

Compressing Probabilistic Prolog Programs

L. De Raedt², K. Kersting¹, A. Kimmig², K. Revoredó¹,
H. Toivonen³

¹ Institut für Informatik, Albert-Ludwigs-Universität,
Georges-Köhler-Allee, Gebäude 079, D-79110 Freiburg im Breisgau, Germany
e-mail: `kersting@informatik.uni-freiburg.de`, `kate@cos.ufrj.br`

² Departement Computerwetenschappen, K.U. Leuven,
Celestijnenlaan 200A - bus 2402, B-3001 Heverlee, Belgium
e-mail: `{luc.deraedt,angelika.kimmig}@cs.kuleuven.be`

³ Department of Computer Science,
P.O. Box 68, FI-00014 University of Helsinki, Finland
e-mail: `hannu.toivonen@cs.helsinki.fi`

Received: date / Revised version: date

Abstract ProbLog is a recently introduced probabilistic extension of Prolog [4]. 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 in a randomly sampled program.

This paper introduces the theory compression task for ProbLog, which consists of selecting that subset of clauses of a given ProbLog program that maximizes the likelihood w.r.t. a set of positive and negative examples. Experiments in the context of discovering links in real biological networks demonstrate the practical applicability of the approach.

1 Introduction

The past few years have seen a surge of interest in the field of probabilistic logic learning or statistical relational learning (e.g. [3,7]). In this endeavor, many probabilistic logics have been developed. Prominent examples include PHA [13], PRISM [14], SLPs [10], and probabilistic Datalog (pD) [6]. These frameworks attach probabilities to logical formulae, most often definite clauses, and typically impose further constraints to facilitate the computation of the probability of queries and simplify the learning algorithms for such representations.

Our work on this topic has been motivated by mining of large biological networks where edges are labeled with probabilities. Such 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 [15]. Such networks can be easily modeled by our recent probabilistic extension of Prolog, called ProbLog [4]. ProbLog is essentially Prolog where all clauses 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 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 a random subprogram. The semantics of ProbLog is not really new, it closely corresponds to that of pD [6] and is closely related though different from the pure distributional semantics of [14], cf. also Section 8. However, the key contribution of ProbLog [4] is the introduction of an effective inference procedure for this semantics, which enables its application to the biological link discovery task.

The key contribution of the present paper is the introduction of the task of compressing a ProbLog theory using a set of positive and negative examples, and the development of an algorithm for realizing this. Theory compression refers to the process of removing as many clauses as possible from the theory in such a manner that the compressed theory explains the examples as good as possible. The compressed theory should be a lot smaller, and therefore easier to understand and employ. It will also contain the essential components of the theory needed to explain the data. The theory compression problem is again motivated by the biological application. In this application, scientists try to analyze large networks of links in order to obtain an understanding of the relationships amongst a typically small number of nodes. The idea now is to remove as many links from these networks as possible using a set of positive and negative examples. The examples take the form of relationships that are either interesting or uninteresting to the scientist. The result should ideally be a small network that contains the essential links and assigns high probabilities to the positive and low probabilities to the negative examples. This task is analogous to a form of theory revision [17] where the only operation allowed is the deletion of rules or facts. The analogy explains why we have formalized the theory compression task within the ProbLog framework. Within this framework, examples are true and false ground facts, and the task is to find a small subset of a given ProbLog program that maximizes the likelihood of the examples.

This paper is organized as follows: in Section 2, the biological motivation for ProbLog and theory compression is discussed; in Section 3, the semantics of ProbLog are briefly reviewed; in Section 4, the inference mechanism for computing the success probability of ProbLog queries as introduced by [4] is reviewed; in Section 5, the task of probabilistic theory compression is defined; and an algorithm for tackling the compression problem is presented

in Section 6. Experiments that evaluate the effectiveness of the approach are presented in Section 7, and finally, in Section 8, we discuss some related work and conclude.

2 Example: ProbLog for biological graphs

As a motivating application, consider link mining in networks of biological concepts. Molecular biological data is available from public sources, such as Ensembl¹, NCBI Entrez², 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 such data has been identified as an important and challenging task (cf. [11]).

For instance, a biologist may be interested in the potential relationships between a given set of proteins. If the original graph contains more than some dozens of nodes, manual and visual analysis is difficult. Within this setting, our goal is to automatically extract a relevant subgraph which contains the most important connections between the given proteins. This result can then be used by the biologist to study the potential relationships much more efficiently.

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 [15]. A ProbLog representation of such a graph could simply consist of probabilistic `edge/2` facts though finer grained representations using relations such as `codes/2`, `expresses/2` would also be possible. In a probabilistic graph, the strength of a connection between two nodes can be measured as the probability that a path exists between the two given nodes [15]³. Such queries are easily expressed in ProbLog by defining the (non-probabilistic) predicate `path(N1,N2)` in the usual way, using probabilistic `edge/2` facts. Obviously, logic – and ProbLog – can easily be used to express much more complex possible relations. For simplicity of exposition we here only consider a simple representation of graphs, and in the future will address more complex applications and settings.

¹ www.ensembl.org

² www.ncbi.nlm.nih.gov/Entrez/

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

3 ProbLog: Probabilistic Prolog

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 that it is true.

