KATHOLIEKE UNIVERSITEIT
LEUVEN

**Arenberg Doctoral School of Science, Engineering & Technology**
Faculty of Engineering
Department of Computer Science

# A Probabilistic Prolog and its Applications

**Angelika Kimmig**

Dissertation presented in partial
fulfillment of the requirements for
the degree of Doctor
in Engineering

November 2010

# A Probabilistic Prolog and its Applications

**Angelika Kimmig**

Jury:
Prof. Dr. ir. Ann Haegemans, president
Prof. Dr. Luc De Raedt, promotor
Prof. Dr. ir. Maurice Bruynooghe
Prof. Dr. ir. Erik Duval
Prof. Dr. ir. Gerda Janssens
Prof. Dr. Taisuke Sato
(Tokyo Institute of Technology, Japan)
Prof. Dr. Vítor Santos Costa
(Universidade do Porto, Portugal)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

November 2010

D/2010/7515/121
ISBN 978-94-6018-282-2

# Abstract

One of the key challenges in artificial intelligence is the integration of machine learning, relational knowledge representation languages and reasoning under uncertainty. Given its multi-disciplinary nature, this topic has been approached from different angles, leading to a very active research area known as probabilistic logic learning or statistical relational learning.

Motivated by the need for a probabilistic logic language with an implementation that supports reasoning in large networks of uncertain links, as they arise for instance when integrating information from various databases, this thesis introduces ProbLog, a simple extension of the logic programming language Prolog with independent random variables in the form of probabilistic facts. While ProbLog shares its distribution semantics with other approaches developed in the field of probabilistic logic learning, its implementation has been the first to allow for scalable inference in networks of probabilistic links. This is due to the use of advanced data structures that make it possible to base probabilistic inference on proofs or explanations even if those are not mutually exclusive in terms of the possible worlds they cover.

As a general purpose probabilistic logic programming language, ProbLog provides a framework for lifting relational learning approaches to the probabilistic context. We apply this methodology to obtain three new probabilistic relational learning techniques. The first one, theory compression, is a form of theory revision that reduces a ProbLog program to a given maximum size by deleting probabilistic facts, using example queries that should or should not be provable with high probability as a guideline. The second example, probabilistic explanation based learning, naturally extends explanation based learning to a probabilistic setting by choosing the most likely proof of an example for generalization. The third approach, probabilistic query mining, combines multi-relational data mining with scoring functions based on probabilistic databases. The latter two techniques are also used to illustrate the application of ProbLog as a framework for reasoning by analogy, where the goal is to identify examples with a high probability of being covered by logical queries that also assign high probability to given training examples. While these approaches all rely on ProbLog programs with known probability labels, we

also introduce a parameter estimation technique for ProbLog that determines those labels based on examples in the form of queries or proofs augmented with desired probabilities. Throughout the thesis, we experimentally evaluate the methods we develop in the context of a large network of uncertain information gathered from biological databases.

While the main focus of this thesis lies on ProbLog and its applications, we also contribute a second framework called $\omega$ProbLog, which generalizes ProbLog to weight labels from an arbitrary commutative semiring. We introduce inference algorithms that calculate weights of derived queries, showing that the ideas underlying the ProbLog system carry over to more general types of weights.

# Acknowledgements

Working on this Ph.D. has been a fun and exciting journey that would not have been possible without the support of many people. First of all, I'd like to thank my supervisor Luc De Raedt. His enthusiasm – together with that of Kristian Kersting – has guided me from that first seminar on probabilistic logic learning via my diploma thesis right into this Ph.D. adventure. I sometimes still wonder how exactly they managed to get me interested in probabilities... But they certainly did! From the beginning, Luc has been a true expert in both motivating and challenging me with a single word or question. I have learned a lot from working with him, often without even realizing it. Moreover, he has also given me the opportunity to explore a new country, and to practice juggling with several (natural) languages on a daily basis. Bedankt, Luc!

Besides Luc, I've had the chance to work with many others, both within the ML groups in Freiburg and Leuven and beyond. Back in Freiburg, Hannu Toivonen, Kristian Kersting and Kate Revoredo have made the initial months a lot easier than I would have imagined, while Ralf Wimmer has introduced me to the practical side of binary decision diagrams. Later on, the first floor PLL island in Leuven has been a great place to work, with the ProbLog and Prolog experts Bernd Gutmann and Theofrastos Mantadelis, and with Daan Fierens quietly ensuring that the three of us wouldn't forget the rest of PLL – and keep at least some of our discussions and bug hunting fights to a bearable noise level ;-) This thesis would not have been possible without the Prolog expertise from both Leuven and Porto. I am grateful to Vítor Santos Costa, Ricardo Rocha, Bart Demoen and Gerda Janssens for joining our team. Maurice Bruynooghe and Joost Vennekens never lost their patience in explaining me the fine details of semantics. I also thank Dimitar Shterionov for letting one of my ideas take over his master thesis topic, and Fabrizio Costa for challenging me with a constant stream of ideas on how to use ProbLog. Ingo Thon and Guy Van den Broeck have provided new insights in many discussions. Finally, I'd also like to thank all the other members of ML and DTAI for their comments on my work, and for the company during conference trips and Alma lunches. Dat laatste groepje will ik ook bedanken voor de dagelijkse lessen Nederlands. I further thank Hannu, Vítor, and David Page and Jude Shavlik for welcoming me as a

visitor in their groups in Helsinki, Porto, and Madison.

This thesis builds on the work of many others, both in machine learning and logic programming. I am lucky to have four of these experts, Taisuke Sato, Vítor Santos Costa, Maurice Bruynooghe, and Gerda Janssens, serving on my Ph.D. committee, and I'd like to thank them for their excellent feedback on my work. I also thank Erik Duval for enriching the committee with a different perspective, and Ann Haegemans for chairing my defense.

I gratefully acknowledge the financial support received for the work performed during this thesis from the European Union FP6-508861 project on "Applications of Probabilistic Inductive Logic Programming II", the GOA/08/008 Probabilistic Logic Learning, and the research foundation Flanders (FWO-Vlaanderen).

Of course, there's a lot more to life than 0s and 1s, and even more than all the probabilities in between. Beyond work, I'd like to especially thank Bettina, Ralf, Hülya and Konrad for many discussions during extended teatimes and countless walks (random and planned, back home and all over Europe), and simply for their friendship throughout the years. I also thank my orchestra colleagues in both Freiburg and Leuven for regularly dragging me off my computer with evenings and weekends full of music, hard work and great fun. Finally and most importantly, I want to thank my parents and my sister for their unconditional support. Danke, daß Ihr immer für mich da seid!

Angelika Kimmig
Leuven, November 2010

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Symbols

| | |
|---|---|
| $T$ | ProbLog program |
| $BK$ | Background knowledge |
| $p_i :: f_i$ | Probabilistic fact |
| $b_i$ | Propositional variable corresponding to $p_i :: f_i$ |
| $F^T$ | Set of groundings of probabilistic facts in $T$ |
| $L^T$ | $F^T$ without probability labels |
| $I$ | Partial interpretation of $L^T$ |
| $I^1$ | Set of facts assigned true in $I$ |
| $I^0$ | Set of facts assigned false in $I$ |
| $I^?$ | Set of facts with unassigned truth value in $I$ |
| $Compl^T(I)$ | Set of complete interpretations of $L^T$ extending $I$ |
| $P^T(I)$ | Probability of interpretation $I$ given by program $T$ |
| $M_I(T)$ | Least Herbrand model of $L^T \cup BK$ extending $I$ |
| $I \models_T q$ | $I$ supports query $q$ |
| $E$ | Explanation |
| $Expl^T(q)$ | Set of explanations of query $q$ in $T$ |

| | |
|---|---|
| $D_s^T(q)$ | DNF encoding of all interpretations supporting query $q$ |
| $D_x^T(q)$ | DNF encoding of $Expl^T(q)$ |
| $D_{xs}^T(q)$ | Syntactical extension of $D_x^T(q)$ to complete interpretations |
| $P_x^T(q)$ | Explanation probability |
| $P_s^T(q)$ | Success probability |
| $P_k^T(q)$ | $k$-probability |
| $S_x^T(q)$ | Sum of probabilities |
| $P_{MAP}^T(q)$ | MAP probability |

# Overture

# Chapter 1

# Introduction

Building computer programs that solve specific types of tasks considered to require intelligence is the driving force behind current progress in the field of artificial intelligence. While the notion of intelligence is difficult to grasp in formal terms, from the point of view of solving a certain type of problem or performing a certain task, it involves abilities such as dealing with possibly large amounts of data of various types and formats as well as general knowledge about the domain, identifying those pieces of information relevant for the specific problem instance at hand, coping with situations that have not been foreseen at the time of programming, and taking into account uncertainty both in the available knowledge and in the reasoning process.

A suitable *knowledge representation* language and corresponding *reasoning* mechanisms are key components to achieve those abilities. A prominent choice are subsets of first order logic, most notably definite clause logic. Such languages make it easy to integrate heterogeneous types of knowledge about specific entities, ranging from simple attribute-value descriptions to structured or interrelated data, with general, abstract knowledge about the domain of interest. Other relational languages are widely used as well. For instance, graph languages are a popular choice for collections of binary relations such as links between web pages, citations between scientific papers, or relations between persons in social networks. Such relational languages can conveniently be emulated in definite clause logic, making it an interesting framework for general language comparisons as well. From a practical perspective, definite clause logic forms the backbone of *logic programming* [Lloyd, 1989], a field that has developed optimized algorithms and implemented corresponding reasoning systems. Inference is typically based on a form of backward chaining from a given query, which naturally focuses on the relevant parts of the database. For more details, we refer to Section 2.1.

While logic programming primarily follows a *deductive* approach to reasoning, where only explicitly available information is used, *machine learning* adds an *inductive* component, where regularities in the available data are made explicit and transferred to other instances of the same type. Formally, a computer program is said to learn if it improves its performance with respect to a class of tasks based on experience [Mitchell, 1997]. Learning often aims at obtaining a model of the data or of some process that could have generated the data. Such a model can then be used for classification of new data points, which in turn provides additional information for problem solving. Alternatively, the model itself can be analyzed to obtain new insights into the domain. This is especially true if learning uses a language understandable for human experts, such as association rules or logical theories. Finding association rules or other types of local patterns in data is also studied in the field of *data mining* as part of the process of knowledge discovery in databases [Fayyad et al., 1996]. While both machine learning and data mining traditionally worked with propositional or attribute-value data, the need to deal with more complex structured or relational data has lead to the growing subfield known as inductive logic programming (ILP) [Muggleton and De Raedt, 1994], multi-relational data mining [Džeroski and Lavrač, 2001], or logical and relational learning [De Raedt, 2008]. Many approaches developed in this field again build on the computational framework of logic programming and definite clause logic.

While logical and relational languages are rich knowledge representation tools, they are not able to explicitly deal with the uncertainty inherent in real world data and problems, whether stemming from imprecision in the data collection process or contradictory information from different sources, or simply from the fact that abstract rules often hold in general, but can have exceptions. To deal with this type of problems, relational languages need to be integrated with some mechanism to cope with uncertainty based on for instance statistical models or probability theory. A variety of such approaches and corresponding learning techniques have been developed in the field known as probabilistic logic learning (PLL) [De Raedt and Kersting, 2003], statistical relational learning (SRL) [Getoor and Taskar, 2007] or probabilistic inductive logic programming (PILP) [De Raedt and Kersting, 2004; De Raedt et al., 2008a]; we refer to Section 2.3 for a brief overview. While these languages are very expressive in general, their implementations often impose restrictions to increase efficiency, or are tailored towards specific tasks, which, despite the wide range of languages, can make it difficult to find one that is directly usable for a new task at hand.

Probabilistic logic languages originating from the areas of machine learning and knowledge representation often focus on the *modeling* aspect of the underlying logical language, even in the case of definite clause languages close to (pure) Prolog. Recently, the *programming* view on such languages is receiving increased attention. Indeed, with probabilistic semantics rooted in the semantics of logic programming and Prolog, as defined for instance by Sato's distribution

semantics [Sato, 1995], languages such as PRISM [Sato and Kameya, 2001] are general programming languages adapted for probabilistic purposes. They have the potential to extend probabilistic definite clause languages similarly to how Prolog as a programming language extends its core definite clause language. Clearly, probabilistic programming is not restricted to logic programming, but similar ideas emerge in other areas, including for instance functional languages such as IBAL [Pfeffer, 2001] and Church [Goodman et al., 2008], or Figaro [Pfeffer, 2009], which combines functional and object-oriented aspects. Such languages typically calculate point probabilities; however, probability intervals have been investigated as well, for instance in probabilistic logic programming as defined by Ng and Subrahmanian [1992].

## Thesis Contributions and Roadmap

This thesis develops and implements a probabilistic logic programming language that can represent a broad range of problems, including link mining tasks in large networks of uncertain relationships. Apart from effective and efficient reasoning algorithms, the resulting system also provides a general framework for lifting traditional ILP tasks to probabilistic ILP, as will be illustrated for a number of techniques. Link mining and reasoning in large biological networks are used as a testbed for all approaches discussed throughout the thesis.

Sato's distribution semantics [Sato, 1995] provides a thorough theoretical basis for extending logic programming or definite clause logic with independent probabilistic facts. Its basic idea is to use logic programs to model discrete probability distributions over logical interpretations. However, existing PLL systems based on the distribution semantics, such as the ones for PRISM [Sato and Kameya, 2001] and ICL [Poole, 2000], have practical limitations. The PRISM system imposes additional requirements on programs to simplify inference. Specifically, each logical interpretation can only contain one of a set of so-called observable ground atoms with a single proof or explanation. While the ICL implementation Ailog2 does not rely on such additional assumptions, it does not scale very well. In particular, these limitations prevent the application of those systems in the context of mining and analyzing large probabilistic networks, which can naturally be represented – and complemented with background knowledge – in probabilistic logic. An example of such a network is Sevon's Biomine network [Sevon et al., 2006] which contains relationships between various types of biological objects, such as genes, proteins, tissues, organisms, biological processes, and molecular functions. These relationships have been extracted from large public databases such as Ensembl and NCBI Entrez, where weights have been added to reflect uncertainty. Mining this type of data has been identified as an important and challenging task, see e.g. [Perez-Iratxeta et al., 2002], but few tools are available to support this process.

The need to overcome the above mentioned limitations of probabilistic logic languages for biological network mining has motivated our work on ProbLog, a probabilistic logic programming language based on the distribution semantics. Our inference algorithms for ProbLog are based on advanced data structures to enable probability calculation without additional assumptions. The implementation of these algorithms uses state-of-the-art Prolog technology as offered by YAP-Prolog, which drastically improves scalability in the network setting. However, as ProbLog is a general probabilistic logic programming language, it can not only be used for mining large networks, but also offers a framework to transfer techniques from inductive logic programming (ILP) to the probabilistic setting. Such probabilistic ILP approaches broaden the notion of logical coverage into a gradual one, where probabilities quantify the degree of truth. As these techniques typically require the evaluation of large amounts of queries, efficient inference methods as provided by ProbLog are crucial for their successful application. In this thesis, we employ ProbLog to develop a number of PILP techniques which either *improve* probabilistic theories with respect to given examples, or exploit probabilistic information to *reason by analogy*. Furthermore, we show that ProbLog inference can directly be adapted to domains with different types of fact labels, such as cost networks or weighted propositional logic, resulting in the introduction of $\omega$ProbLog, which generalizes ProbLog's probability labels to labels from an arbitrary commutative semiring.

The core of this thesis is divided into three main parts, discussing the ProbLog language and its implementation, machine learning techniques that improve ProbLog theories based on examples, and methods to reason by analogy using ProbLog, respectively. In all parts, we will use the Biomine network for experiments demonstrating the applicability of ProbLog techniques in real-world collections of probabilistic data. An intermezzo after the first part takes a step back and investigates the question of how to apply ProbLog techniques if probability labels are replaced by other types of weights. In the following, we give a brief overview of the main contributions of each part.

