# ProbLog: A Probabilistic Prolog and its Application in Link Discovery

**Luc De Raedt**[*], **Angelika Kimmig and Hannu Toivonen**[†]
Machine Learning Lab, Albert-Ludwigs-University Freiburg, Germany

## Abstract

We introduce ProbLog, a probabilistic extension of Prolog. A ProbLog program defines a distribution over logic programs by specifying for each clause the probability that it belongs to a randomly sampled program, and these probabilities are mutually independent. The semantics of ProbLog is then defined by the success probability of a query, which corresponds to the probability that the query succeeds in a randomly sampled program. The key contribution of this paper is the introduction of an effective solver for computing success probabilities. It essentially combines SLD-resolution with methods for computing the probability of Boolean formulae. Our implementation further employs an approximation algorithm that combines iterative deepening with binary decision diagrams. We report on experiments in the context of discovering links in real biological networks, a demonstration of the practical usefulness of the approach.

## 1 Introduction

Over the past two decades an increasing number of probabilistic logics has been developed. The most prominent examples include PHA [Poole, 1993], PRISM [Sato and Kameya, 2001], SLPs [Muggleton, 1996], MLNs [Richardson and Domingos, 2006] and probabilistic Datalog (pD) [Fuhr, 2000]. These frameworks attach probabilities to logical formulae, most often definite clauses. In addition, they often impose various constraints on probabilities. For instance, in SLPs, clauses defining the same predicate are assumed to be mutually exclusive; PRISM and PHA only attach probabilities to factual information, and again constraints are imposed that essentially exclude the possibility that certain combinations of facts are simultaneously true. These assumptions facilitate the computation of the probability of queries and simplify the learning algorithms for such representations. One approach, the pD formalism of [Fuhr, 2000] that is intimately related to ProbLog, does not impose such restrictions but its inference engine has severe limitations. At the same time, it seems that there are – despite a great potential – still only few real-life applications of these probabilistic logics. The reasons for this might well be that the assumptions are too strong and sometimes hard to manage by the user, or that the solvers are often too slow or too limited.

We introduce ProbLog which is – in a sense – the simplest probabilistic extension of Prolog one can design. ProbLog is essentially Prolog where all clauses are labeled with the probability that they are true, and – similar as pD but unlike the other approaches mentioned – these probabilities are mutually independent. ProbLog has been motivated by the real-life application of mining large biological networks where edges are labeled with probabilities. Such networks of biological concepts (genes, proteins, phenotypes, etc.) can be extracted from large public databases, and probabilistic links between concepts can be obtained by various prediction techniques. In this and many other applications, probabilistic links are mutually independent and can easily be described in ProbLog.

A ProbLog program specifies a probability distribution over all possible non-probabilistic subprograms of the ProbLog program. The success probability of a query is then defined simply as the probability that it succeeds in these subprograms. The semantics of ProbLog is not really new, it closely corresponds to that of pD [Fuhr, 2000] and of [Dantsin, 1991]. The key contribution of this paper, however, is the introduction of an effective inference procedure for this semantics, and its application to a real-life link discovery task.

The success probability of a ProbLog query can be computed as the probability of a boolean monotone DNF (Disjunctive Normal Form) formula of binary random variables. Unfortunately, the latter problem is NP-hard [Valiant, 1979]. Since pD employs a naive approach based on inclusion-exclusion for computing the probabilities of these formulae, the evaluation of about 10 or more conjuncts is infeasible in the implementation of pD according to [Fuhr, 2000]. In contrast, ProbLog's approximation algorithm is able to deal with formulae containing up to 100000 formulae. The ProbLog solver has been motivated by and employs recent advances in binary decision diagrams (BDDs) for dealing with Boolean functions. At the same time, it employs an approximation algorithm for computing the success probability along the lines of [Poole, 1992]. Using this algorithm we report on experiments in biological networks that demonstrate the practical

[*] Recently moved to the Katholieke Universiteit Leuven.

[†] Also at University of Helsinki.

usefulness of the approach. Obviously, it is straightforward to transfer ProbLog to other link and network mining domains.

The paper is structured as follows. We describe a motivating application in Section 2. In Section 3, we introduce ProbLog and its semantics. We shall assume some familiarity with the Prolog programming language, see for instance [Flach, 1994] for an introduction. In Section 4, we show how ProbLog queries can be represented by monotone DNF formulae and then computed with BDDs. Section 5 gives an approximation algorithm for ProbLog queries, and experiments on real biological data are reported in Section 6. Finally, in Sections 7 and 8, we discuss related work and conclude.