Example 1 Within bibliographic data analysis, the similarity structure among items can improve information retrieval results. Consider a collection of papers $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$ and some pairwise similarities $\mathbf{similar}(\mathbf{a}, \mathbf{c})$, e.g., based on key word analysis. Two items \mathbf{X} and \mathbf{Y} are $\mathbf{related}(\mathbf{X}, \mathbf{Y})$ if they are similar (such as \mathbf{a} and \mathbf{c}) or if \mathbf{X} is similar to some item \mathbf{Z} which is related to \mathbf{Y} . Uncertainty in the data and in the inference can elegantly be represented by the attached probabilities:

$$\begin{aligned} 1.0 &: \mathbf{related}(\mathbf{X}, \mathbf{Y}) : - \mathbf{similar}(\mathbf{X}, \mathbf{Y}). \\ 0.8 &: \mathbf{related}(\mathbf{X}, \mathbf{Y}) : - \mathbf{similar}(\mathbf{X}, \mathbf{Z}), \mathbf{related}(\mathbf{Z}, \mathbf{Y}). \\ 0.9 &: \mathbf{similar}(\mathbf{a}, \mathbf{c}). \quad 0.7 : \mathbf{similar}(\mathbf{c}, \mathbf{b}). \\ 0.6 &: \mathbf{similar}(\mathbf{d}, \mathbf{c}). \quad 0.9 : \mathbf{similar}(\mathbf{d}, \mathbf{b}). \end{aligned}$$

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

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

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

$$P(q|T) = \sum_{L \subseteq L_T} P(q, L|T) = \sum_{L \subseteq L_T} P(q|L) \cdot P(L|T), \quad (2)$$

where $P(q|L) = 1$ if there exists a θ such that $L \models q\theta$, and $P(q|L) = 0$ otherwise. 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 $T = \{p_1 : c_1, \dots, p_n : c_n\}$ and a query q , the trivial way of computing the success probability $P(q|T)$ enumerates all possible logic programs $L \subseteq L_T$ (cf. Eq. (2)). Clearly this is infeasible for all but the tiniest programs. Therefore, the inference engine proceeds in two steps (cf. [6,4]). The first step reduces the problem of computing the success probability of a ProbLog query to that of computing the probability of a monotone Boolean DNF formula. The second step then computes the probability of this formula.

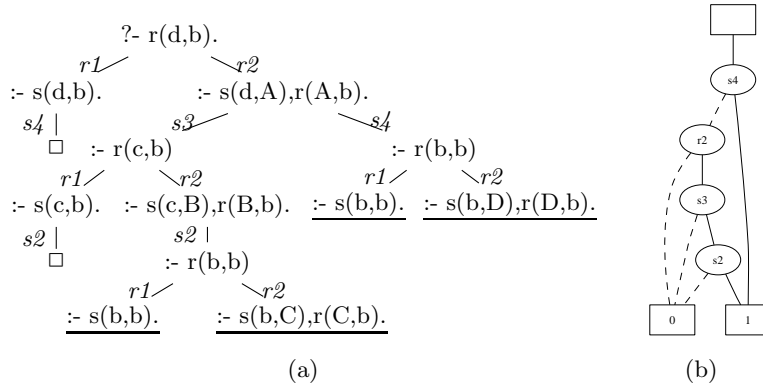


Fig. 1 SLD-tree (a) and BDD (b) for `related(d, b)`, corresponding to formula (4).

4.1 ProbLog queries as DNF formulae

To be able to write down the DNF formula corresponding to a query, we use a Boolean random variable b_i for each clause $p_i : c_i \in T$, indicating whether c_i is in the logic program; i.e., b_i has probability p_i of being true. The probability of a particular proof involving clauses $\{p_1 : d_1, \dots, p_k : d_k\} \subseteq T$ is then the probability of the conjunctive formula $b_1 \wedge \dots \wedge b_k$. 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:

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

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 .

To obtain the set of all proofs of a query, we construct the standard SLD-tree (cf. [5] for more details) for a query q using the logical part of the theory, i.e. L_T , and label each edge in the SLD-tree by the Boolean variable indicating the clause used in the SLD-resolution step. From this labeled SLD-tree, one can then read off the above defined Boolean DNF formula.

Example 2 There are two proofs of `related(d, b)`, obtained using either the base case and one fact, or the recursive case and two facts. The corresponding SLD-tree is depicted in Figure 1(a) and – as r_1 is always true – the Boolean formula is

$$(r_1 \wedge s_4) \vee (r_2 \wedge s_3 \wedge r_1 \wedge s_2) = s_4 \vee (r_2 \wedge s_3 \wedge s_2). \quad (4)$$

4.2 Computing the probability of DNF formulae

Computing the probability of DNF formulae is an NP-hard problem [16] 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. 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, but is still used by the probabilistic Datalog engine Hyspirit [6], which explains why inference with Hyspirit on about 10 or more conjuncts is infeasible according to [6]. In contrast, ProbLog employs the latest advances made in the manipulation and representation of Boolean formulae using binary decision diagrams (BDDs) [1], and is able to deal with up to 100000 conjuncts [4].

A BDD is an efficient graphical representation of a Boolean function over a set of variables. A BDD representing the second formula in Eq. (4) is shown in Figure 1(b). 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 1(b), dashed edges indicate 0's and lead to low children, solid ones indicate 1's and lead to high children. The two leafs are called the 0- and 1-terminal node respectively.

BDDs are one of the most popular data structures used within many branches of computer science, such as computer architecture and verification, even though their use is perhaps not yet so widespread in artificial intelligence and machine learning (but see [2] and [9] for recent work on Bayesian networks using variants of BDDs). Since their introduction by Bryant [1], there has been a lot of research on BDDs and their computation. Many variants of BDDs and off the shelf systems exist today. An important property of BDDs is that their size is highly dependent on the variable ordering used. In ProbLog we use the general purpose package CUDD⁴ for constructing and manipulating the BDDs and leave the optimization of the variable reordering to the underlying BDD package. Nevertheless, the initial variable ordering when building the BDD is given by the order that they are encountered in the SLD-tree, i.e. from root to leaf. The BDD is built by combining BDDs for subtrees, which allows structure sharing.