**Part I** is devoted to ProbLog's syntax and semantics, inference algorithms, and implementation. In **Chapter 3**, we lay the foundations by introducing ProbLog, an extension of Prolog where *probabilistic facts* are used to define a distribution over canonical models of logic programs, which serves as the basis to define the success probability of logical atoms or queries. The semantics of ProbLog is not new: it is an instance of Sato's well-known distribution semantics. However, in contrast to many other languages based on this semantics, ProbLog is targeted at efficient and scalable inference without making any assumptions beyond independence of basic random variables. To this aim, **Chapter 4** contributes various algorithms for exact and approximate inference in ProbLog. Our implementation of ProbLog on top of the state-of-the-art YAP-Prolog system uses binary decision diagrams (BDDs) to efficiently calculate probabilities. To the best of our knowledge, ProbLog has been

the first PLL system using BDDs, an approach that currently receives increasing attention in the fields of probabilistic logic learning and probabilistic databases, cf. for instance [Riguzzi, 2007; Ishihata et al., 2008; Olteanu and Huang, 2008; Thon et al., 2008; Riguzzi, 2009]. The techniques exploited in our implementation enable the use of ProbLog to effectively query Sevon's Biomine network [Sevon et al., 2006] containing about 1,000,000 nodes and 6,000,000 edges. ProbLog is included in the publicly available stable version of YAP.[1] Part I is based mainly on the following publications:

> L. De Raedt, A. Kimmig, and H. Toivonen. *ProbLog: A probabilistic Prolog and its application in link discovery*, in Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI–2007), Hyderabad, India, 2007.
>
> A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt. *On the efficient execution of ProbLog programs*, in Proceedings of the 24th International Conference on Logic Programming (ICLP–2008), Udine, Italy, 2008.
>
> A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. *On the Implementation of the Probabilistic Logic Programming Language ProbLog*, Theory and Practice of Logic Programming, accepted, 2010.

Before moving on to learning and mining techniques for ProbLog, the **Intermezzo** in **Chapter 5** introduces $\omega$ProbLog, a generalization of ProbLog where probability labels are replaced by semiring weight labels, together with a set of algorithms that correspondingly extend ProbLog's inference algorithms.

In **Part II**, we discuss machine learning techniques that improve ProbLog programs with respect to a set of example queries. **Chapter 6** introduces the task of *theory compression*, where the size of a ProbLog database is reduced based on queries that should or should not have a high success probability. It is a form of theory revision where the only operation allowed is the deletion of probabilistic facts, which is evaluated in terms of the effect on the probabilities of the example queries. While deleting a fact can also be seen as setting its probability to 0, *parameter learning* as discussed in **Chapter 7** allows for arbitrary fine-tuning of probabilities. In this regard, we introduce a novel setting for parameter learning in probabilistic databases, which differs from the common setting of parameter learning for generative models, as such databases do not define a distribution over example queries. We provide a parameter estimation algorithm based on a gradient descent method, where examples are labeled with their desired probability. The approach integrates learning from entailment and learning from proofs, as examples can be provided in the form of both queries and proofs. The methods discussed in

[1] `http://www.dcc.fc.up.pt/~vsc/Yap/`

this part directly exploit the BDDs generated by ProbLog's inference engine to efficiently evaluate the effect of possible changes. The material presented in Part II has been published in the following main articles:

> L. De Raedt, K. Kersting, A. Kimmig, K. Revoredo, and H. Toivonen. *Compressing probabilistic Prolog programs*, Machine learning, 70(2-3), 2008.
>
> B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. *Parameter learning in probabilistic databases: A least squares approach*, in Proceedings of the 19th European Conference on Machine Learning (ECML–2008), Antwerp, Belgium, 2008.

**Part III** focuses on explanation learning for reasoning by analogy. Explanation learning is concerned with identifying an abstract explanation with maximal probability on given example queries. Such an explanation can then be used to retrieve analogous examples and to rank them by probability. In **Chapter 8**, we introduce two alternative approaches to explanation learning. In *probabilistic explanation based learning* (PEBL), the problem of multiple explanations as encountered in classical explanation based learning is resolved by choosing the most likely explanation. PEBL deductively constructs explanations by generalizing the logical structure of the most likely proofs of example queries in a domain theory defining a target predicate. *Probabilistic local query mining* extends existing multi-relational data mining techniques to probabilistic databases. It thus follows an inductive approach, where the pattern language is defined by means of a language bias and the search for patterns is structured using a refinement operator. Furthermore, negative examples can be incorporated in the score to find correlated patterns. The work presented in Part III has been published previously in:

> A. Kimmig, L. De Raedt, and H. Toivonen. *Probabilistic explanation based learning*, in Proceedings of the 18th European Conference on Machine Learning (ECML–2007), Warsaw, Poland, 2007. Winner of the ECML Best Paper Award (592 submissions).
>
> A. Kimmig and L. De Raedt. *Local query mining in a probabilistic Prolog*, in Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI–2009), Pasadena, California, USA, 2009.

Finally, some of the work performed during my Ph.D. research has not been included in this text, but will be briefly summarized in Chapter 9 in the context of related future work; more details can be found in [Kimmig and Costa, 2010] and [Bruynooghe et al., 2010].

# Chapter 2

# Foundations

In this chapter, we review a number of important concepts used throughout this thesis. We start in Section 2.1 with logic programming. Section 2.2 presents Sato's distribution semantics, which extends logic programming with probabilistic facts. Section 2.3 provides a brief overview of probabilistic logic learning. Binary decision diagrams as introduced in Section 2.4 are one of the key ingredients of our efficient implementation of ProbLog. Finally, in Section 2.5, we briefly discuss Sevon's Biomine network as an example application to be used in our experiments.

## 2.1 Logic Programming

In this section, we will briefly review the basic concepts of logic programming. For more details, we refer to [Lloyd, 1989; Flach, 1994]. We use Prolog's notational conventions, i.e. variable names start with an upper case letter, names of all other syntactic entities with lower case letters.

**Example 2.1** *Using successor notation, the following program defines the set of natural numbers as well as a relation* smaller *among them.*

```
nat(0).
nat(s(X)) :- nat(X).

smaller(0,s(X)).
smaller(s(X),s(Y)) :- smaller(X,Y).
```

In the example, `0` is the only *constant*, `X` and `Y` are *variables*. The *structured term* `s(X)` is obtained by combining the *functor* `s/1` of *arity* 1 and the term `X`. Constants, variables and structured terms obtained by combining $n$ terms and a functor of arity $n$ are called *terms*. *Atoms*, such as `nat(0)` or `smaller(s(X),s(Y))`, consist of an $n$-ary *predicate* (`nat/1` or `smaller/2` in this case) with $n$ terms as arguments. Atoms and their negations, e.g. `not(nat(0))`, are also called *positive* and *negative literals*, respectively.

A *definite clause* is a formula of the form $h \vee \neg b_1 \vee \ldots \vee \neg b_n$, where $h$ and all $b_i$ are atoms. In Prolog, such a definite clause is written as

$$\mathtt{h} \ :- \ \mathtt{b_1}, \ldots, \mathtt{b_n}.$$

where `h` is called the *head* and $\mathtt{b_1}, \ldots, \mathtt{b_n}$ the *body* of the definite clause. Informally, it is read as "If all body atoms are true, the head atom is true as well". In *normal clauses*, the body is a conjunction of literals. All variables in clauses are (implicitly) universally quantified. If the body contains the single constant *true*, it is omitted, as for `nat(0)` in the example, and such clauses are called *facts*. A *definite clause program* or *logic program* for short is a finite set of definite clauses. A *normal logic program* is a finite set of normal clauses. The set of all clauses in a (normal) logic program with the same predicate in the head is called the *definition* of this predicate. We first focus on definite clause programs and discuss normal logic programs later.

A term or clause is *ground* if it does not contain variables. A *substitution* $\theta = \{V_1/t_1, \ldots, V_m/t_m\}$ assigns terms $t_i$ to variables $V_i$. Applying a substitution to a term or clause means replacing all occurrences of $V_i$ by $t_i$.

**Example 2.2** *Applying* $\theta = \{\mathtt{X}/0, \mathtt{Y}/\mathtt{s}(0)\}$ *to* $t = \mathtt{smaller}(\mathtt{s}(\mathtt{X}), \mathtt{s}(\mathtt{Y}))$ *results in* $t\theta = \mathtt{smaller}(\mathtt{s}(0), \mathtt{s}(\mathtt{s}(0)))$.

Two terms (or clauses) $t_1$ and $t_2$ can be *unified* if there exist substitutions $\theta_1$ and $\theta_2$ such that $t_1\theta_1 = t_2\theta_2$. A substitution $\theta$ is the *most general unifier* $mgu(a, b)$ of atoms $a$ and $b$ if and only if $a\theta = b\theta$ and for each substitution $\theta'$ such that $a\theta' = b\theta'$, there exists a substitution $\gamma$ such that $\theta' = \theta\gamma$ and $\gamma$ maps at least one variable to a term different from itself.

The *Herbrand base* of a logic program is the set of ground atoms that can be constructed using the predicates, functors and constants occurring in the program[1]. Subsets of the Herbrand base are called *Herbrand interpretations*. A Herbrand interpretation is a *model* of a clause $\mathtt{h} \ :- \ \mathtt{b_1}, \ldots, \mathtt{b_n}.$ if for every substitution $\theta$ such that all $b_i\theta$ are in the interpretation, $h\theta$ is in the interpretation as well. It is a model of a logic program if it is a model of all clauses in the program. The model-theoretic semantics of a definite clause program is given by its smallest Herbrand

[1] If the program does not contain constants, one arbitrary constant is added.

model with respect to set inclusion, the so-called *least Herbrand model*. The least Herbrand model can be generated iteratively starting from the groundings of all facts in the program and adding the head $\mathtt{h}\theta$ of each clause $(\mathtt{h} \ :- \ \mathtt{b}_1, \ldots, \mathtt{b}_n)\theta$ for which all $\mathtt{b}_i\theta$ are already known to be true until no further atoms can be derived. We say that a logic program $P$ *entails* an atom $a$, denoted $P \models a$, if and only if $a$ is true in the least Herbrand model of $P$.

**Example 2.3** *The Herbrand base hb of the definite clause program in Example 2.1 contains all atoms that can be built from predicates* `nat/1`, `smaller/2`, *functor* `s/1` *and constant* `0`, *that is,*

$$\begin{aligned} hb = \{&\mathtt{nat}(\mathtt{0}), \mathtt{smaller}(\mathtt{0},\mathtt{0}), \mathtt{nat}(\mathtt{s}(\mathtt{0})), \mathtt{smaller}(\mathtt{0},\mathtt{s}(\mathtt{0})), \mathtt{smaller}(\mathtt{s}(\mathtt{0}),\mathtt{0}), \\ &\mathtt{smaller}(\mathtt{s}(\mathtt{0}),\mathtt{s}(\mathtt{0})), \mathtt{nat}(\mathtt{s}(\mathtt{s}(\mathtt{0}))), \mathtt{smaller}(\mathtt{0},\mathtt{s}(\mathtt{s}(\mathtt{0}))), \ldots\} \end{aligned}$$

*It also is a non-minimal Herbrand model of the program. The least Herbrand model is the subset of the Herbrand base containing all atoms for* `nat/1` *as well as those for* `smaller/2` *whose first argument contains less occurrences of* `s/1` *than the second.*

The main inference task of a logic programming system is to determine whether a given atom, also called *query*, is true in the least Herbrand model of a logic program. In our example, the query `smaller(0,s(0))` has answer *yes*, while `smaller(0,0)` has answer *no*, in which case we also say that the query *fails*. If such a query is not ground, inference asks for the existence of an *answer substitution*, that is, a substitution that grounds the query into an atom that is part of the least Herbrand model. For example, {`X/0`} is an answer substitution for query `nat(X)`.

Prolog answers queries using *refutation*, that is, the negation of the query is added to the program and resolution is used to derive the empty clause. More specifically, *SLD-resolution* takes a *goal* of the form

$$?- \ \mathtt{g}, \mathtt{g}_1, \ldots, \mathtt{g}_n,$$

a clause

$$\mathtt{h} \ :- \ \mathtt{b}_1, \ldots, \mathtt{b}_m$$

such that `g` and `h` unify with most general unifier $\theta$, and produces the resolvent

$$?- \ \mathtt{b}_1\theta, \ldots, \mathtt{b}_m\theta, \mathtt{g}_1\theta, \ldots, \mathtt{g}_n\theta.$$

This process, which continues until the empty goal is reached, can be depicted by means of an *SLD-tree*. The root of such a tree corresponds to the query, each branch to a *derivation*, that is, a sequence of resolution steps. Derivations ending in the empty clause are also called *proofs*.

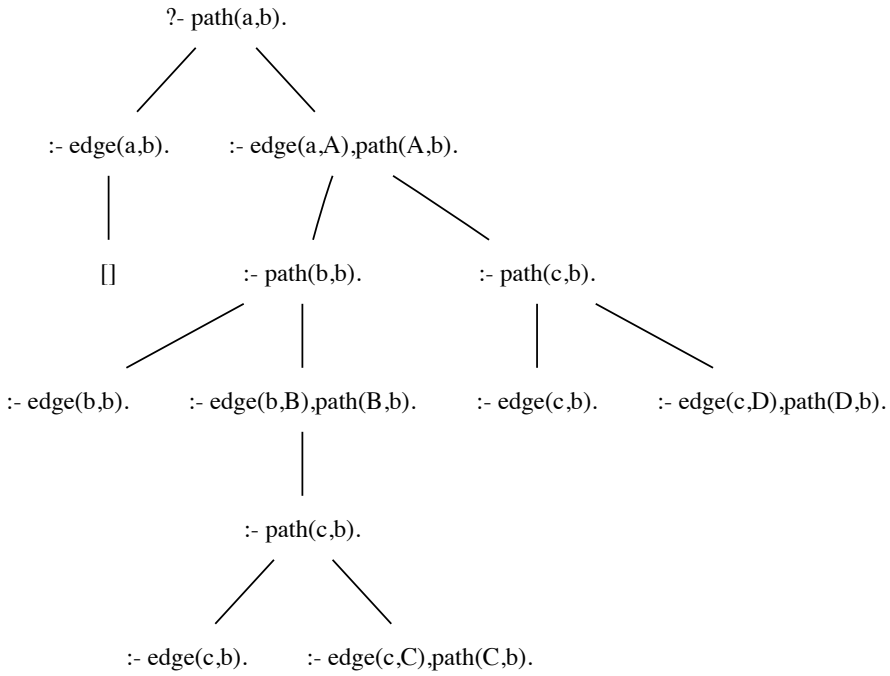



Figure 2.1: SLD-tree for query `path(a,b)` in Example 2.4.