## 2 Example: ProbLog for biological graphs

As a motivating application for ProbLog, consider link mining in large networks of biological concepts. Enormous amounts of molecular biological data are available from public sources, such as Ensembl[1], NCBI Entrez[2], and many others. They contain information about various types of objects, such as genes, proteins, tissues, organisms, biological processes, and molecular functions. Information about their known or predicted relationships is also available, e.g., that gene A of organism B codes for protein C, which is expressed in tissue D, or that genes E and F are likely to be related since they co-occur often in scientific articles. Mining this data has been identified as an important and challenging task (see, e.g., [Perez-Iratxeta *et al.*, 2002]).

Such a collection of interlinked heterogeneous biological data can be conveniently seen as a weighted graph or network of biological concepts, where the weight of an edge corresponds to the probability that the corresponding nodes are related [Sevon *et al.*, 2006]. Probabilities of edges can be obtained from methods that predict their existence based on, e.g., co-occurrence frequencies or sequence similarities. A ProbLog representation of such a graph could in the most simple case consist of probabilistic edge/2 facts though finer grained representations using relations such as codes/2, expresses/2 would also be possible.

A typical query that a life scientist may want to ask from such a database of biological concepts is whether a given gene is connected to a given disease. In a probabilistic graph, the importance of the connection can be measured as the probability that a path exists between the two given nodes, assuming that each edge is true with the specified probability, and that edges are mutually independent [Sevon *et al.*, 2006]. Such queries are easily expressed in logic by defining the (non-probabilistic) predicate path(N1,N2) in the usual way. Now the query ?- path('gene_620', 'disease_alzheimer') would look for paths between the given nodes. Since edges were assumed probabilistic, this query has a certain success probability. This probability, defined below, directly corresponds to the probability that a path exists between the nodes, known as the two-terminal network reliability problem.

Obviously, logic – and ProbLog – can easily be used to express much more complex possible relations. For instance, two proteins that both interact with a third one possibly also interact with each other if they are all expressed in the same tissue. Or, two genes are possibly functionally related if they have closely related annotations from Gene Ontology[3].

## 3 ProbLog

A ProbLog program consists – as Prolog – of a set of definite clauses. However, in ProbLog every clause $c_i$ is labeled with the probability $p_i$.

**Example 1** *As an example, consider:*

```
1.0: likes(X,Y):- friendof(X,Y).
0.8: likes(X,Y):- friendof(X,Z), likes(Z,Y).
0.5: friendof(john,mary).
0.5: friendof(mary,pedro).
0.5: friendof(mary,tom).
0.5: friendof(pedro,tom).
```

Even though we shall focus on the "pure" subset of Prolog, i.e. definite clause logic, ProbLog also allows one to use most built-in predicates in Prolog by assuming that all clauses defining built-in predicates have label 1.

A ProbLog program $T = \{p_1 : c_1, \cdots, p_n : c_n\}$ now defines a probability distribution over logic programs $L \subseteq L_T = \{c_1, \cdots, c_n\}$ in the following way:

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

Unlike in Prolog, where one is typically interested in determining whether a query succeeds or fails, in ProbLog we are interested in computing the probability that it succeeds. The *success probability* $P(q|T)$ of a query $q$ in a ProbLog program $T$ is defined by

$$\begin{aligned} P(q|L) &= \begin{cases} 1 & \exists\theta : L \models q\theta \\ 0 & otherwise \end{cases} && (2) \\ P(q, L|T) &= P(q|L) \cdot P(L|T) && (3) \\ P(q|T) &= \sum_{M \subseteq L_T} P(q, M|T) && (4) \end{aligned}$$

In other words, the success probability of query $q$ corresponds to the probability that the query $q$ has *a* proof, given the distribution over logic programs.

## 4 Computing success probabilities

Given a ProbLog program $T = \{p_1 : c_1, \cdots, p_n : c_n\}$ and a query $q$, the trivial way of computing the success probability $P(q|T)$ proceeds by enumerating all possible logic programs $M \subseteq L_T$ (cf. Equation 4). Clearly this is infeasible for all but the tiniest programs.