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. Es-

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

Algorithm 1 Probability calculation for BDDs

```

PROBABILITY(input: BDD node  $n$ )
1  if  $n$  is the 1-terminal then return 1
2  if  $n$  is the 0-terminal then return 0
3  let  $p_n$  be the probability of the clause represented by  $n$ 's random variable
4  let  $h$  and  $l$  be the high and low children of  $n$ 
5   $prob(h) :=$  PROBABILITY( $h$ )
6   $prob(l) :=$  PROBABILITY( $l$ )
7  return  $p_n \cdot prob(h) + (1 - p_n) \cdot prob(l)$ 

```

sentially, at each inner node, probabilities from both children are calculated recursively and combined afterwards, as indicated in Algorithm 1.

4.3 An approximation algorithm

Since for realistic applications, such as large graphs in biology or elsewhere, the size of the DNF can grow exponentially, we have introduced in [4] an approximation algorithm for computing success probabilities along the lines of [12].

The idea is that one can impose a depth-bound on the SLD-tree and use this to compute two DNF formulae. The first one, called *low*, encodes all successful proofs obtained at or above that level. The second one, called *up*, encodes all derivations till that level that have not yet failed, and, hence, for which there is hope that they will still contribute a successful proof. We then have that $P(low) \leq P(q|T) \leq P(up)$. This directly follows from the fact that $low \models d \models up$ where d is the Boolean DNF formula corresponding to the full SLD-tree of the query.

This observation is then turned into an approximation algorithm by using iterative deepening, where the idea is that the depth-bound is gradually increased until convergence, i.e., until $|P(up) - P(low)| \leq \delta$ for some small δ . The approximation algorithm is described in more detail in [4], where it is also applied to link discovery in biological graphs.

Example 3 Consider the SLD-tree in Figure 1(a) only till depth 2. In this case, d_1 encodes the left success path while d_2 additionally encodes the paths up to **related(c, b)** and **related(b, b)**, i.e. $d_1 = (r_1 \wedge s_4)$ and $d_2 = (r_1 \wedge s_4) \vee (r_2 \wedge s_3) \vee (r_2 \wedge s_4)$. The formula for the full SLD-tree is given in Eq. (4).

5 Compressing ProbLog theories

Before introducing the ProbLog theory compression problem, it is helpful to consider the corresponding problem in a purely logical setting⁵. Assume that, as in traditional theory revision [17,18], one is given a set of positive and negative examples in the form of true and false facts. The problem then is to find a theory that best explains the examples, i.e., one that scores best w.r.t. a function such as accuracy. At the same time, the theory should be *small*, that is it should contain less than k clauses. Rather than allowing any type of revision on the original theory, compression only allows for clause deletion. So, logical theory compression aims at finding a small theory that best explains the examples. As a result the compressed theory should be a better fit w.r.t. the data but should also be much easier to understand and to interpret. This holds in particular when starting with large networks containing thousands of nodes and edges and then obtaining a small compressed graph that consists of say 20 edges. In biological databases such as the ones considered in this paper, scientists can easily analyse the interactions in such small networks but have a very hard time with the large networks.

The *ProbLog Theory Compression Problem* is now an adaptation of the traditional theory revision (or compression) problem towards probabilistic Prolog programs. It can be formalized as follows:

Given

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

find a theory $T \subseteq S$ of size at most k (i.e. $|T| \leq k$) that has a maximum likelihood w.r.t. the examples $E = P \cup N$, i.e.,

$$T = \arg \max_{T \subseteq S \wedge |T| \leq k} \mathcal{L}(E|T), \text{ where } \mathcal{L}(E|T) = \prod_{e \in E} \mathcal{L}(e|T) \text{ and } \quad (5)$$

$$\mathcal{L}(e|T) = \begin{cases} P(e|T) & \text{if } e \in P \\ 1 - P(e|T) & \text{if } e \in N. \end{cases} \quad (6)$$

So, we are interested in finding a small theory that maximizes the likelihood. Here, small means that the number of clauses should be at most k . Also, rather than maximizing the accuracy as in purely logical approaches, in ProbLog we maximize the likelihood of the data. Here, a ProbLog theory T is used to determine a relative class distribution: it gives the probability $P(e|T)$ that any given example e is positive. (This is subtly different from specifying the distribution of (positive) examples.) The examples are assumed to be mutually independent, so the total likelihood is obtained as

⁵ This can – of course – be modelled within ProbLog by setting the labels of all clauses to 1.

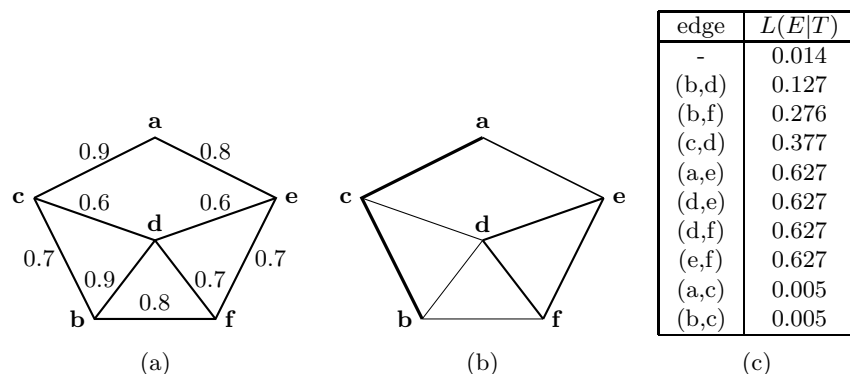


Fig. 2 Illustration of Examples 4 and 5: (a) Initial **related** theory. (b) Result of compression using positive example **related(a, b)** and negative example **related(d, b)**, where edges are removed greedily (in order of increasing thickness). (c) Likelihoods obtained as edges are removed in the indicated order.

a simple product. For an optimal ProbLog theory T , the probability of the positives is as close to 1 as possible, and for the negatives as close to 0 as possible. However, because we want to allow misclassifications but with a high cost in order to avoid overfitting, to effectively handle noisy data, and to obtain smaller theories, we slightly redefine $P(e|T)$ in Eq. (6), as $\hat{P}(e|T) = \max(\min[1 - \epsilon, P(e|T)], \epsilon)$ for some constant $\epsilon > 0$ specified by the user. This avoids the possibility that the likelihood function becomes 0, e.g., when a positive example is not covered by the theory at all.