**Example 2.4** *The following program encodes a graph with three nodes and defines paths between nodes in terms of edges.*

```
edge(a,b).      edge(a,c).      edge(b,c).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

*Figure 2.1 shows the SLD-tree for query* `path(a,b)`*, where the empty clause is depicted by* [].

By default, Prolog uses depth-first search to traverse the SLD-tree during proving, meaning that it can get trapped in infinite loops; however, this can be avoided by using alternative search strategies such as iterative deepening. *Backtracking* forces the proving mechanism to undo previous steps to find alternative solutions. For example, `nat(X)` first returns answer substitution {`X/0`} as said above, but on backtracking will also produce {`X/s(0)`}, {`X/s(s(0))`}, and so forth.

Normal logic programs use the notion of *negation as failure*, that is, for a ground atom $a$, $not(a)$ is true exactly if $a$ cannot be proven in the program. They are not guaranteed to have a unique minimal Herbrand model. Various ways to define the canonical model of such programs have been studied; here, we follow the *well-founded semantics* of Van Gelder et al. [1991]. It uses three-valued logic and partial models, that is, if the truth value of a literal is not determined by the program, it is considered undefined. Similarly to the least Herbrand model of definite clause programs, the well-founded model can be constructed iteratively by considering all clauses whose body is true in the current partial interpretation. However, during this construction, certain literals are also inferred to be false. The underlying idea is to set a set of literals to false if this makes it impossible to derive the value true for any of them. This is formalized using the concept of unfounded sets.

Given a normal logic program $P$ with Herbrand base $hb$ and a partial interpretation $I$, a subset $A \subseteq hb$ is *unfounded* if for each atom $a \in A$ and each grounded rule $a :- b_1, \ldots, b_n$ in $P$, some positive or negative body literal $b_i$ is false in $I$ or some positive body literal $b_i$ occurs in $A$. In the first case, the rule body is false in the current partial interpretation and thus also in all interpretations that could be obtained by specifying truth values for additional literals, meaning that the clause cannot be used to derive $a$. In the second case, using the clause to derive $a$ would require the positive literal $b_i$ to be true. However, if we simultaneously set all literals in $A$ to false, the bodies of all such clauses are false and they could thus not be used to derive any literal in $A$ as true.

The iterative construction of the well-founded model extends the current interpretation $I_k$ into $I_{k+1}$ using two steps. First, for each ground rule whose body is true in $I_k$, the head is set true in $I_{k+1}$. Second, all atoms in the greatest unfounded set with respect to $I_k$ are set false in $I_{k+1}$. These steps are repeated until the least fixed point is reached.

The well-founded model has been shown to be two-valued for several restricted classes of normal logic programs, including stratified and locally stratified programs. A program $P$ is *stratified* if and only if each of its predicates $p$ can be assigned a rank $j$ such that it only depends positively on predicates of rank at most $j$ and negatively on predicates of rank at most $j - 1$. It is *locally stratified* if all atoms $a$ in its Herbrand base can be assigned a rank $j$ such that for any grounded rule $a :- b_1, \ldots, b_m$, the rank of positive literals $b_i$ is at most $j$, that of negative ones at most $j - 1$.

Prolog uses *SLDNF-resolution*, a combination of SLD-resolution with negation as finite failure, for inference in normal logic programs. Negated atoms are commonly required not to *flounder*, that is, their variables need to be bound on calling.

## 2.2 Distribution Semantics

The distribution semantics as rigorously defined by Sato [1995] provides a formal basis for extending logic programming with probabilistic elements. It is a generalization of the least Herbrand model semantics, where the main difference is that logic programs contain a set of dedicated facts whose truth values are not directly set to *true* as in the least Herbrand model of a usual logic program, but determined probabilistically. Once these truth values are fixed, the program again has a unique least model extending the partial interpretation, which of course can be different depending on the initially chosen assignments. The distribution semantics now defines a distribution over these least Herbrand models of the program by extending a joint probability distribution over the set of dedicated facts. In its basic form, where the joint distribution is defined using a set of independent random events, it is a well-known semantics for probabilistic logics that has been (re)defined multiple times in the literature, often under other names or in a more limited database setting; cf. for instance [Dantsin, 1991; Poole, 1993b; Fuhr, 2000; Poole, 2000; Dalvi and Suciu, 2004]. Sato has, however, formalized a more general setting, including the case of a countably infinite set of random variables and using arbitrary discrete distributions over these basic random variables, in his well-known distribution semantics. We briefly repeat the basic ideas in the following; for more details, the interested reader is referred to [Sato, 1995].

We assume a first order language with denumerably many predicate, constant and functor symbols. Let $DB = F \cup R$ be a definite clause program, where $F$ is a set of unit clauses, called *facts*, and $R$ is a set of (possibly non-unit) clauses, called *rules*. For simplicity, it is assumed that $DB$ is ground and denumerably infinite, and no fact in $F$ unifies with the head of a rule in $R$. The distribution semantics can be viewed as a possible worlds semantics, where ground atoms are treated as random variables, and worlds thus correspond to interpretations assigning truth values to all ground atoms in $DB$.

The key idea of the distribution semantics is to extend a *basic distribution* $P_F$ over subsets or interpretations $F' \subseteq F$ into a distribution $P_{DB}$ over the least Herbrand models of $DB$, exploiting the uniqueness of the least Herbrand model of $F' \cup R$ for each such $F'$. We first illustrate this for the finite case by means of an example.

**Example 2.5** *Given the definite clause program $DB = F \cup R$ with*

$$F = \{a(0), a(1)\}$$

$$R = \{(b(0) : -a(0)), (b(1) : -a(1), b(0))\}$$

*we enumerate ground atoms in $F$ and $DB$ as $\langle a(0), a(1)\rangle$ and $\langle a(0), b(0), a(1), b(1)\rangle$, respectively. This allows us to denote interpretations as binary vectors, where the $i$-th bit denotes the truth value of the $i$-th atom in the corresponding enumeration.*

*Based on this notation, we define the basic distribution* $P_F$ *over* $\Omega_F = \{0,1\}^2$ *as*

$$P_F(00) = 0.21 \quad P_F(01) = 0.04 \quad P_F(10) = 0.58 \quad P_F(11) = 0.17$$

$P_F$ *is now extended to a distribution* $P_{DB}$ *over* $\Omega_{DB} = \{0,1\}^4$ *by setting*

$$P_{DB}(\hat{\omega}) = P_F(\omega)$$

*if* $\hat{\omega}$ *corresponds to the least Herbrand model of* $DB$ *extending* $\omega$*, and* $P_{DB}(\hat{\omega}) = 0$ *otherwise, that is*

$$P_{DB}(0000) = 0.21 \quad P_{DB}(0010) = 0.04$$

$$P_{DB}(1100) = 0.58 \quad P_{DB}(1111) = 0.17$$

For an arbitrary sentence $G$ using the vocabulary of $DB$ we define the set of possible worlds $\hat{\omega} \in \Omega_{DB}$ where $G$ is true as

$$[G] = \{\hat{\omega} \in \Omega_{DB} \mid \hat{\omega} \models G\}.$$

Given a distribution $P_{DB}$ over $\Omega_{DB}$, the probability of $G$ is defined as the probability of the set $[G]$, which in the finite case is

$$P_{DB}([G]) = \sum_{\hat{\omega} \in [G]} P_{DB}(\hat{\omega}) \tag{2.1}$$

**Example 2.6** *Continuing our example, the probability of* $b(0)$ *is*

$$P_{DB}([b(0)]) = P_{DB}(\{1100, 1111\}) = 0.58 + 0.17 = 0.75,$$

*while that of* $\forall x.b(x)$ *is*

$$P_{DB}([\forall x.b(x)]) = P_{DB}([b(0) \wedge b(1)]) = P_{DB}(\{1111\}) = 0.17.$$

While for finitely many basic facts, $P_F$ and thus $P_{DB}$ can be defined by exhaustive enumeration of $\Omega_F$, this is no longer possible for infinite $F$. Sato showed how to define $P_{DB}$ based on a series of finite distributions $P_F^{(n)}$ over interpretations $\omega_n$ of the first $n$ variables in $F$. For this to be possible, these distributions have to satisfy the *compatibility condition*, that is

$$P_F^{(n)}(\omega_n) = P_F^{(n+1)}(\omega_n 1) + P_F^{(n+1)}(\omega_n 0) \tag{2.2}$$

Intuitively, this condition ensures that if a sentence $G$ satisfies the *finite support condition*, that is, there are finitely many minimal subsets $F' \subseteq F$ such that $F' \cup R \models G$, we can fix a suitable enumeration of $F$ and restrict probability calculations to a finite prefix of this enumeration covering all facts appearing in these minimal subsets. We do not go into further technical detail here, but instead illustrate one basic and popular choice of such distributions $P_F^{(n)}$ by means of an example.

**Example 2.7** *We extend our example switching to successor notation for natural numbers.*

$$F = \{a(0), a(s(0)), a(s(s(0))), a(s(s(s(0)))), \ldots\}$$

$$R = \{(b(0) : -a(0)), (b(s(N)) : -a(s(N)), b(N))\}$$

*The basic sample space is now* $\Omega_F = \{0,1\}^\infty$*, that is, the space of countably infinite Boolean vectors. We fix enumerations for ground atoms in* $F$ *and* $DB$ *extending the ones used above, that is, following the order of arguments and iterating between* $a$ *and* $b$ *in the case of* $DB$*. Again, an interpretation of* $F$*, for example* $\omega = 110^\infty$*, leads to a unique model of* $DB$*, in this case* $\hat{\omega} = 11110^\infty$*.*

*We consider all random variables corresponding to ground facts in* $F$ *to be mutually independent, and assign a probability of being true to each of them. For the sake of simplicity, we use the same probability* $p$ *for each fact. Consider now a finite prefix* $\omega_n$ *of an interpretation* $\omega \in \Omega_F$*, where* $m$ *variables are assigned* 1*. Given the independence assumption, the joint probability of the first* $n$ *random variables taking value* $\omega_n$ *is thus*

$$P_F^{(n)}(\omega_n) = p^m \cdot (1-p)^{n-m}.$$

*Clearly, this series of distributions respects the compatibility condition of Equation* (2.2)*. To calculate the probability of* $b(s(s(0))$ *in our example, it is sufficient to use* $P_F^{(3)}$*, as the first three elements of* $F$ *already determine the truth value of the query, that is*

$$P_{DB}([b(s(s(0)))]) = P_{DB}(\{\hat{\omega} \in \Omega_{DB} | \hat{\omega}_6 = 111111\}) = P_F^{(3)}(111) = p^3$$

Finally, let us remark that the key to the distribution semantics is the existence of a unique canonical model of the entire program given an interpretation of the basic facts. While in the original distribution semantics, $R$ is a definite clause program and thus has a unique least Herbrand model, it is equally possible to use the well-founded semantics as discussed in Section 2.1, but parameterized by the set of basic facts, and restrict the set of rules $R$ in such a way that for each two-valued interpretation of the basic facts, the well-founded model of $DB$ is two-valued as well. In this view, $R$ is closely related to the definitions in FO(ID) [Denecker and Vennekens, 2007; Vennekens et al., 2009], but restricts rule bodies to conjunctions of literals instead of arbitrary first order formulae.

## 2.3 Probabilistic Logic Learning

The core concept of statistical relational learning (SRL) or probabilistic logic learning (PLL) is the combination of machine learning, statistical techniques and

reasoning in first order logic. Many variants of this theme have been studied, differing both in their logical and in their probabilistic language. Here, we will distinguish two main streams by means of the basic probabilistic framework they employ. We start with the framework we will use throughout this thesis, namely the addition of *independent probabilistic alternatives* to relational languages, and afterwards discuss relational extensions of *graphical models*, which encode dependencies between random variables by means of their underlying graphical structure.

### 2.3.1 Using Independent Probabilistic Alternatives

A *probabilistic alternative*[2] is a basic random event with a finite number of different outcomes, such as tossing a coin or rolling a die. Sets of mutually independent probabilistic alternatives are commonly used to define joint distributions over such events. A simple probabilistic model following this idea are *probabilistic context free grammars* (PCFGs) [Manning and Schütze, 1999]. Formally, a PCFG is a tuple $(\Sigma, N, S, R)$ where $\Sigma$, the *alphabet* of the language defined by the grammar, is a finite set of symbols called *terminal symbols*, $N$ is a finite set of so-called *nonterminal* symbols, $S \in N$ is the designated *start symbol*, $R$ a set of rules of the form $P : A \to \beta$ with *left hand side* $A \in N$ and *right hand side* $\beta \in (\Sigma \cup N)^*$, that is, a finite sequence of symbols from $\Sigma \cup N$, where $\epsilon$ denotes the empty sequence, and $P \in [0, 1]$ such that the sum over all rules in $R$ with the same left hand side $A$ is 1. As common for such grammars, we denote terminal and non-terminal symbols by lower and upper case letters, respectively, and simply write a PCFG as the set of rules $R$ with start symbol $S$, leaving $\Sigma$ and $N$ implicit. Sentences are derived starting from $S$ by replacing the leftmost nonterminal symbol $A$ in the current intermediate sentence by some $\beta$ with $P : A \to \beta \in R$, until no more replacements are possible, where replacement by $\epsilon$ corresponds to simply deleting $A$. The choice of rule for given $A$ is governed by the probability distribution over $A$'s rules given by their labels $P$, and is independent of everything else, including replacements of further occurrences of $A$. Thus, the independent probabilistic alternatives of PCFGs are the choices of rules during derivations, and the probability of a derivation is given as the product of the probability of all its rule applications. Furthermore, the probability of a sentence $\omega \in (\Sigma \cup N)^*$ is the sum of probabilities of all derivations ending in $\omega$.

[2] terminology inspired by [Poole, 2000]

**Example 2.8** *The following grammar defines a probability distribution over all finite non-empty strings over the alphabet* $\{a, b\}$.

$$\begin{array}{lll} 0.3 : S \to aX & 0.5 : X \to aX & 0.6 : Y \to aX \\ 0.7 : S \to bY & 0.1 : X \to bY & 0.2 : Y \to bY \\ & 0.4 : X \to \epsilon & 0.2 : Y \to \epsilon \end{array} \tag{2.3}$$

*Note that the grammar does not contain ambiguities, that is, each sentence can be obtained by a single derivation only. For instance, the sentence aab is generated by the derivation*

$$S \xrightarrow{0.3} aX \xrightarrow{0.5} aaX \xrightarrow{0.1} aabY \xrightarrow{0.2} aab$$

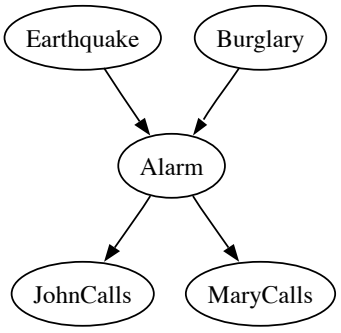
*and thus has probability*

$$0.3 \cdot 0.5 \cdot 0.1 \cdot 0.2 = 0.003.$$

Stochastic Logic Programs (SLPs) [Muggleton, 1995] directly upgrade the idea of PCFGs to definite clauses, that is, instead of probability distributions over all rules with the same left hand side, they use probability distributions over all definite clauses with the same head predicate. Further probabilistic logic languages using independent probabilistic alternatives include the probabilistic logic programs of Dantsin [1991], PHA and ICL [Poole, 1993b, 2000], probabilistic Datalog [Fuhr, 2000], PRISM [Sato and Kameya, 2001], LPADs and CP-logic [Vennekens et al., 2004; Vennekens, 2007] and ProbLog as presented in Chapter 3 of this thesis; we will discuss this group of languages in more detail in Section 3.4. While most other formalisms use rule-based logical languages, FOProbLog [Bruynooghe et al., 2010] combines arbitrary first order formulae with independent probabilistic alternatives.

### 2.3.2 Using Graphical Models

While the probabilistic languages discussed in the previous section define joint distributions in terms of mutually independent random variables, in *Bayesian Networks* (BNs) [Pearl, 1988], a joint probability distribution over a finite set of random variables with finite domains is defined in terms of a conditional distribution for each variable given a subset of the others. More specifically, a BN is a directed acyclic graph whose nodes correspond to the random variables and whose edges represent direct dependencies between random variables. Each node in the network has an associated probability distribution over its values given the values of its *parents*, the starting nodes of the node's incoming edges. The full joint distribution over all variables is then given by the product of the individual distributions.




| P(E) | 0.002 |
|---|---|

| P(B) | 0.001 |
|---|---|

| | E B | E ¬ B | ¬ E B | ¬ E ¬ B |
|---|---|---|---|---|
| P(A) | 0.95 | 0.29 | 0.94 | 0.001 |

| | A | ¬ A |
|---|---|---|
| P(J) | 0.9 | 0.05 |

| | A | ¬ A |
|---|---|---|
| P(M) | 0.7 | 0.01 |

Figure 2.2: Bayesian network

**Example 2.9** *Figure 2.2 shows the well-known alarm Bayesian network [Pearl, 1988; Russell and Norvig, 2004], where all random variables have domain* $\{0, 1\}$*. It defines the joint distribution*

$$P(E, B, A, J, M) = P(E) \cdot P(B) \cdot P(A|E, B) \cdot P(J|A) \cdot P(M|A) \tag{2.4}$$

*For instance, the probability of* $\{E = 1, B = 0, A = 1, J = 1, M = 1\}$ *thus is*

$$P(1, 0, 1, 1, 1) = 0.002 \cdot (1 - 0.001) \cdot 0.29 \cdot 0.9 \cdot 0.7 = 0.000365$$

While Bayesian networks can be mirrored in terms of independent alternatives, as we will see in Section 3.4.1, inference for special purpose languages can directly exploit the underlying independencies.

Relational extensions of Bayesian networks typically specify the graph structure at an abstract level in some relational language and use this specification as a kind of template, from which concrete instances of Bayesian networks can be obtained by grounding out logical variables. Prominent examples of such extensions include Relational Bayesian Networks [Jäger, 1997], Probabilistic Relational Models [Friedman et al., 1999], CLP($\mathcal{BN}$) [Santos Costa et al., 2003], Logical Bayesian Networks [Fierens et al., 2005], Bayesian Logic Programs [Kersting and De Raedt, 2008], and P-log [Baral et al., 2009]. In contrast to these languages, Markov Logic Networks [Richardson and Domingos, 2006] are a first order variant of undirected graphical models, using weighted first order logic formulae as templates to construct Markov Networks, thereby defining probability distributions over possible worlds.

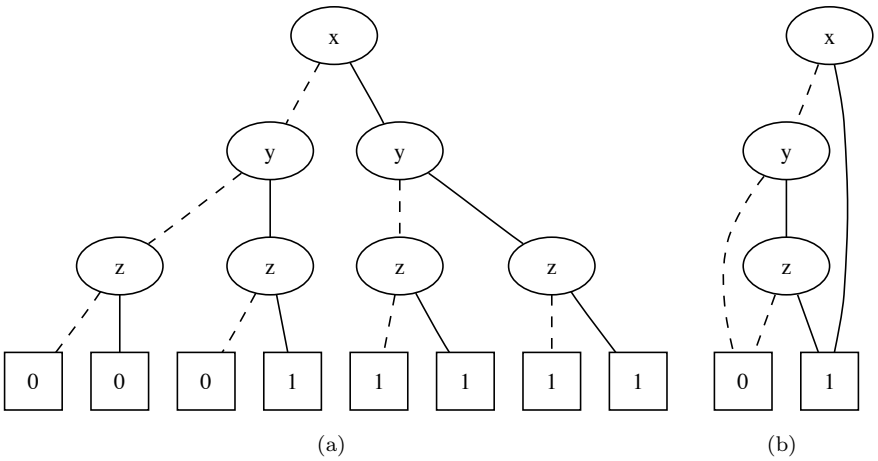



Figure 2.3: Full binary decision tree and corresponding BDD for formula $x \vee (y \wedge z)$; dotted edges correspond to value 0, solid ones to 1.

## 2.4 Binary Decision Diagrams

A binary decision diagram (BDD) [Bryant, 1986] is a data structure that graphically represents a Boolean function. Roughly speaking, a BDD is a rooted directed acyclic graph, where nodes correspond to Boolean variables, edges to truth value assignments to their source node's variable, and the two designated sink nodes, called 0- and 1-terminal node (or 0- and 1-leaf), to the function values 1 (or *true*) and 0 (or $false$), respectively. Each path through such a diagram thus encodes a truth value assignment together with the corresponding function value. While various variants of such diagrams exist, in this thesis, we will use the term BDD to refer to *reduced ordered binary decision diagrams*. As the name states, in this variant, all paths through the diagram respect the same variable *ordering*, and furthermore, the diagram is *reduced* as much as possible to achieve maximal compression. We will now discuss the basics of BDDs by means of an example.

**Example 2.10** *Consider the propositional formula $x \vee (y \wedge z)$, defining a Boolean function over three variables. Alternatively, this function could be specified by means of a truth table, that is, by listing all truth assignments to the variables together with the truth value of the formula. In Figure 2.3(a), such an explicit encoding is graphically depicted as a Boolean decision tree, where each branch corresponds to one assignment. Leaves are labeled with the truth value of the formula under the branch's assignment. All edges are implicitly directed top-down. Dotted edges denote the assignment of* 0 *to the variable of their source node, solid ones that of* 1*. Corresponding child nodes are called* low *and* high *child, respectively. The*

*leftmost branch thus assigns* 0 *to all three variables, the next one assigns* 0 *to both* $x$ *and* $y$*, but* 1 *to* $z$*, and so forth. Clearly, already for such a small example, this encoding contains redundant information. For instance, once* $x$ *is set to* 1*, the truth value of the entire formula is determined and the remaining tests listed in the corresponding subtree are unnecessary. The key idea of BDDs is to remove such redundancies by dropping nodes or sharing identical subtrees, which will transform the tree into a directed acyclic graph. For our example, this graph – which is a canonical representation given the variable ordering – is shown in Figure 2.3(b).*

Two BDDs $g_1$ and $g_2$ are *isomorphic* if there exists a one-to-one mapping $\sigma$ from edges in $g_1$ to edges in $g_2$ such that if $\sigma(s_1, t_1) = (s_2, t_2)$, the edges $(s_1, t_1)$ and $(s_2, t_2)$ are of the same type and each of the associated node pairs $(s_1, s_2)$ and $(t_1, t_2)$ shares the same label. Starting from a full binary tree with the same variable ordering on all branches, a BDD can be obtained using the following two *reduction operators*:

**Subgraph Merging** If two subgraphs $g_1$ and $g_2$ are isomorphic, all edges leading from some node outside $g_2$ to some node in $g_2$ are redirected to the corresponding node in $g_1$, and $g_2$ is removed from the graph.

**Node Deletion** If both outgoing edges of a node $n$ lead to the same node $c$, all incoming edges of $n$ are redirected to c and $n$ is removed from the graph.

**Example 2.11** *In Figure 2.3(a), the two rightmost trees with root label* $z$ *are isomorphic and can thus be merged, resulting in both outgoing edges of their parent node* $y$ *leading to the same node. Thus, this parent node can be deleted.*

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 [Chavira and Darwiche, 2007] and [Minato et al., 2007] for recent work on Bayesian networks using variants of BDDs). ProbLog is the first probabilistic logic programming system using BDDs as a basic data structure for probability calculation, a principle that receives increased interest in the fields of probabilistic logic learning and probabilistic databases, cf. for instance [Riguzzi, 2007; Ishihata et al., 2008; Olteanu and Huang, 2008; Thon et al., 2008; Riguzzi, 2009]. Since their introduction by Bryant [1986], there has been a lot of research on BDDs and their computation, leading to many variants of BDDs and off the shelf systems.

The reduction approach to BDD construction described above is clearly impractical, as it starts from an exponential encoding of the Boolean formula. However, BDDs can also be constructed by applying Boolean operators to smaller BDDs, starting with BDDs corresponding to single variables and following the structure of the

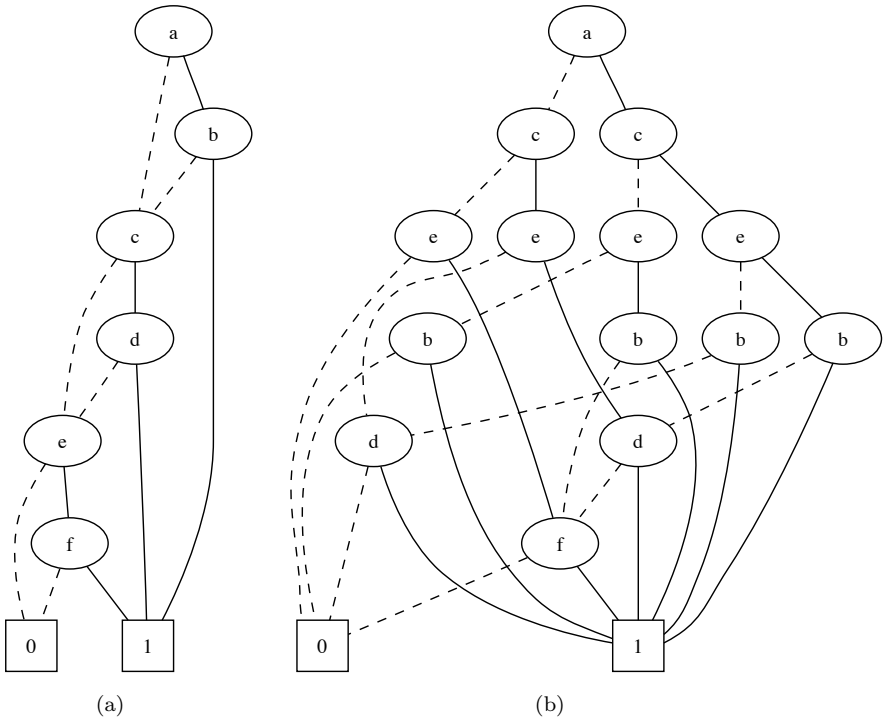



Figure 2.4: Example illustrating the effect of variable ordering on BDD size for formula $(a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$, taken from [Bryant, 1986].

formula to be encoded, where reduction operators are applied on intermediate results. Denoting the number of nodes in a BDD $g$ as $|g|$, reducing $g$ has time complexity $O(|g| \cdot \log(|g|))$, while combining $g_1$ and $g_2$ has time complexity $O(|g_1| \cdot |g_2|)$; for further details, we refer to [Bryant, 1986]. BDD tools construct BDDs following a user-defined sequence of operations. The size of a BDD is highly dependent on its variable ordering, as this determines the amount of structure sharing that can be exploited for reduction; see Figure 2.4 for an example. As computing the order that minimizes the size of a BDD is a coNP-complete problem [Bryant, 1986], BDD packages include heuristics to reduce the size by reordering variables. While reordering is often necessary to handle large BDDs, it can be quite expensive. To control the complexity of BDD construction, it is therefore crucial to aim at small intermediate BDDs and to avoid redundant steps when specifying the sequence of operations to be performed by the BDD tool.

## 2.5 Probabilistic Networks of Biological Concepts

Molecular biological data is available from public sources, such as Ensembl[3], NCBI Entrez[4], 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 [Perez-Iratxeta et al., 2002].

In the Biomine project[5], such data is viewed as a network with nodes and edges corresponding to objects and relations, respectively. Furthermore, Sevon et al. [2006] associate weights to edges, indicating the probability that the corresponding nodes are related. These weights are obtained as the product of three factors, indicating the *reliability*, the *relevance* as well as the *rarity* (specificity) of the information. Graph-based algorithms can be used to analyze and predict connections in such networks. Sevon et al. use two-terminal network reliability to estimate the strength of connection between two entities and also consider finding strongest paths. Finding reliable subgraphs involving a given set of nodes has been studied in [Hintsanen, 2007; Hintsanen and Toivonen, 2008; Kasari et al., 2010]. Further approaches to simplify such networks include methods to identify representative nodes [Langohr and Toivonen, 2009] and to prune edges whilst maintaining the quality of strongest paths between any pairs of nodes [Toivonen et al., 2010].

Combining the Biomine network with a language that makes it easy to add background knowledge and to formulate complex queries has been a key motivation for the development of our probabilistic programming language ProbLog. Throughout this thesis, we report on experiments evaluating the learning and mining techniques introduced for ProbLog in the context of the Biomine network; see Appendix A for details on the datasets.

[3] `www.ensembl.org`
[4] `www.ncbi.nlm.nih.gov/Entrez/`
[5] `http://www.cs.helsinki.fi/group/biomine/`

# Part I

# ProbLog

# Outline Part I

This part is devoted to the **probabilistic logic programming language ProbLog**, which lies at the basis of the learning and mining techniques discussed in later parts of this thesis. The development of ProbLog has been motivated by the need to combine the simple model of a probabilistic graph or database with independent edges or tuples with the deductive power of definite clause logic or, more generally, logic programming. Probabilistic graphs provide a natural framework to reason about uncertainty in collections of databases, such as the biological databases integrated in the Biomine network of Sevon et al. [2006]. Logic, on the other hand, makes it easy to reason about properties and parts of large networks, as those can be defined on an abstract level. However, as reasoning in probabilistic logics comes with a high cost, existing probabilistic logic systems often do not scale to large databases or simplify probabilistic inference by restricting the logical language to theories where atoms have mutually exclusive explanations only, which makes it impossible to e.g. query for the existence of some connection between two nodes in a probabilistic graph by simply defining a path predicate, which is one of the basic queries when exploring networks such as Biomine. The aim of ProbLog therefore is to overcome these limitations and to develop a scalable system for inference in probabilistic databases without simplifying assumptions.

In **Chapter 3**, we lay the grounds by introducing the **language** ProbLog, an extension of Prolog where probabilistic facts are used to define a distribution over canonical models of logic programs, which serves as the basis to define the success probability of logical atoms or queries. The **semantics** of ProbLog is not new: it is an instance of Sato's well-known distribution semantics. However, in contrast to many other languages based on this semantics, ProbLog is targeted at efficient and scalable inference without making any assumptions beyond independence of basic random variables. We present a reduction to DNF formulae that forms the core to inference in ProbLog, and discuss ProbLog's relationship to a number of alternative languages based on the distribution semantics.

We turn to the algorithmic side in **Chapter 4**, where we contribute various algorithms for exact and approximate **inference** in ProbLog. Furthermore, we

discuss their **implementation** on top of the state-of-the-art YAP-Prolog system, where we use binary decision diagrams (BDDs) to efficiently calculate probabilities. To the best of our knowledge, ProbLog has been the first probabilistic logic system using BDDs, an approach that currently receives increasing attention in the field. The techniques exploited in our implementation enable the use of ProbLog to effectively query Sevon's Biomine network containing about 1,000,000 nodes and 6,000,000 edges. This implementation is included in the publicly available stable version of YAP.

# Chapter 3

# The ProbLog Language*

This chapter introduces ProbLog, the probabilistic logic programming language used throughout this thesis. We start by defining the ProbLog language and its semantics in Section 3.1. Sections 3.2 and 3.3 discuss the key elements of ProbLog inference and introduce additional language concepts, respectively. In Section 3.4, we discuss other probabilistic languages using the distribution semantics.

## 3.1 ProbLog

ProbLog closely follows the ideas of the distribution semantics as summarized in Section 2.2: a set of ground facts is used to specify a basic distribution over interpretations of these facts, this distribution is extended towards interpretations including additional logical atoms by adding a set of rules, and this extended distribution is used to calculate probabilities of arbitrary logical atoms appearing in the interpretations.[1]

The following example is a simplified variant of the probabilistic graph model used for the Biomine network throughout this thesis.

**Example 3.1** *Figure 3.1 shows a small probabilistic graph that we shall use as running example in the text. In ProbLog, the graph is encoded by means of*

*This chapter builds on [De Raedt et al., 2007b; Kimmig et al., 2008, 2009, 2010]

[1]While in early work on ProbLog probabilities were attached to arbitrary definite clauses and all groundings of such a clause were treated as a single random event, we later on switched to a clear separation of logical and probabilistic part and random events corresponding to ground facts. This is often more natural and convenient, but can be used to model the original type of clauses (by adding a special ground probabilistic fact to the clause body) if desired, cf. also Section 3.3.1.

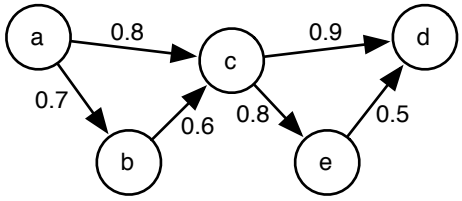



Figure 3.1: Example of a probabilistic graph, where edge labels indicate the probability that the edge is part of the graph.

*probabilistic facts:*

