ed.: ILP. (1995)
4. Lakshmanan, L.V.S., Leone, N., Ross, R.B., Subrahmanian, V.S.: ProbView: A flexible probabilistic database system. ACM Trans. Database Syst. **22**(3) (1997) 419–469
5. Santos Costa, V., Page, D., Qazi, M., Cussens, J.: CLP(BN): constraint logic programming for probabilistic knowledge. In Meek, C., Kjærulff, U., eds.: UAI, Morgan Kaufmann (2003) 517–524
6. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In Demoen, B., Lifschitz, V., eds.: ICLP. Volume 3132 of LNCS., Springer (2004) 431–445
7. Widom, J.: Trio: A system for integrated management of data, accuracy, and lineage. In: CIDR. (2005) 262–276
8. Fuhr, N.: Probabilistic Datalog: Implementing logical information retrieval for advanced applications. JASIS **51**(2) (2000) 95–110
9. Dalvi, N.N., Suciu, D.: Efficient query evaluation on probabilistic databases. In Nascimento, M.A., Özsu, M.T., Kossmann, D., Miller, R.J., Blakeley, J.A., Schiefer, K.B., eds.: VLDB, Morgan Kaufmann (2004) 864–875
10. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In Veloso, M.M., ed.: IJCAI. (2007) 2462–2467
11. Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., Toivonen, H.: Link discovery in graphs derived from biological databases. In Leser, U., Naumann, F., Eckman, B.A., eds.: DILS. Volume 4075 of LNCS., Springer (2006) 35–49
12. Sato, T.: A statistical learning method for logic programs with distribution semantics. In Sterling, L., ed.: ICLP, MIT Press (1995) 715–729
13. Dantsin, E.: Probabilistic logic programs and their semantics. In Voronkov, A., ed.: RCLP. Volume 592 of LNCS., Springer (1991) 152–164
14. Poole, D.: Abducing through negation as failure: stable models within the independent choice logic. J. Log. Program. **44**(1-3) (2000) 5–35
15. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8) (1986) 677–691
16. Poole, D.: Logic programming, abduction and probability. New Generation Computing **11** (1993) 377–400
17. Kimmig, A., De Raedt, L., Toivonen, H.: Probabilistic explanation based learning. In Kok, J.N., Koronacki, J., de Mántaras, R.L., Matwin, S., Mladenic, D., Skowron, A., eds.: ECML. Volume 4701 of LNCS., Springer (2007) 176–187
18. Lloyd, J.W.: Foundations of Logic Programming. 2. edn. Springer, Berlin (1989)
19. Gutmann, B., Kimmig, A., Kersting, K., De Raedt, L.: Parameter learning in probabilistic databases: A least squares approach. In Daelemans, W., Goethals, B., Morik, K., eds.: ECML/PKDD (1). Volume 5211 of LNCS., Springer (2008) 473–488
20. De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic Prolog programs. Machine Learning **70**(2-3) (2008) 151–168
21. Kimmig, A., De Raedt, L.: Probabilistic local pattern mining. In: ILP. (2008)
22. Riguzzi, F.: A top down interpreter for LPAD and CP-logic. In Basili, R., Pazienza, M.T., eds.: AI*IA. Volume 4733 of LNCS., Springer (2007) 109–120