Example 4 Figure 2(a) graphically depicts a slightly extended version of our bibliographic theory from Example 1. Assume we are interested in items related to item **b**, and user feedback revealed that item **a** is indeed related to item **b**, but **d** is actually not. We might then use those examples to compress our initial theory to the most relevant parts, giving k as the maximal acceptable size.

6 The ProbLog theory compression algorithm

As already outlined in Figure 2, the ProbLog compression algorithm removes one clause at a time from the theory, and chooses the clause greedily to be the one whose removal results in the largest likelihood. The more detailed ProbLog compression algorithm as given in Algorithm 2 works in two phases. First, it constructs BDDs for proofs of the examples using the standard ProbLog inference engine. These BDDs then play a key role in the second step where clauses are greedily removed, as they make it very efficient to test the effect of removing a clause.

More precisely, the algorithm starts by calling the approximation algorithm sketched in Section 4.3, which computes the DNFs and BDDs for

Algorithm 2 ProbLog theory compression

```

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

```

lower and upper bounds. The compression algorithm only employs the lower bound DNFs and BDDs, since they are simpler and, hence, more efficient to use. All clauses used in at least one proof occurring in the (lower bound) BDD of some example constitute the set R of possible revision points. All other clauses do not occur in any proof contributing to probability computation, and can therefore be immediately removed; this step alone often gives high compression factors. Alternatively, if the goal is to minimize the changes to theory, rather than the size of the resulting theory, then all these other clauses should be left intact.

After the set R of revision points has been determined — and the other clauses potentially removed — the ProbLog theory compression algorithm performs a *greedy* search in the space of subsets of R . At each step, the algorithm finds that clause whose deletion results in the best likelihood score, and then deletes it. This process is continued until both $|R| \leq k$ and deleting further clauses does not improve the likelihood.

Compression is efficient, since the (expensive) construction of the BDDs is performed only once per example. Given a BDD for a query q (from the approximation algorithm) one can easily evaluate (in the revision phase) conditional probabilities of the form $P(q|T, b'_1 \wedge \dots \wedge b'_k)$, where the b'_i s are possibly negated Booleans representing the truth-values of clauses. To compute the answer using Algorithm 1, one only needs to reset the probabilities p'_i of the b'_i s. If b'_j is a positive literal, the probability of the corresponding variable is set to 1, if b'_j is a negative literal, it is set to 0. The structure of the BDD remains the same. When compressing theories by only deleting clauses, the b'_i s will be negative, so one has to set $p'_i = 0$ for all b'_i s. Figure 3 gives an illustrative example of deleting one clause.

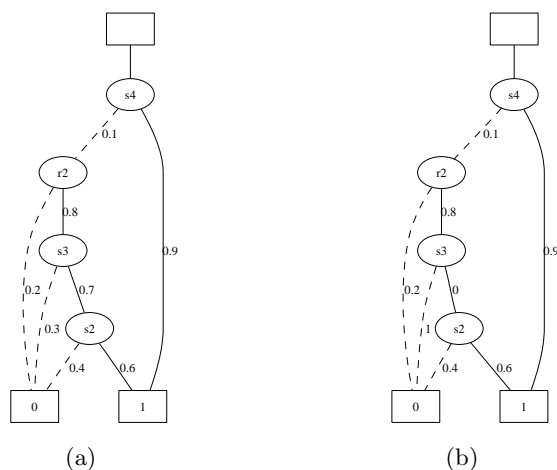


Fig. 3 Effect of deleting clause $s3$ in Example 2: (a) Initial BDD: $P(e|T) = 0.9 + 0.1 \cdot 0.8 \cdot 0.7 \cdot 0.6 = 0.9336$. (b) BDD after deleting $s3$ by setting its probability to 0: $P(e|T) = 0.9 + 0.1 \cdot 0.8 \cdot 0 \cdot 0.6 = 0.9$.

Example 5 Reconsider the **related** theory from Example 4, Figure 2(a), where edges can be used in both directions and a positive example **related(a, b)** as well as a negative example **related(d, b)** are given. With default probability $\epsilon = 0.005$, the initial likelihood of those two examples is 0.014. The greedy approach first deletes 0.9 : **similar(d, b)** and thereby increases the likelihood to 0.127. The probability of the positive example **related(a, b)** is now 0.863 (was 0.928), and that of the negative example **related(d, b)** is 0.853 (was 0.985). The final result is a theory just consisting of the two edges **similar(a, c)** and **similar(c, b)** which leads to a total likelihood of 0.627, cf. Figures 2(b) and 2(c).

7 Experiments

We performed a number of experiments to study both the quality and the complexity of ProbLog theory compression. The quality issues that we address empirically concern (1) the relationship between the amount of compression and the resulting likelihood of the examples, and (2) the impact of compression on clauses or hold-out test examples, where desired or expected results are known. We next describe two central components of the experiment setting: data and algorithm implementation.

7.1 Data

We performed tests primarily with real biological data. For some statistical analyses we needed a large number of experiments, and for these we used simulated data, i.e., random graphs with better controlled properties.

Real Data: Real biological graphs were extracted from NCBI Entrez and some other databases as described in [15]. Since NCBI Entrez alone has tens of millions of objects, we adopted a two-staged process of first using a rough but efficient method to extract a graph G that is likely to be a supergraph of the desired final result, and then applying ProbLog on graph G to further identify the most relevant part.