```
0.8 :: edge(a,c).   0.7 :: edge(a,b).   0.8 :: edge(c,e).
0.6 :: edge(b,c).   0.9 :: edge(c,d).   0.5 :: edge(e,d).
```

*For each fact, the attached label specifies the probability that the edge is actually present in the graph, that is, such a probabilistic graph can be used to sample subgraphs by tossing a coin for each edge. Such a model can for instance be used to analyze network reliability, where two nodes are able to communicate if they are connected via at least one sequence of edges. Connectivity can be expressed by the following background knowledge:*

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

*We can then ask for the probability that two nodes, say* c *and* d*, are connected in our probabilistic graph by posing query* `path(c,d)` *to the ProbLog program. This probability corresponds to 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). Denoting the random variable corresponding to* `edge(x,y)` *by* $xy$*, the set of subgraphs containing the longer path can be described by the conjunction* $ce \wedge ed$*.*

A ProbLog program thus has two parts: a set of so-called *probabilistic facts*, and a set of rules, called *background knowledge* (BK). Probabilistic facts are written as $p_i :: f_i$, where $f_i$ is a logical atom and $p_i$ the probability of a grounding of $f_i$ being assigned *true* in an interpretation, that is, if $f_i$ contains logical variables, each grounding corresponds to a different random variable. To ensure a natural interpretation of these random variables, no two different facts $f_i$, $f_j$ are allowed to unify, as otherwise, probabilities of ground facts would be higher than the individual probability given by different non-ground facts. We assume all random variables corresponding to ground probabilistic facts to be mutually independent, and define the basic distribution as their joint distribution. We denote random variables by $b_i$ and use conjunctions of corresponding literals to compactly write

(partial) interpretations of the set of ground probabilistic facts. We also view interpretations as specifications of logic programs, where the logic program contains exactly those facts whose random variables are assigned *true* in the interpretation.

A *background knowledge clause* in ProbLog is of the form

$$\mathtt{h} \ :- \ \mathtt{b_1}, \ldots \mathtt{b_n}. \tag{3.1}$$

where $\mathtt{h}$ is a positive literal not unifying with any probabilistic fact and each $\mathtt{b_j}$ is either a probabilistic fact $f_i$, the negation $not(f_i)$ of such a fact without free variables, or a positive literal not unifying with any probabilistic fact.[2] Due to these conditions, each ProbLog program is locally stratified and thus has a unique two-valued well-founded model, cf. Section 2.1. Therefore, each interpretation of the probabilistic facts can be extended into a unique minimal Herbrand model of the program. Intuitively, once the truth values of probabilistic facts are fixed in an interpretation $I$, the background knowledge could be simplified into an equivalent definite clause program by deleting body literals that are *true* in $I$ and removing clauses containing body literals that are $false$ in $I$. The minimal Herbrand model of this program then uniquely determines the truth values of remaining atoms. Alternatively, ProbLog's background knowledge can be viewed as a syntactically restricted form of the definitions in FO(ID) [Denecker and Vennekens, 2007; Vennekens et al., 2009], where the probabilistic facts are the open symbols parameterizing the well-founded semantics and rule bodies are conjunctions of literals instead of first order formulae.

Note that negation of probabilistic facts is used merely for convenience: within the distribution semantics, this form of negation could alternatively be modeled by introducing a new fact $not\_f$ for each probabilistic fact $p :: f$ and defining the joint probability of the facts as $P_F(f = 1, not\_f = 0) = p$, $P_F(f = 0, not\_f = 1) = 1 - p$, and 0 otherwise.

The set of probabilistic facts can also be seen as a probabilistic database with independent tuples [Suciu, 2008]. To emphasize this view of uncertainty in the data as opposed to uncertainty in the background knowledge, we will sometimes also refer to a ProbLog program as a ProbLog database or probabilistic database.

The following example illustrates that ProbLog, while initially inspired by the biological network application, is not limited to this setting, but a general purpose probabilistic programming language. Specifically, it is an example for the use of negated probabilistic facts in clause bodies as well as the specification of an infinite set of random variables by means of non-ground facts.

[2] While the restriction to positive non-probabilistic literals is not necessary, it simplifies presentation of the core concepts. We will discuss dropping this requirement in Section 3.3.3.

**Example 3.2** *The following ProbLog program encodes the grammar of Example 2.8, which we briefly repeat here:*

$$
\begin{array}{lll}
0.3 : S \to aX & 0.5 : X \to aX & 0.6 : Y \to aX \\
0.7 : S \to bY & 0.1 : X \to bY & 0.2 : Y \to bY \\
 & 0.4 : X \to \epsilon & 0.2 : Y \to \epsilon
\end{array}
$$