We develop a method involving two components. The first is concerned with the computation of the proofs of the query $q$ in the logical part of the theory $T$, that is in $L_T$. Its result will be a monotone DNF formula. The second component computes the probability of this formula.

[1] www.ensembl.org

[2] www.ncbi.nlm.nih.gov/Entrez/

[3] www.geneontology.org

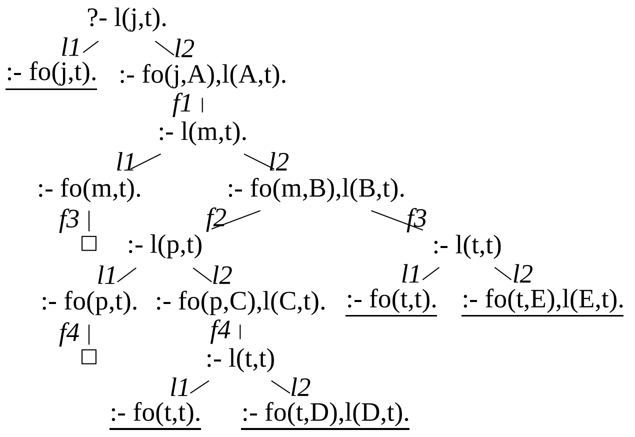


Figure 1: The SLD-tree for the goal likes(john,tom), using obvious abbreviations.

### 4.1 ProbLog queries as DNF formulae

To study the set of logic programs where a given query can be proved we consider the logical part $L_T$ of the theory $T$. We now show how this set of logic programs can be represented by a DNF formula.

We employ SLD-resolution, from which the execution mechanism of Prolog is derived. As an example, the SLD-tree for the query ?- likes(john,tom). is depicted in Figure 1. The paths from the root to individual leaves of the SLD-tree represent either a successful or a failed proof. From a path that ends in the empty goal denoted by □, one can construct an answer substitution $\theta$ making the original goal true. The other proofs end at an (underlined) goal that fails.

The standard SLD-resolution computes the SLD-tree in a top-down fashion. It initializes the root of the SLD-tree with the query $?-l_1,\cdots,l_n$ to be proven and then recursively generates a subgoal of the form $? - b_1\theta,\cdots,b_m\theta,l_2\theta,\cdots,l_n\theta$ for each clause $h : -b_1,\cdots,b_m$ in the logic program for which the most general unifier of $h$ and $l_1$ is the substitution $\theta$. See e.g. [Flach, 1994] for a more extensive treatment.

Each succesful proof in the SLD-tree has a set of clauses $\{p_1 : d_1,\cdots,p_k : d_k\} \subseteq T$ employed in that proof. These clauses are necessary for the proof, and the proof is independent of other clauses in $T$. As a consequence, the probability that this proof succeeds is $\prod_i p_i$. (In other words, the sum of probabilities of programs containing these clauses is $\prod_i p_i$.)

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; i.e., $b_i$ has probability $p_i$ of being true. The probability of a particular proof involving clauses $\{p_1 : d_1,\cdots,p_k : d_k\} \subseteq T$ is then the probability of the conjunctive formula $b_1\wedge\cdots\wedge b_k$. Since a goal can have multiple proofs, the probability that goal $q$ succeeds equals the probability that the disjunction of these conjunctions is true. More formally, this yields:

$$P(q|T) = P\left(\bigvee_{b\in pr(q)} \bigwedge_{b_i\in cl(b)} b_i\right) \tag{5}$$

where we use the convention that $pr(q)$ denotes the set of proofs of the goal $q$ and $cl(b)$ denotes the set of Boolean variables (clauses) used in the proof $b$. 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, which is monotone as all variables appear positively.

**Example 2** *Continuing our running example, and using* $l_1, l_2$ *as names (and Boolean variables) for the two clauses defining* likes *and* $f_1,\cdots,f_4$ *for* friendof,

$$\begin{aligned} &P(\mathsf{likes}(\mathsf{john},\mathsf{tom})|T)\\ &\quad = P((l_1\wedge l_2\wedge f_1\wedge f_2\wedge f_4)\vee(l_1\wedge l_2\wedge f_1\wedge f_3)). \end{aligned} \tag{6}$$