To obtain a realistic test setting with natural example queries, we extracted two graphs related to four random Alzheimer genes (HGNC ids 620, 582, 983, and 8744). Pairs of genes are used as positive examples, in the form `? - path(gene_620, gene_582)`. Since the genes all relate to Alzheimer disease, they are likely to be connected via nodes of shared relevance, and the connections are likely to be stronger than for random pairs of nodes.

A smaller and a larger graph were obtained by taking the union of subgraphs of radius 2 (smaller graph) or 3 (larger graph) from the four genes and producing weights as described in [15]. In the smaller graph, gene 8744 was not connected to the other genes, and the corresponding component was left out. As a result the graph consists of 144 edges and 79 nodes. Paths between the three remaining Alzheimer genes were used as positive examples. All our tests use this smaller data and the three positive examples unless otherwise stated. The larger graph consists of 11530 edges and 5220 nodes and was used for scalability experiments. As default parameter values we used probability $\epsilon = 10^{-8}$ and interval width $\delta = 0.1$.

Simulated Data: Synthetic graphs with a given number of nodes and a given average degree were produced by generating edges randomly and uniformly between nodes. This was done under the constraint that the resulting graph must be connected, and that there can be at most one edge between any pair of nodes. The resulting graph structures tend to be much more challenging than in the real data, due to the lack of structure in the data. The default values for parameters were $\epsilon = 0.001$ and $\delta = 0.2$.

7.2 Algorithm and implementation

The approximation algorithm that builds BDDs is based on iterative deepening. We implemented it in a best-first manner: the algorithm proceeds from the most likely proofs (derivations) to less likely ones, as measured by the product of the probabilities of clauses in a proof. Proofs are added in batches to avoid repetitive BDD building.

We implemented the inference and compression algorithms in Prolog (Yap-5.1.0). For efficiency reasons, proofs for each example are stored internally in a prefix tree. Our ProbLog implementation then builds a BDD for an example by using the available CUDD operators, according to the structure of the prefix tree. The compiled result is saved for use in the revision phase.

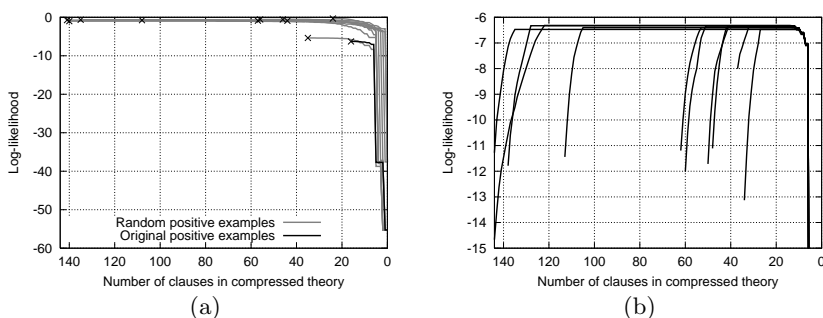


Fig. 4 Evolution of log-likelihood for 10 test runs with positive examples only (a) and with both positive and negative examples (b). Different starting points of lines reflect the number of clauses in the BDDs used in theory compression.

7.3 Quality of ProbLog theory compression

The quality of compressed ProbLog theories is hard to investigate using a single objective. We therefore carried out a number of experiments investigating different aspects of quality, which we will now discuss in turn.

How does the likelihood evolve during compression? Figure 4(a) shows how the log-likelihood evolves during compression in the real data, using positive examples only, when all revision points are eventually removed. In one setting (black line), we used the three original paths connecting Alzheimer gene pairs as examples. This means that for each gene pair $(g1, g2)$ we provided the example `path(g1, g2)`. To obtain a larger number of results, we artificially generated ten other test settings (grey lines), each time by randomly picking three nodes from the graph and using paths between them as positive examples, i.e., we had three `path` examples in each of the ten settings. Very high compression rates can be achieved: starting from a theory of 144 clauses, compressing to less than 20 clauses has only a minor effect on the likelihood. The radical drops in the end occur when examples cannot be proven anymore.

To test and illustrate the effect of negative examples (undesired paths), we created 10 new settings with both positive and negative examples. This time the above mentioned 10 sets of random paths were used as negative examples. Each test setting uses the paths of one such set as negative examples together with the original positive examples (paths between Alzheimer genes). Figure 4(b) shows how the total log-likelihood curves have a nice convex shape, quickly reaching high likelihoods, and only dropping with very small theories. A factorization of the log-likelihood to the positive and negative components (Figure 5(a)) explains this: clauses that affect the negative examples mostly are removed first (resulting in an improvement of the likelihood). Only when no other alternatives exist, clauses important for the positive examples are removed (resulting in a decrease in the likelihood). This suggests that negative examples can be used effectively to guide the

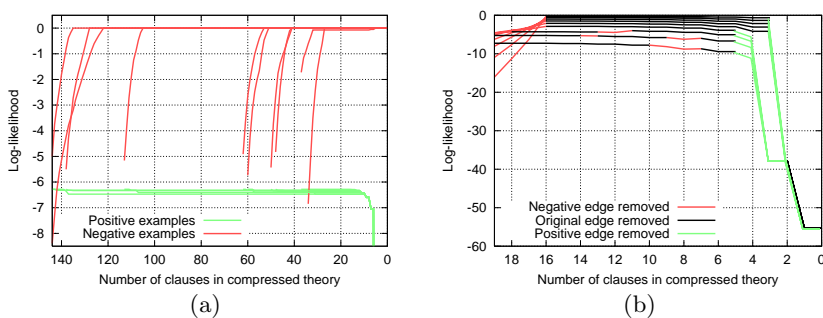


Fig. 5 Evolution of log-likelihood for 10 test runs with both positive and negative examples (a). Evolution of log-likelihood for settings with artificially implanted edges with negative and positive effects (b). In (b), curve color indicates the type of edge that was removed in the corresponding revision step. Likelihoods in the middle reflect the probability on artificial edges: topmost curve is for $p = 0.9$, going down in steps of size 0.1

compression process, an issue to be studied shortly in a cross-validation setting.