```
s([F|R])          :- rule(s,ax,0), a(F), x(R,1).
s([F|R])          :- rule(s,by,0), b(F), y(R,1).

x([F|R],N)        :- rule(x,ax,N), NN is N+1, a(F), x(R,NN).
x([F|R],N)        :- rule(x,by,N), NN is N+1, b(F), y(R,NN).
x([],N)           :- rule(x,e,N).

y([F|R],N)        :- rule(y,ax,N), NN is N+1, a(F), x(R,NN).
y([F|R],N)        :- rule(y,by,N), NN is N+1, b(F), y(R,NN).
y([],N)           :- rule(y,e,N).

a(a).                b(b).

rule(s,ax,N)      :- use(s,ax,N).
rule(s,by,N)      :- not(use(s,ax,N)).
0.3 :: use(s,ax,_).

rule(x,ax,N)      :- use(x,ax,N).
rule(x,by,N)      :- not(use(x,ax,N)), use(x,by,N).
rule(x,e,N)       :- not(use(x,ax,N)), not(use(x,by,N)).
0.5 :: use(x,ax,_).  0.2 :: use(x,by,_).

rule(y,ax,N)      :- use(y,ax,N).
rule(y,by,N)      :- not(use(y,ax,N)), use(y,by,N).
rule(y,e,N)       :- not(use(y,ax,N)), not(use(y,by,N)).
0.6 :: use(y,ax,_).  0.5 :: use(y,by,_).
```

*Apart from the* `rule/3` *literals and the counter passed along by the recursive nonterminals $X$ and $Y$, the first three blocks directly translate the grammar rules into Prolog, introducing one predicate per rule symbol and combining results into lists as done for DCGs in Prolog. The* `rule/3` *literals as defined in the second half of the program are responsible for the probabilistic choice of rules. Their truth values are decided by corresponding probabilistic facts with predicate* `use/3`*, ensuring that only one rule for the same left hand side can be applied at any point of the derivation. Note that the probabilities of these facts have been adjusted such that*

*the probability of for instance* `use(x,ax,N)` *being* $false$ *and* `use(x,by,N)` *being true, that is,* $(1-0.5)\cdot 0.2$*, equals the probability of the corresponding grammar rule* $0.1 : X \to bY$*.*[3] *The third argument indicating the current position in the sentence ensures that all applications of the same rule within a derivation are independent random events, and that derivations differing at some position belong to mutually exclusive partial interpretations. Each complete interpretation of the probabilistic facts permits a derivation of exactly one* `s/1` *atom. For instance, as each interpretation extending the partial interpretation*

$$\{\mathtt{use}(\mathtt{s},\mathtt{ax},0)=1,\mathtt{use}(\mathtt{x},\mathtt{ax},1)=1,\mathtt{use}(\mathtt{x},\mathtt{by},2)=1,$$

$$\mathtt{use}(\mathtt{x},\mathtt{ax},2)=0,\mathtt{use}(\mathtt{y},\mathtt{by},3)=0,\mathtt{use}(\mathtt{y},\mathtt{ax},3)=0\} \tag{3.2}$$

*covers the only derivation of* `s([a,a,b])`*, cf. Example 2.8, the probability of* `s([a,a,b])` *is*

$$0.3\cdot 0.5\cdot((1-0.5)\cdot 0.2)\cdot((1-0.6)\cdot(1-0.5))=0.3\cdot 0.5\cdot 0.1\cdot 0.2=0.003. \tag{3.3}$$

*Note that without the third argument of the probabilistic facts, this query could not be proven, as it requires* $\mathtt{use}(\mathtt{x},\mathtt{ax},1)=1$ *but* $\mathtt{use}(\mathtt{x},\mathtt{ax},2)=0$*, a distinction that could not be made using a single fact* $\mathtt{use}(\mathtt{x},\mathtt{ax})$*.*

Non-ground facts allow one to specify models with countably infinitely many random variables in ProbLog. As discussed in Section 2.2, thanks to the compatibility condition of Equation (2.2) and the finite support condition, finite subsets of random variables and thus finitely many groundings of such facts are sufficient to compute probabilities in practice, though the actual number of groundings needed might be determined by the query only, as in Equation (3.2). We therefore restrict the discussion to the finite case in the following. Formally, a *ProbLog program* is of the form

$$T=\{p_1 :: f_1,\cdots,p_n :: f_n\}\cup BK \tag{3.4}$$

where $BK$ contains rules as given in Equation (3.1). Given such a program $T$ and a finite set of possible grounding substitutions $\{\theta_{j1},\dots\theta_{ji_j}\}$ for each probabilistic fact $p_j :: f_j$, we define the set $L^T$ of *logical facts* as the maximal set of ground facts $f_i\theta_{ij}$ that can be added to $BK$, that is,

$$L^T=\{f_1\theta_{11},\dots,f_1\theta_{1i_1},\cdots,f_n\theta_{n1},\dots,f_n\theta_{ni_n}\}. \tag{3.5}$$

Correspondingly, we use $F^T$ to denote the set of ground probabilistic facts

$$F^T=\{p_1 :: f_1\theta_{11},\dots,p_1 :: f_1\theta_{1i_1},\cdots,p_n :: f_n\theta_{n1},\dots,p_n :: f_n\theta_{ni_n}\}. \tag{3.6}$$

[3] Obviously, manually encoding such dependencies is tedious; we will therefore introduce a specific language construct for this type of choices in Section 3.3.1.

For ease of readability, in the remainder of this section we will assume that all probabilistic facts in a ProbLog program are ground, such that substitutions need not be written explicitly. We write *partial interpretations* $I$ of $L^T$ as a tuple $(I^1, I^0)$ with $I^1 \cup I^0 \subseteq L^T$, where $I^1$ and $I^0$ correspond to the sets of facts assigned $true$ and $false$ in $I$, respectively:

$$I^1 = \{f_i \in L^T | f_i = 1 \in I\} \tag{3.7}$$

$$I^0 = \{f_i \in L^T | f_i = 0 \in I\} \tag{3.8}$$

We denote the set of facts whose truth values are not specified in $I$ by $I^?$, that is,

$$I^? = L^T \setminus (I^1 \cup I^0) \tag{3.9}$$

If $I^?$ is empty, the interpretation is *complete*. We say that a partial interpretation $J$ *extends* a partial interpretation $I$, written as $I \subseteq J$, if $J$ agrees on all truth values assigned in $I$. The set of *completions* of a partial interpretation $I$ contains all complete interpretations $J$ extending $I$, that is,

$$Compl^T(I) = \{(J^1, J^0) \mid I \subseteq J \wedge J^1 \cup J^0 = L^T\} \tag{3.10}$$

For complete interpretations, $I^1$ can also be seen as a subprogram $L$ of $L^T$, a point of view taken in most papers on ProbLog so far. As the random variables corresponding to facts in $L^T$ are mutually independent, the ProbLog program defines a *probability distribution over interpretations* $I$ of $L^T$ as follows

$$P^T(I) = \prod_{f_i \in I^1} p_i \prod_{f_i \in I^0} (1 - p_i). \tag{3.11}$$

Since the background knowledge $BK$ is fixed and probabilistic facts cannot unify with rule heads, each interpretation $I$ of $L^T$ gives rise to exactly one least Herbrand model $M_I(T)$ of $L^T \cup BK$, and a ProbLog program thus defines a distribution over its least Herbrand models as in the general version of the distribution semantics presented in Section 2.2.

The probability of a partial interpretation $I$ can easily be obtained from $P^T$ by marginalizing out all random variables whose truth values are not fixed, that is, by summing the probabilities of all complete interpretations $J$ extending $I$:

$$P^T(I) = \sum_{J \in Compl^T(I)} P^T(J) = \prod_{f_i \in J^1} p_i \prod_{f_i \in J^0} (1 - p_i) \tag{3.12}$$

A partial interpretation $I$ of $L^T$ *supports* a query $q$, written as $I \models_T q$, if and only if for all $J \in Compl^T(I)$ there exists a substitution $\theta$ such that $M_J(T) \models q\theta$. Intuitively, such a partial interpretation provides sufficient information to prove an

instance of the query independently of the truth values of remaining probabilistic facts.[4] The set of *explanations* (sometimes also called *proofs*) of a query $q$ contains all minimal partial interpretations $E$ of $L^T$ (w.r.t. $\subseteq$) supporting $q$:[5]

$$Expl^T(q) = \{E \mid E \models_T q \wedge \neg\exists I : I \neq E \ \wedge \ I \subseteq E \ \wedge \ I \models_T q\} \tag{3.13}$$

We also write explanations as conjunctions

$$E = \bigwedge_{f_i \in E^1} b_i \wedge \bigwedge_{f_i \in E^0} \neg b_i \tag{3.14}$$

where $b_i$ denotes the random variable corresponding to ground fact $f_i$. As an explanation is a partial interpretation, its probability is given by Equation (3.12).

The *explanation probability* $P_x^T(q)$ is now defined as the probability of the most likely explanation of the query $q$

$$P_x^T(q) = \max_{E \in Expl^T(q)} P^T(E). \tag{3.15}$$

**Example 3.3** *In Example 3.1, $E^0$ is always empty, as the background knowledge does not contain clauses with negated facts in the body. In the following, we denote random variables corresponding to facts of type* `edge(x,y)` *by $xy$. There are two possible explanations for* `path(c,d)`*: the edge from* c *to* d*, that is $E = cd$ (with probability 0.9) as well as the path consisting of the edges from* c *to* e *and from* e *to* d*, that is $E = ce \wedge ed$ (with probability $0.8 \cdot 0.5 = 0.4$). Thus, $P_x^T(\texttt{path(c,d)}) = 0.9$. In Example 3.2, on the other hand, queries have unique explanations, that is, the explanation probability of* `s([a,a,b])` *is $P_x^T(\texttt{s}([\texttt{a},\texttt{a},\texttt{b}])) = 0.003$, as its only explanation is $use(s, ax, 0) \wedge use(x, ax, 1) \wedge use(x, by, 2) \wedge \neg use(x, ax, 2) \wedge \neg use(y, by, 3) \wedge \neg use(y, ax, 3)$.*

The *success probability* $P_s^T(q)$ of a query $q$ in a ProbLog program $T$ is defined as the sum of the probabilities of all complete interpretations of $T$ supporting $q$:

$$P_s^T(q) = \sum_{I \models_T q} P^T(I). \tag{3.16}$$

Formulated differently, the success probability of query $q$ is the probability that the query $q$ is *provable* in a randomly sampled logic program, where free variables are considered existentially quantified as in Prolog.

[4] $I$ typically fixes truth values of all literals needed for some proof of $q$ and thus also determines the substitution $\theta$. However, if $q$ cannot be proven from $I$ alone, but in all its completions, $\theta$ can also depend on the completion. For instance, if $q(a)$ is true whenever probabilistic fact $f$ is true, and $q(b)$ is true whenever $f$ is false, the empty partial interpretation supports $q(X)$, as some instance of the query is guaranteed to be true independently of the truth value of $f$. Similar situations also occur in ground programs, cf. [Poole, 2000].

[5] The distribution semantics [Sato, 1995] calls explanations *minimal support sets* and restricts them to positive assignments, as it does not use negated probabilistic facts to encode binary choices.

**Example 3.4** *In Example 3.1,* 40 *of the* 64 *possible subprograms allow one to prove* `path(c,d)`*, namely all those that contain at least* `edge(c,d)` *or both* `edge(c,e)` *and* `edge(e,d)`*, so the success probability of that query is the sum of the probabilities of these programs:* $P_s^T(\texttt{path(c,d)}) = P^T(\{ab, ac, bc, cd, ce, ed\}) + \ldots + P^T(\{cd\}) = 0.94$*. Clearly, listing all subprograms is infeasible in practice; an alternative approach based on explanations will be discussed in Section 3.2. In Example 3.2, the success probability of query* $\texttt{s}([\texttt{a}, \texttt{a}, \_])$ *takes into account two complete interpretations or possible worlds: one containing* `s([a,a,a])`*, the other* `s([a,a,b])`*, with total probability* $P_s^T(\texttt{s}([\texttt{a}, \texttt{a}, \_])) = 0.03 + 0.003 = 0.033$*.*

We omit the superscript $T$ and simply write $P$, $P_x$ and $P_s$ if $T$ is clear from the context.

## 3.2 The Core of ProbLog Inference

This section discusses the reduction to a formula in disjunctive normal form, or DNF for short, that forms the key to inference in ProbLog. Concrete inference algorithms based on this reduction will be presented in Chapter 4.

Calculating the *success probability* of a query using Equation (3.16) directly is infeasible for all but the tiniest programs, as the number of subprograms to be considered is exponential in the number of probabilistic facts. However, as we have seen in Example 3.1, we can describe all complete interpretations extending a specific explanation by means of all ground probabilistic facts used – positively or negatively – in that explanation, denoted by conjunctions of corresponding random variables and their negations.

**Example 3.5** *In Example 3.3, we have seen the two explanations of query* `path(c,d)`*:* $cd$ *and* $ce \wedge ed$*. The set of all complete interpretations extending* some *explanation can be described by the disjunction of all possible explanations, in our case,* $cd \vee (ce \wedge ed)$*. Note that in all interpretations assigning true to variables* $cd$*,* $ce$ *and* $ed$*, both explanations are true, and the explanations are thus not mutually exclusive, an issue we will come back to at the end of this section.*

This idea, which will be elaborated in more detail below, forms the basis for ProbLog's two-step inference method:

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

Similar approaches are used for PRISM [Sato and Kameya, 2001], ICL [Poole, 2000] and pD [Fuhr, 2000], cf. also Section 4.5. In the following, we discuss the reduction to DNF in more detail. Constructing the formula using Prolog inference as well as computing its probability will be elaborated in Section 4.1.

As we have seen in Equation (3.14), a particular explanation $E$ of a query $q$ is a conjunctive formula $\bigwedge_{f_i \in E^1} b_i \wedge \bigwedge_{f_i \in E^0} \neg b_i$, which at the same time represents the set of all interpretations extending $E$. Furthermore, the set of all interpretations extending *some* explanation of $q$ can be denoted by

$$D_x^T(q) = \bigvee_{E \in Expl^T(q)} \left( \bigwedge_{f_i \in E^1} b_i \wedge \bigwedge_{f_i \in E^0} \neg b_i \right), \tag{3.17}$$

as the following derivation shows:

$$\bigvee_{E \in Expl^T(q)} \left( \bigwedge_{f_i \in E^1} b_i \wedge \bigwedge_{f_i \in E^0} \neg b_i \right) \tag{3.18}$$

$$= \bigvee_{E \in Expl^T(q)} \left( \bigwedge_{f_i \in E^1} b_i \wedge \bigwedge_{f_i \in E^0} \neg b_i \wedge \bigwedge_{f_i \in E^?} (b_i \vee \neg b_i) \right) \tag{3.19}$$

$$= \bigvee_{E \in Expl^T(q)} \bigvee_{I \in Compl^T(E)} \left( \bigwedge_{f_i \in I^1} b_i \wedge \bigwedge_{f_i \in I^0} \neg b_i \right) \tag{3.20}$$

$$= \bigvee_{I \models_T q} \left( \bigwedge_{f_i \in I^1} b_i \wedge \bigwedge_{f_i \in I^0} \neg b_i \right) \tag{3.21}$$

Starting from the DNF in (3.18), the first step (3.19) adds all possible ways of extending an explanation $E$ to a complete interpretation by considering each fact whose truth value is not specified by $E$ in turn. We then note that combinations of these fact-wise extensions lead to complete interpretations extending $E$, leading to the transformation in (3.20). Finally, in (3.21), we rewrite the condition of the disjunction in the terms of Equation (3.16). As shown in Section 3.1, even in the presence of negated probabilistic facts in background clause bodies, for each interpretation of $L^T$, there is a unique least Herbrand model extending this interpretation to the full Herbrand base of the program. Thus, the least Herbrand model based on an interpretation extending an explanation of $q$ contains some ground instance of $q$, and vice versa, if such a least Herbrand model includes $q$, the underlying interpretation of $L^T$ is an extension of some explanation of $q$.[6] As the

[6]This is not a one-to-one mapping, but redundant notation does not influence the truth value of the formula.

DNF now contains conjunctions representing complete interpretations for $L^T$, its probability is a sum of products, which directly corresponds to Equation (3.16):

$$P\left(\bigvee_{I\models_T q}\left(\bigwedge_{f_i\in I^1} b_i \wedge \bigwedge_{f_i\in I^0} \neg b_i\right)\right) \tag{3.22}$$

$$= \sum_{I\models_T q}\left(\prod_{f_i\in I^1} p_i \cdot \prod_{f_i\in I^0}(1-p_i)\right) \tag{3.23}$$

$$= \sum_{I\models_T q} P^T(I) \tag{3.24}$$

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

$$P_s^T(q) = P\left(\bigvee_{E\in Expl^T(q)}\left(\bigwedge_{f_i\in E^1} b_i \wedge \bigwedge_{f_i\in E^0} \neg b_i\right)\right) \tag{3.25}$$

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. Note that we here rely on the DNF formula being finite, or the *finite support condition* in terms of [Sato, 1995]. This is a reasonable assumption in practice: as explanations are *minimal* partial interpretations, an infinite DNF would require infinitely many random variables, in which case the sum over interpretations in Equation (3.16) would be infinite as well.

If all conjunctions in a DNF are *mutually exclusive*, that is, at most one such conjunction is true for each interpretation of the underlying Boolean variables, the probability of the DNF is the sum of the probabilities of these conjunctions. However, as we have seen in Example 3.5, the explanation-based DNF of Equation (3.25) typically contains overlap, that is, there are interpretations extending multiple explanations. In this case, the sum of these probabilities exceeds the probability of the DNF. The problem of transforming such a DNF into an equivalent DNF with mutually exclusive conjunctions only is known as the *disjoint-sum-problem* or the two-terminal network reliability problem, which is #P-complete [Valiant, 1979]. We will discuss this problem in more detail in Section 4.1.1.

## 3.3 Additional Language Concepts

This section highlights three general concepts for probabilistic programming that are covered by ProbLog's semantics, but not directly visible in the basic definitions

as outlined so far. The aim of making these concepts explicit language elements is twofold: from a practical point of view, they often allow for more convenient modeling; from a theoretical point of view, they provide insights into the relationship to closely related languages. The latter will be elaborated in Section 3.4.

### 3.3.1 Annotated Disjunctions

An *annotated disjunction* (AD) is an expression of the form

$$\mathtt{p_1} :: \mathtt{h_1} \;\; ; \;\; \ldots \;\; ; \;\; \mathtt{p_n} :: \mathtt{h_n} \;\; :- \;\; \mathtt{b_1}, \;\ldots\;, \mathtt{b_m}. \tag{3.26}$$

where $\mathtt{b_1}, \ldots, \mathtt{b_m}$ is a possibly empty conjunction of literals, the $\mathtt{p_i}$ are probabilities and $\sum_{i=1}^{n} \mathtt{p_i} \leq 1$. It states that if the body $\mathtt{b_1}, \ldots, \mathtt{b_m}$ is true at most one of the $\mathtt{h_i}$ is true as well, where the choice is governed by the probabilities. As for probabilistic facts, a non-ground AD denotes the set of all its groundings, and for each such grounding, choosing one of its head atoms to be true is seen as an independent random event. Alternatively, this can again be viewed as adding one Horn clause per grounding to a sampled program. If the $\mathtt{p_i}$ in an annotated disjunction do not sum to one, with probability $1 - \sum_{i=1}^{n} \mathtt{p_i}$ none of the head atoms is proven by this clause.

**Example 3.6** *The following set of ADs defines the* `rule/3` *predicate used in Example 3.2:*