*Since* $P(l_1) = 1$, *this is equal to*

$$P((l_2\wedge f_1\wedge f_2\wedge f_4)\vee(l_2\wedge f_1\wedge f_3)). \tag{7}$$

### 4.2 Computing the probability of DNF formulae

Computing the probability of DNF formulae is an NP-hard problem even if all variables are independent, as they are in our case. There are several algorithms for transforming a disjunction of conjunctions into mutually disjoint conjunctions, for which the probability is obtained simply as a sum. In the literature, this is sometimes referred to as the problem of transforming sum-of-products into sum-of-disjoint-products. One basic approach relies on the inclusion-exclusion principle from set theory. It requires the computation of conjunctive probabilities of all sets of conjunctions appearing in the DNF formula. This is clearly intractable in general. More advanced techniques expand the conjunctions also with negated subformulae, in order to disjoin each of them from all previous ones [Luo and Trivedi, 1998]. However, these algorithms seem to be limited to a few dozens of variables and a few hundreds of sums. Motivated by the advances made in the manipulation and representation of Boolean formulae using binary decision diagrams (BDDs) since their introduction by [Bryant, 1986], we employ this class of techniques.

A BDD is an efficient graphical representation of a Boolean function over a set of variables. A BDD representing the formula in Equation 7 is shown in Figure 2. Given a fixed variable ordering, a Boolean function $f$ can be represented as a full Boolean decision tree where each node on the $i$th level is labeled with the $i$th variable and has two children called low and high. Each path from the root to some leaf stands for one complete variable assignment. If variable $x$ is assigned 0 (1), the branch to the low (high) child is taken. Each leaf is labeled by the outcome of $f$ given the variable assignment represented by the corresponding path. 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 iff the subgraphs rooted at its children are isomorphic. In Figure 2, dashed edges indicate 0's and lead to low children, solid ones indicate 1's and lead to high children.

In practice, the chosen variable ordering determines the extent to which substructures can be shared in the BDD and thus has an enormous influence on the size and complexity of the resulting BDD. State-of-the-art BDD implementations therefore employ heuristics to automatically reorder the variables during BDD construction, which help to control the combinatorial explosion.

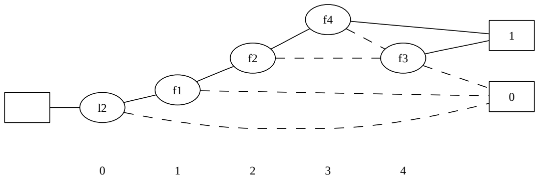


Figure 2: A BDD representing the Boolean in equation 7.

Given a BDD, it is easy to compute the probability of the corresponding Boolean function by traversing the BDD from the root node to a leaf. At each inner node, probabilities from both children are calculated recursively and combined afterwards as it is done in the following procedure. In practice, memorization of intermediate results is used to avoid the recomputation at nodes that are shared between multiple paths.

```
Probability(input: 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 $p_n \cdot prob(h) + (1 - p_n) \cdot prob(l)$
```

As we shall see in Section 6, the resulting algorithm can be applied to ProbLog programs containing hundreds of clauses (Boolean variables in the BDD) and tens of thousands of proofs (products of variables). In our implementation of ProbLog, we store the conjunctions corresponding to proofs in a prefix-tree for reasons of efficiency.

One interesting further use of the already constructed BDD for a given query $q$ is that it becomes very efficient to answer conditional probability questions of the form $P(q|T, b'_1 \wedge \cdots \wedge b'_k)$ where the $b'_i$s are possibly negated Booleans representing the truth-values of clauses. To compute the answer, one only needs to reset the probabilities of the corresponding nodes to 1 or 0, and call the procedure Probability.

## 5 An approximation algorithm

In this section we introduce an algorithm for approximating the success probability of queries in ProbLog. This is useful for large ProbLog programs, since the size of the monotone DNF formula can explode.

A first obvious observation leading towards a faster algorithm is that – as in Example 2 – one can remove all Boolean variables that correspond to clauses with probability 1.

A second and more important observation allows one to eliminate complete proofs from the monotone DNF formula. During the computation of the SLD-tree, which proceeds depth-first, from left to right in Prolog, we keep track of a DNF formula $d$ that represents the DNF of the already computed proofs. If further proofs are encountered whose conjunction $b_1 \wedge \ldots \wedge b_n$ is logically entailed by $d$, i.e. $d \models b_1 \wedge \ldots \wedge b_n$, then these proofs can be removed from further consideration. Because we work with monotone DNF formulae, this condition can be checked efficiently by verifying that $b_1 \wedge \ldots \wedge b_n$ is not entailed by any of the conjuncts $d_1 \wedge \cdots \wedge d_k$ in $d$. This corresponds to testing whether $\{d_1, \cdots, d_k\} \subseteq \{b_1, \cdots, b_n\}$, a kind of subsumption.