How appropriately are edges of known positive and negative effects handled? Next we inserted new nodes and edges into the real biological graph, with clearly intended positive or negative effects. We forced the algorithm to remove all generated revision points and obtained a ranking of edges. To get edges with a negative effect, we added a new “negative” node `neg` and edges `edge(gene_620,neg)`, `edge(gene_582,neg)`, `edge(gene_983,neg)`. Negative examples were then specified as paths between the new negative node and each of the three genes. For a positive effect, we added a new “positive” node and three edges in the same way, to get short artificial connections between the three genes. As positive examples, we again used `path(g1,g2)` for the three pairs of genes. All artificial edges were given the same probability p .

(Different values of p lead to different sets of revision points. This depends on how many proofs are needed to reach the probability interval δ . To obtain comparable results, we computed in a first step the revision points using $p = 0.5$ and interval width 0.2. All clauses not appearing as a revision point were excluded from the experiments. On the resulting theory, we then reran the compression algorithm for $p = 0.1, 0.2, \dots, 0.9$ using $\delta = 0.0$, i.e., using exact inference.)

Figure 5(b) shows the log-likelihood of the examples as a function of the number of clauses remaining in the theory during compression; the types of removed edges are coded by colors. In all cases, the artificial negative edges were removed early, as expected. For probabilities $p \geq 0.5$, the artificial positive edges were always the last ones to be removed. This also corresponds to the expectations, as these edges can contribute quite a lot to the positive examples. Results with different values of p indicate how sensitive the

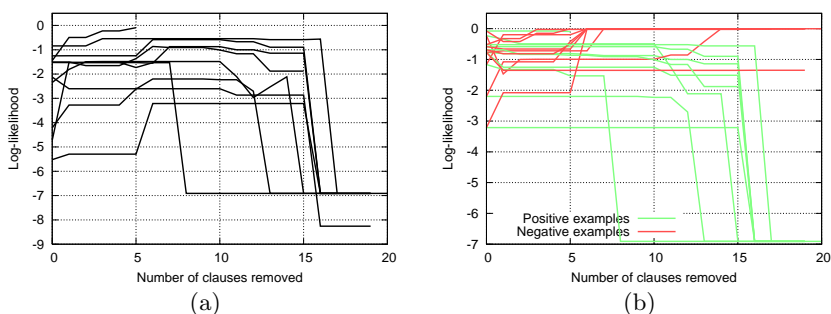


Fig. 6 Evolution of log-likelihoods in test sets for 10 random runs: total log-likelihood (a) and log-likelihoods of positive and negative test examples separately (b).

method is to recognize the artificial edges. Their influence drops together with p , but negative edges are always removed before positive ones.

How does compression affect unknown test examples? We next study generalization beyond the (training) examples used in compression. We illustrate this in an alternative setting for ProbLog theory revision, where the goal is to minimize changes to the theory. More specifically, we do not modify those parts of the initial theory that are not relevant to the training examples. This is motivated by the desire to apply the revised theory also on unseen test examples that may be located outside the subgraph relevant to training examples, otherwise they would all be simply removed. Technically this is easily implemented: clauses that are not used in any of the training example BDDs are kept in the compressed theory.

Since the difficulty of compression varies greatly between different graphs and different examples, we used a large number of controlled random graphs and constructed positive and negative examples as follows. First, three nodes were randomly selected: a target node (“disease”) from the middle of the graph, a center for a positive cluster (“disease genes”) closer to the perimeter, and a center for a negative cluster (“irrelevant genes”) at random. Then a set of positive nodes was picked around the positive center, and for each of them a positive example was specified as a path to the target node. In the same way, a cluster of negative nodes and a set of negative examples were constructed. A cluster of nodes is likely to share subpaths to the target node, resulting in a concept that can potentially be learnt. By varying the tightness of the clusters, we can tune the difficulty of the task. In the following experiments, negative nodes were clustered but the tightness of the positive cluster was varied. We generated 1000 random graphs. Each of our random graphs had 20 nodes of average degree 3. There were 3 positive and 3 negative examples, and one of each was always in the hold-out dataset, leaving 2 + 2 examples in the training set. This obviously is a challenging setting.

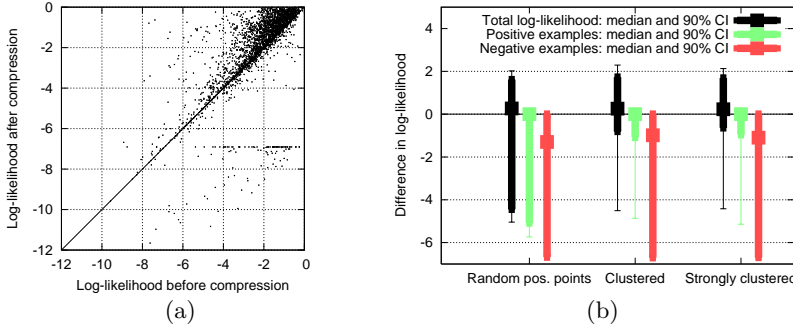


Fig. 7 Log-likelihoods of the test sets before and after compression (a). Distributions of differences of log-likelihoods in test examples before and after compression, for three different densities of positive nodes (b) (thick line: 90% confidence interval; thin line: 95% confidence interval; for negative test examples, differences in $\log(1-\text{likelihood})$ are reported to make results comparable with positive examples).