```
0.3 :: rule(s,ax,N) ; 0.7 :: rule(s,by,N).
0.5 :: rule(x,ax,N) ; 0.1 :: rule(x,by,N) ; 0.4 :: rule(x,e,N).
0.6 :: rule(y,ax,N) ; 0.2 :: rule(y,by,N) ; 0.2 :: rule(y,e,N).
```

Annotated disjunctions were introduced in LPADs [Vennekens et al., 2004] and are used as well in CP-logic [Vennekens, 2007], where they are named *clauses with annotated disjunctions* and *CP-events*, respectively. They are closely related to probabilistic facts (annotated disjunctions with a single atom in the head and an empty body), the alternatives used in ICL (unconditional annotated disjunctions, that is, annotated disjunctions with empty body), and the switches used in PRISM (unconditional annotated disjunctions where all atoms share the same predicate, but using implicit trial identifiers to distinguish repeated occurrences); cf. also Section 3.4.

Annotated disjunctions can be automatically transformed into the core primitives of ProbLog. However, as ProbLog's probabilistic facts are independent random variables, but head atoms of ADs are dependent in the sense that at most one of them can be true in each possible world, these dependencies have to be modeled on top of the probabilistic facts, as was done in Example 3.2. For non-ground ADs, all

variables of the AD have to be included as arguments of the probabilistic facts to ensure that different groundings of the AD correspond to different random events. This includes the case of variables occurring only in the body of the AD, as in the following example.

**Example 3.7** *This AD models the fact that the probability that some window breaks increases with the number of balls hitting it:*

$$0.3 :: \texttt{broken}(\texttt{Window}) \ :- \ \texttt{hits}(\texttt{Ball}, \texttt{Window}). \tag{3.27}$$

*Using probabilistic facts, it can be written as*

$$\texttt{broken}(\texttt{Window}) \ :- \ \texttt{hits}(\texttt{Ball}, \texttt{Window}), \texttt{breaks}(\texttt{Ball}, \texttt{Window}).$$

$$0.3 :: \texttt{breaks}(\texttt{Ball}, \texttt{Window}).$$

In general, an annotated disjunction $\texttt{p}_1 :: \texttt{h}_1 \ ; \ \ldots \ ; \ \texttt{p}_n :: \texttt{h}_n \ :- \ \texttt{b}_1, \ \ldots \ , \texttt{b}_m.$ can be written in terms of ProbLog's probabilistic facts as the set of probabilistic facts $\{\tilde{\texttt{p}}_\texttt{i} :: \texttt{x}(\texttt{h}_\texttt{i}, \mathbf{v}) | 1 \leq i \leq n\}$ and corresponding clauses

$$\texttt{h}_\texttt{i} \ :- \ \texttt{b}_1, \cdots, \texttt{b}_m, \texttt{not}(\texttt{x}(\texttt{h}_1, \mathbf{v})), \ldots, \texttt{not}(\texttt{x}(\texttt{h}_{\texttt{i}-1}, \mathbf{v})), \texttt{x}(\texttt{h}_\texttt{i}, \mathbf{v}). \tag{3.28}$$

where $\texttt{x}$ is a new predicate not appearing elsewhere in the ProbLog program and $\mathbf{v} = \texttt{v}_1, \ldots, \texttt{v}_\texttt{a}$ are the variables in the AD. If the $p_i$ sum to 1, it is possible to drop the last probabilistic fact $\texttt{x}(\texttt{h}_\texttt{n}, \mathbf{v})$, since the last option has to be chosen if it is reached in the sequential decision process. The probability $\tilde{p}_1$ is defined as $p_1$, for $i > 1$, the following transformation applies:

$$\tilde{p}_i := \begin{cases} p_i \cdot \left(1 - \sum_{j=1}^{i-1} p_j\right)^{-1} & \text{if } p_i > 0 \\ 0 & \text{if } p_i = 0 \end{cases} . \tag{3.29}$$

One can recover the original probabilities from $\tilde{p}$ by setting $p_1 := \tilde{p}_1$ and iteratively applying the following transformation for $i = 2, 3, \ldots, n$

$$p_i := \tilde{p}_i \cdot \left(1 - \sum_{j=1}^{i-1} p_j\right) . \tag{3.30}$$

Equations (3.29) and (3.30) together define a bijection between $p$ and $\tilde{p}$.

Since the translation process can be completely automated, we will use ADs as a basic language construct in ProbLog from now on whenever this is more convenient than probabilistic facts.

### 3.3.2 Repeated Trials

In ProbLog, ground facts directly correspond to random variables, that is, once the truth value of such a fact is fixed, it can be used an arbitrary number of times to derive the truth values of further facts in the interpretation without changing the probability of the interpretation. In contrast, in probabilistic grammars, each application of a rule lowers the probability of a derived sentence. In our earlier grammar examples, we therefore introduced an extra index argument for probabilistic facts or annotated disjunctions to generate a different ground instance for each repeated use of a rule.

**Example 3.8** *The following is a variant of Example 3.6 without indexing:*

```
0.3 :: rule(s,ax)  ;  0.7 :: rule(s,by).
0.5 :: rule(x,ax)  ;  0.1 :: rule(x,by)  ;  0.4 :: rule(x,e).
0.6 :: rule(y,ax)  ;  0.2 :: rule(y,by)  ;  0.2 :: rule(y,e).


s([F|R])  :- rule(s,ax),a(F),x(R).
s([F|R])  :- rule(s,by),b(F),y(R).

x([F|R])  :- rule(x,ax),a(F),x(R).
x([F|R])  :- rule(x,by),b(F),y(R).
x([])     :- rule(x,e).

y([F|R])  :- rule(y,ax),a(F),x(R).
y([F|R])  :- rule(y,by),b(F),y(R).
y([])     :- rule(y,e).

a(a).       b(b).
```

*In this program,* $P^T(\texttt{s}([\texttt{a},\texttt{a},\texttt{b}])) = 0$*, as opposed to* $0.003$ *in the original example. The reason is that deriving* `aab` *requires two different outcomes of the second AD:* `rule(x,ax)` *to generate the second letter, and* `rule(x,by)` *to obtain the third. However, as each interpretation contains exactly one of these,* `s([a,a,b])` *is false in all possible interpretations.*

While the distribution semantics as introduced in [Sato, 1995] follows the ProbLog view, PRISM [Sato and Kameya, 2001] follows the grammar view of implicitly distinguishing repeated calls to the same probabilistic alternative as independent identically distributed events.

Maintaining trial identifiers as done in our grammar examples can be completely automated. To this aim, we introduce a declaration `:- repeated pred/arity` to indicate that the corresponding predicate has to be treated as a different random event each time it occurs.[7] We require such predicates to be defined by one or more unconditional ADs whose head elements all use this predicate, where for each AD, the set of variables is the same in all elements.[8]

Formally, the following declaration of a group of $m$ ADs for a predicate `q/n`

$$:-\texttt{repeated q/n}.$$

$$\mathtt{p}_{11} :: \mathtt{q}(\mathbf{t_{11}}); \ldots; \mathtt{p}_{1\mathtt{k}_1} :: \mathtt{q}(\mathbf{t_{1k_1}}).$$

$$\ldots$$

$$\mathtt{p}_{\mathtt{m}1} :: \mathtt{q}(\mathbf{t_{m1}}); \ldots; \mathtt{p}_{\mathtt{mk}_\mathtt{m}} :: \mathtt{q}(\mathbf{t_{mk_m}}).$$

will be extended using a new predicate `qx/n`:

$$\mathtt{q}(\mathbf{X}) :- \mathtt{next\_id}(\mathtt{q}(\mathbf{X}), \mathtt{ID}), \mathtt{qx}(\mathtt{ID}, \mathbf{X}).$$

$$\mathtt{p}_{11} :: \mathtt{qx}(\mathtt{ID}, \mathbf{t_{11}}); \ldots; \mathtt{p}_{1\mathtt{k}_1} :: \mathtt{qx}(\mathtt{ID}, \mathbf{t_{1k_1}}).$$

$$\ldots$$

$$\mathtt{p}_{\mathtt{m}1} :: \mathtt{qx}(\mathtt{ID}, \mathbf{t_{m1}}); \ldots; \mathtt{p}_{\mathtt{mk}_\mathtt{m}} :: \mathtt{qx}(\mathtt{ID}, \mathbf{t_{mk_m}}).$$

The transformation replaces the original ADs defining predicate `q/n` by ADs that use a new head predicate of arity `n+1`, where we introduce an extra argument corresponding to the trial identifier. A wrapper clause for the original predicate `q/n` is used to generate unique identifiers and call the new AD.[9]

**Example 3.9** *Declaring* `rule/2` *repeated in Example 3.8 would again lead to an encoding of the original grammar, as it automatically introduces a distinguishing argument. For instance,*

$$:-\texttt{repeated rule/2}.$$

$$0.5 :: \mathtt{rule}(\mathtt{x}, \mathtt{ax}) \;\; ; \;\; 0.1 :: \mathtt{rule}(\mathtt{x}, \mathtt{by}) \;\; ; \;\; 0.4 :: \mathtt{rule}(\mathtt{x}, \mathtt{e}).$$

[7] Using declarations on predicate level allows programs mixing standard and repeated probabilistic events and simplifies automatic transformation.

[8] This ensures that all relevant groundings can be identified during proving; conditional ADs could be used under the additional requirement of all variables in the body appearing in each head element.

[9] While it is in principle possible to transform the entire program such that ID information for all predicates is passed around as extra argument, interfacing this information by means of the `next_id` predicate makes the transformation clearer to present. The implementation of this predicate uses backtrackable global variables to ensure correct interaction with our ProbLog implementation.

*will be transformed into*

$$\texttt{rule}(\texttt{L},\texttt{R}) :- \texttt{next\_id}(\texttt{rule}(\texttt{L},\texttt{R}),\texttt{ID}),\texttt{r}(\texttt{ID},\texttt{L},\texttt{R}).$$

$$0.5 :: \texttt{r}(\texttt{ID},\texttt{x},\texttt{xa}); 0.1 :: \texttt{r}(\texttt{ID},\texttt{x},\texttt{by}); 0.4 :: \texttt{r}(\texttt{ID},\texttt{x},\texttt{e}).$$

### 3.3.3 Negation

As we have seen in Section 2.2, the distribution semantics relies on the fact that each model of the basic set of facts can be extended into a unique canonical model for the entire program. We can thus generalize background knowledge clauses to normal clauses, as long as the resulting program still has a total well-founded model for each interpretation of the probabilistic facts. Viewing background knowledge clauses as syntactically restricted definitions in the spirit of FO(ID) [Denecker and Vennekens, 2007; Vennekens et al., 2009], a partial well-founded model for some choice of probabilistic facts would indicate ambiguity in a definition due to some non well-founded use of negation, which is undesirable from both the computational and the modeling perspective. We thus require ProbLog programs to have a total well-founded model for each interpretation of probabilistic facts without committing to a particular class of such programs.

In this case, and again assuming finite support, ProbLog's DNF-based inference as described in Section 3.2 can be extended to construct a nested formula instead of a DNF [Kimmig et al., 2009]. While the DNF $D_x^T(g)$ corresponding to a goal $g$ encodes the set of all interpretations entailing $g$, the formula $\neg D_x^T(g)$ encodes the complement of this set. Under negation as failure, this is exactly the set of interpretations where the corresponding negated subgoal $not(g)$ is true. Following a lazy approach, we therefore describe sets of interpretations where a goal is true by *generalized explanations*, which in addition to positive and negative literals representing probabilistic facts also contain negated subformulae corresponding to generalized explanations of negated subgoals. Note that such generalized explanations can contain logical contradictions if different truth values for probabilistic facts are required in different subformulae, in which case the set of interpretations is empty.

**Example 3.10** *The following is a ProbLog encoding of the alarm Bayesian network of Example 2.9:*