This observation motivates the use of iterative deepening instead of depth-first search to compute the SLD-tree and the corresponding DNF formula. In this way, it becomes more likely that later proofs will be subsumed by already computed ones. Iterative deepening essentially proceeds as depth-first search but does not expand goals in the SLD-tree whose depth exceeds a threshold. It then iteratively increases this depth-bound. Iterative deepening also avoids getting trapped into possibly infinite paths in the SLD-tree. Instead of using a depth-bound one can also employ a probability bound in ProbLog, resulting in a best-first kind of search.

A final observation, leading to approximations of success probabilities, is that an incomplete SLD-tree can be used (during iterative deepening) to derive an upper and a lower bound on the success probability of the query. This observation and the corresponding algorithm are related to work by [Poole, 1992] in the context of PHA, but adapted towards ProbLog. For ProbLog, the bounds can be obtained by constructing two DNF formulae from the incomplete SLD-tree. The first DNF formula $d_1$ encodes the successful proofs already occurring in the tree. The second DNF formula $d_2$ encodes the successful proofs already occurring in the tree as well as the proofs that have been cut off. We then have that

$$P(d_1) \leq P(q|T) \leq P(d_2). \tag{8}$$

This directly follows from the fact that $d_1 \models d \models d_2$ where $d$ is the Boolean DNF formula corresponding to the full SLD-tree of the query.

**Example 3** *Consider the SLD-tree in Figure 1 only till depth 4. In this case, $d_1$ encodes the left success path while $d_2$ additionally encodes the paths up to* likes(pedro,tom) *and* likes(tom,tom), *i.e.*

$$\begin{aligned} d_1 &= (l_1 \wedge l_2 \wedge f_1 \wedge f_3) \\ d_2 &= (l_1 \wedge l_2 \wedge f_1 \wedge f_3) \vee (l_2 \wedge f_1 \wedge f_2) \vee (l_2 \wedge f_1 \wedge f_3) \end{aligned}$$

*The formula for the full SLD-tree is given in equation 6.*

Better approximations will be obtained by stretching the bound on the depth of SLD-trees. The following algorithm approximates the success probability of a ProbLog query $q$ to $\epsilon$.

```
Approximate(query $q$; program $T$; bound $\epsilon$)
depthbound := 1;
$d_1 := false$
repeat
        $d_2 := d_1$
        call Iterate($q$, $true$, depthbound, $d_1$, $d_2$)
        $p_1$ := call Probability($d_1$)
        $p_2$ := call Probability($d_2$)
        increment depthbound
    until $p_2 - p_1 \leq \epsilon$
return $p_1$, $p_2$
```

```
Iterate(query $q$; conjunction $c$; depthbound $d$; DNF $d_1$, $d_2$)
if $q$ is empty and $d_1 \not\models c$
    then $d_1 := d_1 \vee c$
         $d_2 := d_2 \vee c$
elseif $d < 0$ and $d_1 \not\models c$
    then $d_2 := d_2 \vee c$
    else let $q$ be $l, q_1, \cdots, q_n$
         select rule $h : -b_1, \cdots, b_m$
         such that $mgu(h, l) = \theta$ and the rule is
         represented by Boolean variable $b$
         $d := d - 1$
         call Iterate($b_1\theta, \cdots, b_m\theta, q_1\theta, \cdots, q_n\theta$;$c \wedge b$; $d$)
```

**Theorem 1** *Upon termination, the procedure Approximate($q$, $T$, $\epsilon$) returns values $p_1$ and $p_2$ such that $p_1 \leq P(q|T) \leq p_2$ and $p_2 - p_1 \leq \epsilon$.*

The algorithm may not always terminate for all inputs, but this is hardly avoidable given that Prolog does not always terminate (even when using iterative deepening).