We compressed each of the ProbLog theories based on the training set only; Figure 6 shows 10 random traces of the log-likelihoods in the hold-out test examples. The figure also gives a break-down to separate log-likelihoods of positive and negative examples. The behavior is much more mixed than in the training set (cf. Figures 4(b) and 5(a)), but there is a clear tendency to first improve likelihood (using negative examples) before it drops for very small theories (because positive examples become unprovable).

To study the relationship between likelihood in the training and test sets more systematically, we took for each random graph the compressed theory that gave the maximum likelihood in the training set. A summary over the 1000 runs, 3-fold cross-validation each, is given in Figure 7(a). The first observation is that compression is on average useful for the test set. The improvement over the original likelihood is on average about 30% (0.27 in log-likelihood scale), but the variance is large. The large variance is primarily caused by cases where the positive test example was completely disconnected from the target node, resulting in the use of probability $\epsilon = 0.001$ for the example, and probabilities ≤ 0.001 for the pair of examples. These cases are visible as a cloud of points below log-likelihood $\log(0.001) = -6.9$. The joint likelihood of test examples was improved in 68% of the cases, and decreased only in about 17% of the cases (and stayed the same in 15%). Statistical tests of either the improvement of the log-likelihood (paired t-test) or the proportion of cases of increased likelihood (binomial test) show that the improvement is statistically extremely significant (note that there are $N = 3000$ data points).

Another illustration of the powerful behaviour of theory compression is given in Figure 7(b). It shows the effect of compression, i.e., the change in test set likelihood that resulted from maximum-likelihood theory compression for three different densities of positive examples; it also shows separately

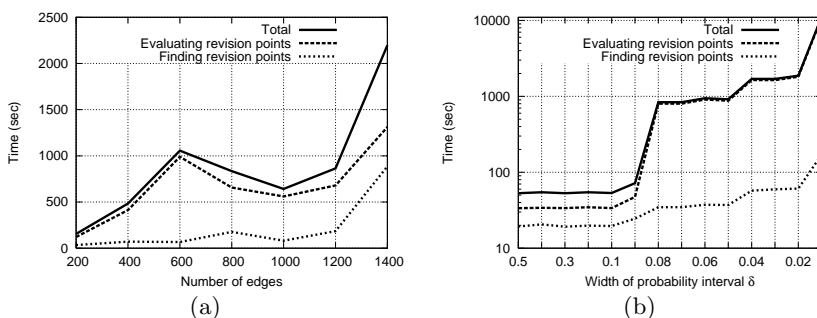


Fig. 8 Running time as a function of graph size (number of edges) (a) and as a function of interval width δ (b). Note that in (b) the y axis is in log-scale and the x axis is discretized in 0.1 steps for $[0.5, 0.1]$ and in 0.01 steps in $[0.1, 0.01]$.

the result for positive and negative test examples. Negative test examples experience on average a much clearer change in likelihood than the positive ones, demonstrating that ProbLog compression does indeed learn. Further, in all these settings, the median change in positive test examples is zero, i.e., more than one half of the cases experienced no drop in the probability of positive examples, and for clustered positive examples 90% of the cases are relatively close to zero. Results for the negative examples are markedly different, with much larger proportions of large differences in log-likelihood.

To summarize, all experiments show that our method yields good compression results. The likelihood is improving, known positive and negative examples are respected, and the result generalizes nicely to unknown examples. Does this, however, come at the expense of very high running times? This question will be investigated in the following subsection.

7.4 Complexity of ProbLog theory compression

There are several factors influencing the running time of our compression approach. The aim of the following set of experiments was to figure out the crucial factors on running times.

How do the methods scale up to large graphs? To study the scalability of the methods, we randomly subsampled edges from the larger biological graph, to obtain subgraphs $G_1 \subset G_2 \subset \dots$ with 200, 400, ... edges. Each G_i contains the three genes and consists of one connected component. Average degree of nodes ranges in G_i s approximately from 2 to 3. The ProbLog compression algorithm was then run on the data sets, with k , the maximum size of the compressed theory, set to 15.

Running times are given in Figure 8(a). Graphs of 200 to 1400 edges were compressed in 3 to 40 minutes, which indicates that the methods can be useful in practical, large link mining tasks. The results also nicely illustrate how difficult it is to predict the problem complexity: up to 600 edges the running times increase, but then drop when increasing the graph size to

1000 edges. Larger graphs can be computationally easier to handle, if they have additional edges that result in good proofs and remove the need of deep search for obtaining tight bounds.

What is the effect of the probability approximation interval δ ?

Smaller values of δ obviously are more demanding. To study the relationship, we ran the method on the smaller biological graph with the three positive examples using varying δ values. Figure 8(b) shows the total running time as well as its decomposition to two phases, finding revision points and removing them. The interval width δ has a major effect on running time; in particular, the complexity of the revision step is greatly affected by δ .

What are the crucial technical factors for running times?

We address the two phases, finding revision points and removing them, separately.

Given an interval δ , resources needed for running the approximation algorithm to obtain the revision points are hard to predict, as those are extremely dependent on the sets of proofs and especially stopped derivations encountered until the stopping criterion is reached. This is not only influenced by the structure of the theory and δ , but also by the presentation of examples. (In one case, reversing the order of nodes in some path atoms decreased running time by several orders of magnitude.) Based on our experiments, a relatively good indicator of the complexity is the total size of the BDDs used.

The complexity of the revision phase is more directly related to the number of revision points. Assuming a constant time for using a given BDD to compute a probability, the time complexity is quadratic in the number of revision points. In practice, the cost of using a BDD depends of course on its size, but compared to the building time, calling times are relatively small. One obvious way of improving the efficiency of the revision phase is to greedily remove more than one clause at a time.