```
alarm      :- burglary, earthquake, alarm(b, e).
alarm      :- burglary, not(earthquake), alarm(b, not_e).
alarm      :- not(burglary), earthquake, alarm(not_b, e).
alarm      :- not(burglary), not(earthquake), alarm(not_b, not_e).

johnCalls  :- alarm, johnCalls(a).
johnCalls  :- not(alarm), johnCalls(not_a).

maryCalls  :- alarm, maryCalls(a).
maryCalls  :- not(alarm), maryCalls(not_a).


0.001 :: burglary.          0.002 :: earthquake.

0.95 :: alarm(b, e).        0.94 :: alarm(b, not_e).
0.29 :: alarm(not_b, e).    0.001 :: alarm(not_b, not_e).

0.9 :: johnCalls(a).        0.05 :: johnCalls(not_a).

0.7 :: maryCalls(a).        0.01 :: maryCalls(not_a).
```

*The following formulae describe the conditions under which the nodes in the BN evaluate to true ($D_x(maryCalls)$ is analogous to $D_x(johnCalls)$):*

$$D_x(burglary) = b$$

$$D_x(earthquake) = e$$

$$\begin{aligned} D_x(alarm) = {} & (b \wedge e \wedge a(b,e)) \vee (b \wedge \neg e \wedge a(b,ne)) \\ & \vee (\neg b \wedge e \wedge a(nb,e)) \vee (\neg b \wedge \neg e \wedge a(nb,ne)) \end{aligned}$$

$$\begin{aligned} D_x(johnCalls) = {} & (b \wedge e \wedge a(b,e) \wedge j(a)) \vee (b \wedge \neg e \wedge a(b,ne) \wedge j(a)) \\ & \vee (\neg b \wedge e \wedge a(nb,e) \wedge j(a)) \vee (\neg b \wedge \neg e \wedge a(nb,ne) \wedge j(a)) \\ & \vee (\neg D_x(alarm) \wedge j(na)) \end{aligned}$$

*Here, $D_x(alarm)$ is still in DNF, as all its negated subgoals correspond to probabilistic facts. However, the last conjunction of $D_x(johnCalls)$ no longer corresponds to an individual explanation or partial interpretation, as $\neg D_x(alarm)$ is not a conjunction of literals. Indeed, negating $D_x(alarm)$ results in a conjunctive formula which has models extending each possible truth value assignment to variables*

*b and e, as given by the four mutually exclusive explanations*

$$(b \wedge e \wedge \neg a(b,e)) \vee (b \wedge \neg e \wedge \neg a(b,ne))$$

$$\vee (\neg b \wedge e \wedge \neg a(nb,e)) \vee (\neg b \wedge \neg e \wedge \neg a(nb,ne))$$

Obviously, the formula describing the generalized explanations of a query could be transformed in DNF using the standard laws of logic. However, as we will see in Section 4.3.4, this is not necessary for ProbLog's approach to probability calculation. Furthermore, the approach can be generalized by also including subformulae for (selected) positive subgoals, as done in tabled inference for ProbLog [Mantadelis and Janssens, 2010].

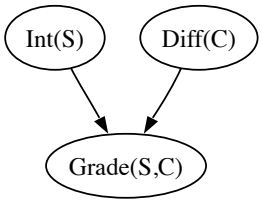
## 3.4 Related Languages

In this section, we briefly discuss ProbLog's relationship to a number of alternative probabilistic logic languages. We start with some general remarks concerning relational extensions of Bayesian networks, cf. Section 2.3.2, and then focus on languages that add independent alternatives to logical languages, cf. Section 2.3.1, and thus can be viewed as instances of the distribution semantics. While these languages share their core semantics, they originate from different fields, ranging from logic programming to probabilistic information retrieval and knowledge representation.

### 3.4.1 Relational Extensions of Bayesian Networks

As their propositional counterpart, probabilistic logic languages extending Bayesian networks essentially encode a joint probability distribution over a set of random variables by means of a set of conditional distributions over subsets of those variables. The key difference, however, is that in the relational case, the graph structure specifying independencies is typically parameterized by logical variables. As already indicated in Example 3.10, Bayesian networks with binary domains can be modeled in ProbLog by introducing a logical atom for each random variable in the network and directly encoding probability tables using a combination of background knowledge clauses and probabilistic facts. In the following example, we generalize this to both larger domains and relational extensions of Bayesian networks.

**Example 3.11** *Figure 3.2 shows a fragment of the school CLP($\mathcal{BN}$) program of [Santos Costa et al., 2003], where for any given student and course taken by the student, the grade of the student for this course depends on the difficulty*




| Int | | high | | | medium | | | low | |
|---|---|---|---|---|---|---|---|---|---|
| Diff | high | med | low | high | med | low | high | med | low |
| a | 0.20 | 0.70 | 0.85 | 0.10 | 0.20 | 0.50 | 0.01 | 0.05 | 0.10 |
| b | 0.60 | 0.25 | 0.12 | 0.30 | 0.60 | 0.35 | 0.04 | 0.15 | 0.40 |
| c | 0.15 | 0.04 | 0.02 | 0.40 | 0.15 | 0.12 | 0.50 | 0.60 | 0.40 |
| d | 0.05 | 0.01 | 0.01 | 0.20 | 0.05 | 0.03 | 0.45 | 0.20 | 0.10 |

Figure 3.2: A fragment of the school example encoding the dependency of a student's grade for a course on the student's intelligence and the course's difficulty.

*of the course as well as on the student's intelligence. The following ProbLog encoding of this fragment directly follows the original encoding in CLP(BN). Which student is following which course is defined by the background knowledge predicate* `registration(ID,Course,Student)`*. As in the figure, we omit the definitions of this predicate as well as the predicates corresponding to the parent nodes* `course_difficulty(CKey,Dif)` *and* `student_intelligence(SKey,Int)` *for brevity. The graph structure is encoded by the following background knowledge clause:*

```
registration_grade(Key,Grade) :- registration(Key,CKey,SKey),
                                 course_difficulty(CKey,Dif),
                                 student_intelligence(SKey,Int),
                                 grade(Key,[Int,Dif],Grade).
```

*Furthermore, given domains* {`high`, `medium`, `low`} *for both intelligence and difficulty and* {`a`, `b`, `c`, `d`} *for grade, we get one AD for each grounding of parent variables Intelligence and Difficulty, i.e. for each column of the CPT:*

```
0.20 :: grade(Key,[high,high],a) ; 0.60 :: grade(Key,[high,high],b) ;
0.15 :: grade(Key,[high,high],c) ; 0.05 :: grade(Key,[high,high],d).

0.70 :: grade(Key,[high,medium],a) ; 0.25 :: grade(Key,[high,medium],b) ;
0.04 :: grade(Key,[high,medium],c) ; 0.01 :: grade(Key,[high,medium],d).
```

$0.85 :: \texttt{grade}(\texttt{Key}, [\texttt{high}, \texttt{low}], \texttt{a})$ ; $0.12 :: \texttt{grade}(\texttt{Key}, [\texttt{high}, \texttt{low}], \texttt{b})$ ;
$0.02 :: \texttt{grade}(\texttt{Key}, [\texttt{high}, \texttt{low}], \texttt{c})$ ; $0.01 :: \texttt{grade}(\texttt{Key}, [\texttt{high}, \texttt{low}], \texttt{d})$.

...

We refer to [Sato and Kameya, 2001] for a more in-depth discussion on modeling Bayesian networks in PRISM, and to Section 3.4.4 for a mapping of PRISM programs into ProbLog.

### 3.4.2 Probabilistic Logic Programs

The probabilistic logic programs of Dantsin [1991] correspond exactly to finite, stratified ProbLog programs: a probability is assigned to each element of a finite set of facts, and a finite set of clauses is added such that no probabilistic fact is an instance of a head of any such clause. To correctly deal with negation, programs are required to be stratified. However, it seems that this approach has neither been implemented nor pursued any further.

### 3.4.3 Independent Choice Logic

The Independent Choice Logic (ICL) [Poole, 2000] builds on probabilistic Horn abduction (PHA) [Poole, 1993b], but generalizes PHA in two aspects, namely by including negation as failure and by dropping the requirement of mutually exclusive explanations. The key difference between ProbLog and ICL is the choice of basic probabilistic entities: while ProbLog uses probabilistic facts that are either true or false, ICL uses probabilistic *alternatives* that are added to an acyclic normal logic program. An alternative is a set of ground atoms that are called *atomic choices*. Possible worlds or *total choices* contain exactly one atom from each alternative. Different alternatives are probabilistically independent.

**Example 3.12** *The following ICL theory is taken from [Poole, 2008]. We use the syntax of Poole's ICL implementation AILog2 for alternatives.*

$\texttt{prob}\ \ \texttt{c}_1 : 0.5, \texttt{c}_2 : 0.3, \texttt{c}_3 : 0.2.$
$\texttt{prob}\ \ \texttt{b}_1 : 0.9, \texttt{b}_2 : 0.1.$

$\texttt{f} \leftarrow \texttt{c}_1 \wedge \texttt{b}_1.$ $\quad$ $\texttt{d} \leftarrow \texttt{c}_1.$ $\quad$ $\texttt{e} \leftarrow \texttt{f}.$
$\texttt{f} \leftarrow \texttt{c}_2 \wedge \texttt{b}_2.$ $\quad$ $\texttt{d} \leftarrow \neg\texttt{c}_2 \wedge \texttt{b}_1.$ $\quad$ $\texttt{e} \leftarrow \neg\texttt{d}.$

*The first two lines define the set of alternatives, that is, each possible world contains one of the* $\texttt{c}_\texttt{i}$ *and one of the* $\texttt{b}_\texttt{j}$*. The remaining two lines define the acyclic logic program (the* facts *in ICL terminology).*

In general, the core language constructs of ICL are normal clauses and alternatives of the form

$$\texttt{prob}\ \ \mathtt{a}_1 : \mathtt{p}_1, \ldots, \mathtt{a}_n : \mathtt{p}_n.$$

Atomic choices $\mathtt{a}_i$ in the same or different alternatives can neither unify with each other nor with any head $\mathtt{h}_k$ of a clause in the logic program. For each alternative, the probabilities $\mathtt{p}_i$ of its atomic choices $\mathtt{a}_i$ sum to one.

Given the additional language constructs of Section 3.3, ICL theories can therefore directly be mapped into ProbLog: the logic program is kept as is (modulo syntax), and each alternative

$$\texttt{prob}\ \ \mathtt{a}_1 : \mathtt{p}_1, \ldots, \mathtt{a}_n : \mathtt{p}_n.$$

corresponds to an annotated disjunction with empty body

$$\mathtt{p}_1 :: \mathtt{a}_1 \ ; \ \cdots \ ; \ \mathtt{p}_n :: \mathtt{a}_n.$$

**Example 3.13** *Example 3.12 thus corresponds to the following ProbLog program:*

$$0.5 :: \mathtt{c}_1 \ ; \ 0.3 :: \mathtt{c}_2 \ ; \ 0.2 :: \mathtt{c}_3.$$
$$0.9 :: \mathtt{b}_1 \ ; \ 0.1 :: \mathtt{b}_2.$$

$$\mathtt{f} \ :- \ \mathtt{c}_1, \mathtt{b}_1. \qquad \mathtt{d} \ :- \ \mathtt{c}_1. \qquad\qquad \mathtt{e} \ :- \ \mathtt{f}.$$
$$\mathtt{f} \ :- \ \mathtt{c}_2, \mathtt{b}_2. \qquad \mathtt{d} \ :- \ \mathtt{not}(\mathtt{c}_2), \mathtt{b}_1. \qquad \mathtt{e} \ :- \ \mathtt{not}(\mathtt{d}).$$

ProbLog's probabilistic facts correspond to alternatives with two atomic choices: the fact itself and some fact that is not used anywhere else (as probabilities need to sum to one). To encode conditional ADs in ICL, a similar approach as in the transformation to basic ProbLog can be used, where we use one clause per head element and introduce an extra body element carrying the probability. In contrast to Equation (3.28), atomic choices are used instead of sequences of facts. That is, an AD

$$\mathtt{p}_1 :: \mathtt{h}_1 \ ; \ \ldots \ ; \ \mathtt{p}_n :: \mathtt{h}_n \ \ :- \ \ \mathtt{b}_1, \ldots, \mathtt{b}_m.$$

can be expressed in ICL as:

$$\mathtt{h}_1 \leftarrow \mathtt{b}_1 \wedge \ldots \wedge \mathtt{b}_m \wedge \mathtt{a}_1.$$
$$\ldots$$
$$\mathtt{h}_n \leftarrow \mathtt{b}_1 \wedge \ldots \wedge \mathtt{b}_m \wedge \mathtt{a}_n.$$

$$\texttt{prob}\ \ \mathtt{a}_1 : \mathtt{p}_1, \ldots, \mathtt{a}_n : \mathtt{p}_n.$$

**Example 3.14** *The AD*

```
0.3 :: word([a|X])  ;  0.7 :: word([b|X])  :-  word(X).
```

*can be written in ICL as*

```
word([a|X])  :-  word(X),next(a,X).
word([b|X])  :-  word(X),next(b,X).

prob  next(a,X) : 0.3,next(b,X) : 0.7.
```

*As ProbLog, AILog2 allows non-ground alternatives to compactly encode sets of independent ground alternatives as in the last line.*

As in ProbLog, the use of negation in ICL has to be restricted to guarantee a single canonical model for each total choice. Poole [2000] therefore considers acyclic ICL programs under the stable model semantics [Gelfond and Lifschitz, 1988], which for this class of programs coincides with the well-founded semantics. A program is *acyclic* if there is a function *num* from the Herbrand base to the natural numbers such that for each grounding $h : -b_1, \ldots, b_n$ of a program clause, $num(h) > num(b_i)$ for all $i$. This is then further relaxed to *contingently acyclic* programs, where those groundings whose body directly fails[10] given the current database are not considered when checking the order constraint. Riguzzi [2009] introduces an inference procedure for *bounded* queries in *modularly acyclic* ICL programs. Boundedness essentially ensures the finite support condition, while modular acyclicity generalizes acyclicity while still guaranteeing the well-founded model to be total; we refer to [Riguzzi, 2009] for the technical details.

### 3.4.4 PRISM

In [Sato, 1995], so-called BS-programs are introduced as a simple type of programs covered by the distribution semantics. In BS-programs, random events are atoms of the form `bs(i,n,_)`. Given a group identifier `i` and an event identifier `n`, the third argument probabilistically takes value 0 or 1, that is, such atoms can be seen as *binary switches*. Given `i`, atoms `bs(i,n,_)` with different `n` are independent random events with identical probability distribution over the outcomes. This idea is generalized into events with more than two outcomes, or *multi-ary random switches* `msw(i,_)`, in PRISM [Sato and Kameya, 2001], where `n` is no longer an explicit argument, but still implicitly used for inference. Strictly speaking, an atom `msw(i,_)` thus corresponds to a *group* of switches, but we refer to it as a switch for simplicity.

[10] due to semantic constraints such as inequality, or because there is no rule to resolve some body atom with

**Example 3.15** *We illustrate PRISM using our grammar example:*

```
s([F|R])  :- msw(s,RHS),
             (RHS == ax -> a(F),x(R); b(F),y(R)).
x(S)      :- msw(x,RHS),
             (RHS == e -> S = [];
             S = [F|R],(RHS == ax -> a(F),x(R); b(F),y(R))).
y(S)      :- msw(y,RHS),
             (RHS == e -> S = [];
             S = [F|R],(RHS == ax -> a(F),x(R); b(F),y(R))).

values(s,[ax,by]).
values(x,[ax,by,e]).
values(y,[ax,by,e]).