**Example 4** *Consider calling Approximate with the query $? - p$; the program $1.0\ p : -p$; and the bound $\epsilon = 0.5$. The procedure will never terminate (because the SLD-tree is infinite and repetetive).*

## 6 Experiments

We implemented the approximation algorithm in Prolog (Yap-5.1.0) and used CUDD[4] for BDD operations. Motivated by the fact that even for simple connection queries, the number of proofs quickly explodes in our biological networks, we primarily try to answer the following question:

**Q** How well does our approximation algorithm scale?

As our test graph $G$, we used a real biological graph around four random Alzheimer genes (HGNC ids 620, 582, 983, and 8744), with 11530 edges and 5220 nodes. The graph was extracted from NCBI and some other databases by taking the union of subgraphs of radius 3 from the four genes and producing weights as described in [Sevon *et al.*, 2006]. As a test case, we focused on evaluating the connection between two of the genes (620, 983). For scaling experiments, we randomly subsampled edges from $G$ to obtain subgraphs $G_1 \subset G_2 \subset \ldots$ of sizes $200, 400, ..., 5000$ edges. Each $G_i$ contains the two genes and consists of one connected component. Average degree of nodes ranges in $G_i$s approximately from 2 to 3. Subsampling was repeated 10 times.

The ProbLog approximation algorithm was then run on the data sets with the goal of obtaining $\epsilon = 0.01$ approximations. As the minimal length of paths in between those two nodes is 4 in the original graph, we initialized the iterative deepening threshold (number of used clauses) to 4.

In this setting, the connection query could be solved to $\epsilon = 0.01$ for graphs with up to 1400 to 4600 edges, depending on the random sample. As an illustrative example, Figure 3(a) shows the convergence of the bounds for one graph with 1800 edges. Figure 3(b) shows the average width of the probability interval for the 10 random subgraphs of 1400 edges. In most cases, results are quite accurate already after levels 6–8. The maximal level used over all $G_i$ was 14 clauses, corresponding to at most eleven search levels. Running times for one subgraph typically range from seconds for small graphs up to four hours for the largest successful runs. Figure 3(c) shows running times for the 10 random graphs of 1400 edges on a 3.2 GHz machine. In our tests, BDD construction typically became infeasible if the number of conjunctions used for the upper bound exceeded 100000. In all those cases, the width of the last calculated probability interval was already smaller than 0.03. Together all these results indicate that the approach can give good bounds for quite large problems, which answers question Q positively.

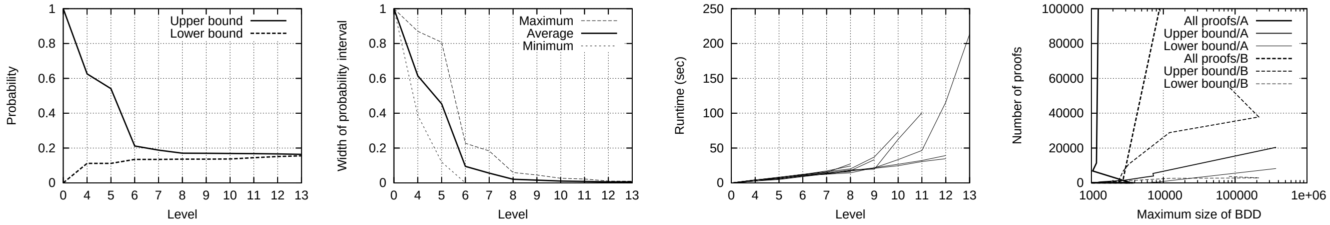
A partial explanation of the good scalability is given in Figure 3(d). It shows the total number of proofs as well as the numbers of proofs needed for the bounds ($\epsilon = 0.01$) for two illustrative random sequences of subgraphs. The numbers are given as a function of the maximal size of the BDD, used here as an approximation of the problem complexity. As can be seen, the number of proofs explodes badly, while the numbers of proofs needed for the bounds scale up much nicer.

## 7 Related Work

The ProbLog semantics is not really new, it closely corresponds to the semantics proposed by [Dantsin, 1991] and of pD [Fuhr, 2000] even though there are some subtle differences. Whereas ProbLog and the approach by Dantsin employ Prolog, pD focusses on Datalog and hence, does not allow for functors. Furthermore, the work on pD comes more from a database perspective and has also devoted a lot of attention to negation. Whereas Dantsin does not report on an implementation, pD has been implemented in the HySpirit system, which computes success probabilities in two steps. The first step employs magic sets to compute the answers to the Datalog component of the query, and in a second step employs the inclusion-exclusion principle to compute the probability of the resulting DNF expressions. This second step makes the approach severely limited. Indeed, [Fuhr, 2000] states: "Practical experimentation with Hyspirit has shown that the evaluation of about 10 or more conjuncts is not feasible." In contrast, using ProbLog's approximation algorithm one can deal with up to 100000 conjuncts as shown in the experiments. This is also needed in order to cope with realistic applications in link mining.