Although in Figure 8(b) the revision time almost always dominates the total time, we have found in our experiments that there is a lot of variance here, too, and in practice either phase can strongly dominate the total time.

Given a fixed value for δ , there is little we can do to affect the size of the BDDs or the number of revision points. However, these observations may be useful for designing alternative parameterizations for the revision algorithm that have more predictable running times (with the cost of less predictable probability intervals).

8 Related work and conclusions

Using the probabilistic variant of Prolog, called ProbLog, we have introduced a novel framework for theory compression in large probabilistic databases. ProbLog’s only assumption is that the probabilities of clauses are independent of one another. The semantics of ProbLog are related to those of some other well-known probabilistic extensions of Prolog such as

pD [6], SLPs [10], PRISM [14] and PHA [13], but also subtly different. For instance, pD assigns a Boolean variable to each *instance* of a clause and its inference procedure only works for very small problems (cf. above). In SLPs and PRISM, each (repeated) use of a probabilistic clause (or switch) explicitly contributes to the probability of a proof, whereas in ProbLog there is a single random variable that covers all calls to that fact or clause. The number of random variables is thus fixed in ProbLog, but proof-dependent in the other frameworks. Nevertheless, it would be interesting to explore the possibility of transferring algorithms between those systems.

ProbLog has been used to define a new type of theory compression problem, aiming at finding a small subset of a given program that maximizes the likelihood w.r.t. a set of positive and negative examples. This problem is related to the traditional theory revision problem studied in inductive logic programming [17] in that it allows particular operations on a theory (in this case only deletions) on the basis of positive and negative examples but differs in that it aims at finding a small theory and that is also grounded in a sound probabilistic framework. This framework bears some relationships to the PTR approach by [8] in that possible revisions in PTR are annotated with weights or probabilities. Still, PTR interpretes the theory in a purely logical fashion to classify examples. It only uses the weights as a kind of bias during the revision process in which it also updates them using examples in a kind of propagation algorithm. When the weights become close to 0, clauses are deleted. ProbLog compression is also somewhat related to Zelle and Mooney’s work on Chill [18] in that it specializes an overly general theory but differs again in the use of a probabilistic framework.

A solution to the ProbLog theory compression problem has then been developed. Central in the development of the ProbLog inference and compression algorithms is the use of BDDs [1]. BDDs are a popular and effective representation of Boolean functions. Although different variants of BDDs are used by Chavira and Darwiche [2] and Minato *et al.* [9] to benefit from local structure during probability calculation in Bayesian networks, ProbLog is – to the best of our knowledge – the first probabilistic logic that uses BDDs.

Finally, we have shown that ProbLog inference and compression is not only theoretically interesting, but is also applicable on various realistic problems in a biological link discovery domain.

Acknowledgements We are grateful to P. Sevon, L. Eronen, P. Hintsanen and K. Kulovesi for the real data. K. Kersting and A. Kimmig have been supported by the EU IST FET project April II. K. Revoredo has been supported by Brazilian Research Council, CNPq. H. Toivonen has been supported by the Humboldt foundation and Tekes.

References

1. Bryant, R.E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35:8, 677–691.
2. Chavira, M. & Darwiche, A. (2007) Compiling Bayesian Networks Using Variable Elimination. In M. Veloso (ed.), *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (pp. 2443–2449). AAAI Press.
3. De Raedt, L., & Kersting, K. (2003). Probabilistic logic learning. *SIGKDD Explorations*, 5:1, 31–48.
4. De Raedt, L., Kimmig, A., & Toivonen, H. (2007). ProbLog: A Probabilistic Prolog and its Application in Link Discovery. In M. Veloso (ed.), *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (pp. 2468–2473). AAAI Press.
5. Flach, P.A. (1994). *Simply Logical: Intelligent Reasoning by Example*. John Wiley.
6. Fuhr, N. (2000). Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science*, 51, 95–110.
7. Getoor, L., & Taskar, B. (2007) editors, *Statistical Relational Learning*. MIT Press.
8. Koppel, M., Feldman, R., & Segre, A.M. (1994). Bias-Driven Revision of Logical Domain Theories. *Journal of Artificial Intelligence Research*, 1, 159–208.
9. Minato, S., Satoh, K., & Sato, T. (2007) Compiling Bayesian Networks by Symbolic Probability Calculation Based on Zero-suppressed BDDs. In M. Veloso (ed.), *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (pp. 2550–2555). AAAI Press.
10. Muggleton S. H (1996). Stochastic logic programs. In L. De Raedt (ed.), *Advances in Inductive Logic Programming*. IOS Press.
11. Perez-Iratxeta, C., Bork, P., & Andrade, M.A. (2002). Association of genes to genetically inherited diseases using data mining. *Nature Genetics*, 31, 316–319.
12. Poole, D. (1992). Logic programming, abduction and probability. *Fifth Generation Computing Systems*, 530–538.
13. Poole, D. (1993). Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64, 81–129.
14. Sato, T. & Kameya, Y. (2001). Parameter learning of logic programs for symbolic-statistical modeling. *Journal of AI Research*, 15, 391–454.
15. Sevón, P., Eronen, L., Hintsanen, P., Kulovesi, K., & Toivonen, H. (2006). Link discovery in graphs derived from biological databases. In U. Leser, F. Naumann, and B. Eckman (Eds.), *Data Integration in the Life Sciences 2006*, volume 4075 of *LNBI*. Springer.
16. Valiant, L.G. (1979). The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8, 410– 411.
17. Wrobel, S. (1996). *First Order Theory Refinement*. In L. De Raedt (ed.), *Advances in Inductive Logic Programming*. IOS Press.
18. Zelle, J.M & Mooney, R.J. (1994). Inducing Deterministic Prolog Parsers From Treebanks: A Machine Learning Approach. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)* (pp. 748–753).