Second, the ProbLog semantics extends also the distributional semantics by [Sato and Kameya, 2001] and the proposal by [Dantsin, 1991] in that it also allows for attaching probabilities to clauses, not just to ground facts. Although this ability has to be exercised with caution (as the truth of two clauses, or non-ground facts, need not be independent, e.g. when one clause subsumes the other one), it does provide new abilities, as illustrated in the likes/2 example. More importantly, systems such as PRISM [Sato and Kameya, 2001] and also PHA [Poole, 1993], avoid the combinatorial problem of the second step by imposing various constraints on the allowed programs which basically guarantee that the formula describing all proofs of a query is a sum-of-*disjoint*-products. Whereas such conditions may be natural for some types of

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



(a) (b) (c) (d)

Figure 3: (a) Convergence of bounds for one graph with 1800 edges, as a function of the search level. (b) Convergence of the probability interval for 10 test graphs with 1400 edges. (c) Running times for 10 test graphs with 1400 edges. (d) Total number of proofs and numbers of proofs needed for bounds ($\epsilon = 0.01$) for two illustrative test graph sequences ('A' and 'B').

applications, such as those involving context-free grammars, they hinder the application of these frameworks to some other applications. For instance, the characteristics of the biological network mining task are incompatible with the restrictions imposed by systems such as PRISM and PHA.

Finally, pure SLPs [Muggleton, 1996] differ in the way the clauses are labeled and, hence, possesses a quite different semantics. The labels in SLPs correspond – as in probabilistic context-free grammars – to the probability that the clause is used to continue a proof if some atom having the same predicate as the head of the clause has to be proven.

## 8 Conclusions

We have defined ProbLog, a probabilistic version of Prolog, and have shown that the success probability of ProbLog queries can be computed using a reduction to a monotone DNF formula, whose probability can then be determined using BDDs, and we have also introduced an effective and efficient approximation algorithm for computing these probabilities. The technique has been experimentally evaluated on a challenging and involved real-life problem of mining biological networks. ProbLog's key advantage in this type of application is its simplicity as well as the flexibility for posing complex queries that support the analyst. There are several questions for further research. The most prominent ones are whether techniques employed for learning PRISM programs or SLPs could be adapted for learning ProbLog, and vice versa, whether the independence assumption made in ProbLog could be adapted for use in PRISM or PHA.

## Acknowledgements

We thank Sevon et al. [Sevon *et al.*, 2006] for the biological data. Hannu Toivonen has been supported by Alexander von Humboldt Foundation and Tekes. This work was also partly supported by the EU IST FET projects April and IQ.

## References

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

[Dantsin, 1991] E. Dantsin. Probabilistic logic programs and their semantics. In Andrei Voronkov, editor, *Proc. 1st Russian Conf. on Logic Programming*, volume 592 of *LNCS*, pages 152–164. Springer, 1991.

[Flach, 1994] P.A. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley, 1994.

[Fuhr, 2000] N. Fuhr. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science*, 51:95–110, 2000.

[Luo and Trivedi, 1998] T. Luo and K.S. Trivedi. An improved algorithm for coherent-system reliability. *IEEE Transactions on Reliability*, 47(1):73–78, 1998.

[Muggleton, 1996] S. H Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*. IOS Press, 1996.

[Perez-Iratxeta *et al.*, 2002] C. Perez-Iratxeta, P. Bork, and M.A Andrade. Association of genes to genetically inherited diseases using data mining. *Nature Genetics*, 31:316–319, 2002.

[Poole, 1992] D. Poole. Logic programming, abduction and probability. In *Fifth Generation Computing Systems*, pages 530–538, 1992.

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

[Richardson and Domingos, 2006] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[Sato and Kameya, 2001] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of AI Research*, 15:391–454, 2001.

[Sevon *et al.*, 2006] P. Sevon, L. Eronen, P. Hintsanen, K. Kulovesi, and H. Toivonen. Link discovery in graphs derived from biological databases. In *Data Integration in the Life Sciences 2006*, volume 4075 of *LNBI*, pages 35 – 49, 2006. Springer.

[Valiant, 1979] L.G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8:410– 411, 1979.