:- set_sw(s,[0.3,0.7]).
:- set_sw(x,[0.5,0.1,0.4]).
:- set_sw(y,[0.6,0.2,0.2]).
```

*As sampling execution in the PRISM system requires programs to be written in a purely generative manner [Sato et al., 2010], the clauses use Prolog's if-then-else to continue depending on the random trial. Due to implicit trial identifiers, the program is syntactically closer to the (wrong) ProbLog program in Example 3.8, but in fact directly corresponds to Example 3.6. Switches in PRISM are identified by names in the form of atoms. The example contains three switches called* `s`*,* `x` *and* `y`*, respectively. Switches and their possible outcomes are declared using the predicate* `value/2`*, where the first argument is the name of the switch, the second an ordered list of values; for instance, switch* `s` *returns either* `ax` *or* `by`*. To evaluate a switch in a clause body, the predicate* `msw/2` *is used. Again, the first argument is the name of the switch, whereas the second will be unified randomly with one of the possible values. The probabilities of a switch are set using* `set_sw/2`*, where the second argument is the list of probabilities associated to the outcomes, using the same order as in the values declaration.*

In general, the important language constructs of PRISM are thus the following:

$$\texttt{values}(\texttt{name}(\texttt{t}_1,\ldots,\texttt{t}_n),[\texttt{v}_1,\ldots,\texttt{v}_m]).$$

$$:-\texttt{set\_sw}(\texttt{name}(\texttt{t}_1,\ldots,\texttt{t}_n),[\texttt{p}_1,\ldots,\texttt{p}_m]). \tag{3.31}$$

$$\texttt{h} :- \texttt{b}_1,\ldots,\texttt{b}_{i-1},\texttt{msw}(\texttt{name}(\texttt{t}_1,\ldots,\texttt{t}_n),\texttt{Value}),\texttt{b}_{i+1},\ldots,\texttt{b}_k.$$

where the $\texttt{b}_j$ are atoms (including msw-atoms), `name` is an atom, the $\texttt{v}_j$ are ground terms and the $\texttt{p}_j$ are probabilities summing to 1, meaning that querying any

instance of the switch `name` using $\texttt{msw}(\texttt{name}, \texttt{Value})$ will bind `Value` to $\texttt{v}_\texttt{j}$ with probability $\texttt{p}_\texttt{j}$.

A multi-valued switch in PRISM is closely related to an unconditional annotated disjunction in terms of ProbLog: in both cases, a probabilistic choice from a finite set of alternatives is specified by attaching a probability to each possible value, where probabilities sum to one. However, in PRISM, repeated calls to the same switch are considered independent random events. To achieve this behaviour in ProbLog, the repeated declaration introduced in Section 3.3.2 can be used.

PRISM programs can be translated to ProbLog by mapping each multi-ary switch to an AD with predicate `msw/2`, which is declared repeated. The structure of the Prolog program remains unchanged.

**Example 3.16** *Mapping the PRISM program of Example 3.15 to ProbLog does not affect the clauses for* `s/1`*,* `x/1` *and* `y/1`*. Switch definitions are mapped to*

$$:- \texttt{repeated msw/2}.$$

$$\begin{aligned}
&0.3 :: \texttt{msw}(\texttt{s}, \texttt{ax}) \;\; ; \;\; 0.7 :: \texttt{msw}(\texttt{s}, \texttt{by}). \\
&0.5 :: \texttt{msw}(\texttt{x}, \texttt{ax}) \;\; ; \;\; 0.1 :: \texttt{msw}(\texttt{x}, \texttt{by}) \;\; ; \;\; 0.4 :: \texttt{msw}(\texttt{x}, \texttt{e}). \\
&0.6 :: \texttt{msw}(\texttt{y}, \texttt{ax}) \;\; ; \;\; 0.2 :: \texttt{msw}(\texttt{y}, \texttt{by}) \;\; ; \;\; 0.2 :: \texttt{msw}(\texttt{y}, \texttt{e}).
\end{aligned}$$

The general fragment in Equation (3.31) is mapped to:

$$:- \texttt{repeated msw/2}.$$

$$\texttt{p}_1 :: \texttt{msw}(\texttt{name}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n}), \texttt{v}_1) \;\; ; \;\; \ldots \;\; ; \;\; \texttt{p}_\texttt{m} :: \texttt{msw}(\texttt{name}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n}), \texttt{v}_\texttt{m}). \quad (3.32)$$

$$\texttt{h} \;\; :- \;\; \texttt{b}_1, \ldots, \texttt{b}_{\texttt{i}-1}, \texttt{msw}(\texttt{name}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n}), \texttt{Value}), \texttt{b}_{\texttt{i}+1}, \ldots, \texttt{b}_\texttt{k}.$$

It might be possible to simulate the default behaviour of ProbLog within PRISM by storing the outcome of the first call `msw(i,X)` and only calling `msw/2` atoms if their outcome has not been stored yet.[11] A simple way to realize such a store would be to introduce a dynamic predicate `sampled/2` and to replace each occurrence of an atom `msw(i,X)` in the body of some clause by

$$(\texttt{sampled}(\texttt{i}, \texttt{X}) \; -> \; \texttt{true}; \; \texttt{msw}(\texttt{i}, \texttt{X}), \texttt{assert}(\texttt{sampled}(\texttt{i}, \texttt{X}))).$$

In practice, this simple approach seems to miss alternative explanations for non-ground queries, presumably due to interactions between asserting and backtracking.[12] However, alternative implementations might solve this problem.

[11] Taisuke Sato (personal communication).

[12] Querying the corresponding transformation of Example 3.15 for the probability of $\texttt{s}([\_, \_])$ only takes into account $[\texttt{a}, \texttt{b}]$ and ignores $[\texttt{b}, \texttt{a}]$ even with a new top level predicate that clears `sampled/2` before (or after) calling `s/1`.

In terms of inference, PRISM follows a similar two-step approach as described for ProbLog in Section 3.2. However, PRISM requires programs to be written in such a way that each query has at most one explanation in each possible world. The system thus avoids solving the disjoint-sum-problem (cf. Section 4.1.1) by restricting the class of possible programs. Our ProbLog implementation discussed in Chapter 4 does not impose such restrictions.

### 3.4.5 Probabilistic Datalog

Probabilistic Datalog (pD) [Fuhr, 2000] is motivated by the need to combine logical and probabilistic information for information retrieval. Probabilities are attached to clauses in modularly stratified Datalog programs (under the well-founded semantics), denoting for each grounding of the clause the probability that it holds in a model. Independence between all random events is assumed, but choices between multiple alternative clauses (defining the same predicate) can be specified by means of so-called *disjointness keys*. Such a key is a subset of the arguments of the predicate, denoting that for each grounding of the key, only one grounding of the remaining arguments can be true in every model. Disjointness keys for predicates `p/n` are declared in the beginning of a program as `#p(t1,...,tn)` with `ti=dk` denoting key arguments, `ti=av` denoting attribute value arguments, where declarations without `av` arguments are omitted.

**Example 3.17** *The following pD program is an information retrieval example inspired by [Fuhr, 2000].*

```
#year(dk,av).
#weight(av).

0.7 year(d1,2006).     0.3 year(d2,2007).
0.3 year(d1,2007).     0.6 year(d2,2008).
                       0.1 year(d2,2009).

0.8 keyword(d1,ml).     0.7 keyword(d1,lp).
0.5 keyword(d2,ml).     0.9 keyword(d2,lp).

0.7 weight(ml).     0.3 weight(lp).

0.9 relevant(D) ← year(D,Y),Y >= 2008,keyword(D,T),weight(T).
0.6 relevant(D) ← year(D,Y),Y > 2006,Y < 2008,
                  keyword(D,T),weight(T).
```

*For two documents* `d1` *and* `d2`*, there is uncertainty over their publication year. For each document, only one year is possible, but years are independent for different documents, thus, the document is the disjointness key for predicate* `year/2`*. Keywords are assigned probabilistically to each document, all these assignments are independent (the default). To use probabilistic query term weighting for retrieval, weights are associated to keywords, such that in each model only one keyword is relevant. Finally, a document is considered relevant if it contains at least one keyword, and more recent documents are more relevant. Documents can thus be ranked using the probabilities of ground instances of query* `relevant(D)`*.*

The key differences between pD and ProbLog lie in the use of functors, which is not supported in pD, and the way choices between exclusive events are specified. While ProbLog's ADs explicitly impose a shared condition on such choices, but do not restrict the form of elements in the choice, pD specifies exclusiveness on the predicate level by means of disjointness keys, but does not impose further restrictions on clauses defining the predicate. More specifically, two instantiated clause heads with the same predicate are considered disjoint if they agree on the key arguments, but differ in at least one attribute value argument, and independent else, which includes the case of identical such heads. To deal with negation, pD programs need to be *modularly stratified*, a requirement that generalizes local stratification while still guaranteeing the well-founded model to be total; we refer to [Fuhr, 2000] for the technical details.

### 3.4.6 CP-Logic

The key syntactical construct of both LPADs [Vennekens et al., 2004] and CP-logic [Vennekens, 2007] are rules of the form

$$(p_1 : \alpha_1) \vee \ldots \vee (p_n : \alpha_n) \leftarrow \varphi$$

with $\varphi$ a first order sentence, $p_i$ ground atoms, and $\alpha_i$ non-zero probabilities with $\sum_{i=1}^{n} \alpha_i \leq 1$. Non-ground rules can be used as a compact notation of all their groundings, provided that constants, predicate and function symbols are typed in a way that assures a finite number of groundings for each such rule, as the semantics is defined in terms of finite ground theories. While the definition of the semantics follows different routes in both formalisms, their equivalence has been established [Vennekens, 2007]. ProbLog's ADs directly follow the idea of these rules, but restrict rule bodies to conjunctions of literals. Furthermore, ProbLog requires rules to ensure that the well-founded model is total, cf. Section 3.3.3. However, ProbLog does not restrict non-ground ADs to finitely many groundings, as the distribution semantics also covers the case of infinitely many random variables. Characterizing the semantics of CP-logic in terms of the distribution semantics might thus be a way to drop the requirement of finitely many groundings.

## 3.5 Conclusions

In this chapter, we have introduced ProbLog, a simple yet expressive extension of Prolog with a distribution semantics. The key to inference in ProbLog is a DNF encoding of all explanations of a query; techniques to construct and evaluate such DNFs will be presented in Chapter 4. We also discussed the relations between ProbLog and a number of probabilistic logic languages with related semantics.

# Chapter 4

# The ProbLog System*

While the previous chapter has introduced the ProbLog language and the reduction to DNF at the core of inference in ProbLog, in this chapter, we discuss various ProbLog inference algorithms as well as their tight integration in the state-of-the-art YAP-Prolog system.

Section 4.1 on exact inference starts with the generation of explanations during SLD-resolution, which is then used to introduce algorithms for finding the most likely explanation or all explanations, respectively. This section is completed by discussing the disjoint-sum-problem faced in the second step of calculating the success probability. Section 4.2 introduces two groups of methods to approximate success probabilities. While bounded approximation and $k$-best use subsets of explanations to obtain DNFs of tractable size, both program sampling and DNF sampling estimate probabilities based on randomly generated interpretations. They differ in that program sampling interleaves searching for explanations with sample generation, whereas DNF sampling first constructs the DNF corresponding to all explanations and then uses it as basis for sampling.

Section 4.3 is devoted to the details of our implementation, covering the representation of labeled facts, explanations and sets of explanations, the encoding of DNFs as BDDs, and lazy sampling techniques. The different inference algorithms are experimentally compared on probabilistic networks of varying size in Section 4.4.

Finally, we discuss related work in Section 4.5, and conclude in Section 4.6.

*This chapter includes work published in [Kimmig et al., 2008, 2009, 2010; Mantadelis et al., 2010; Shterionov et al., 2010].

## 4.1 Exact Inference

In this section, we discuss exact inference methods to calculate both explanation and success probabilities based on the reduction to DNF introduced in Section 3.2. From a logic programming perspective, this DNF can be viewed as a concise representation of the successful branches in the SLD-tree. Following Prolog, we thus employ SLD-resolution to obtain all different explanations. Each successful branch in the SLD-tree relies on a set of ground probabilistic facts $\{p_1 :: f_1, \cdots, p_k :: f_k\} \subseteq L^T$ being true and a set of ground probabilistic facts $\{p_{k+1} :: f_{k+1}, \cdots, p_l :: f_l\} \subseteq L^T$ being false and thus gives rise to an explanation $E = \bigwedge_{1 \leq i \leq k} b_i \wedge \bigwedge_{k < i \leq l} \neg b_i$. These truth value assignments are necessary for the explanation, but the explanation is *independent* of truth value assignments to other probabilistic facts in $L^T$, that is, those in $E^?$.

**Example 4.1** *Figure 4.1 depicts the SLD-tree for the query* $? -$ `path(c,d)`. *in Example 3.1, where branches corresponding to resolution steps involving probabilistic facts are labeled with the corresponding random variable. The corresponding DNF* $cd \vee (ce \wedge ed)$ *is obtained by collecting these labels on succeeding branches of the tree.*

To construct explanations during SLD-resolution, we thus keep track of partial explanations as well as their probabilities. The basic resolution step is detailed in Algorithm 4.1, which distinguishes negated probabilistic facts (lines 3-10) and positive literals (lines 12-20). In both cases, the first two lines correspond to a regular SLD-resolution step. If a probabilistic fact was used, the algorithm checks whether the explanation already contains this fact with opposite sign, which would lead to inconsistency and therefore requires backtracking to alternative choices. If this is not the case, and the fact has not been used previously, the explanation and its probability are updated. Algorithm 4.2 constructs a complete explanation by repeating such resolution steps until reaching the empty goal. As for Prolog, backtracking will result in additional answers being returned, and the algorithm can thus be used to generate all explanations of a query. This algorithm is the core of ProbLog inference.

The *explanation probability* $P_x$ as given in Equation (3.15) can be calculated by successively generating all explanations using Algorithm 4.2, keeping the one with highest probability found so far[1]. However, as $P_x$ exclusively depends on the probabilistic facts used in one explanation, this potentially large search space can be pruned using a simple branch-and-bound approach based on the SLD-tree,

[1] Algorithm 4.2 could also be used to construct the DNF corresponding to the query, on which Viterbi-like dynamic programming techniques could be used to calculate the explanation probability. However, this often requires significantly more memory and should thus only be considered if the DNF is needed for other purposes as well.

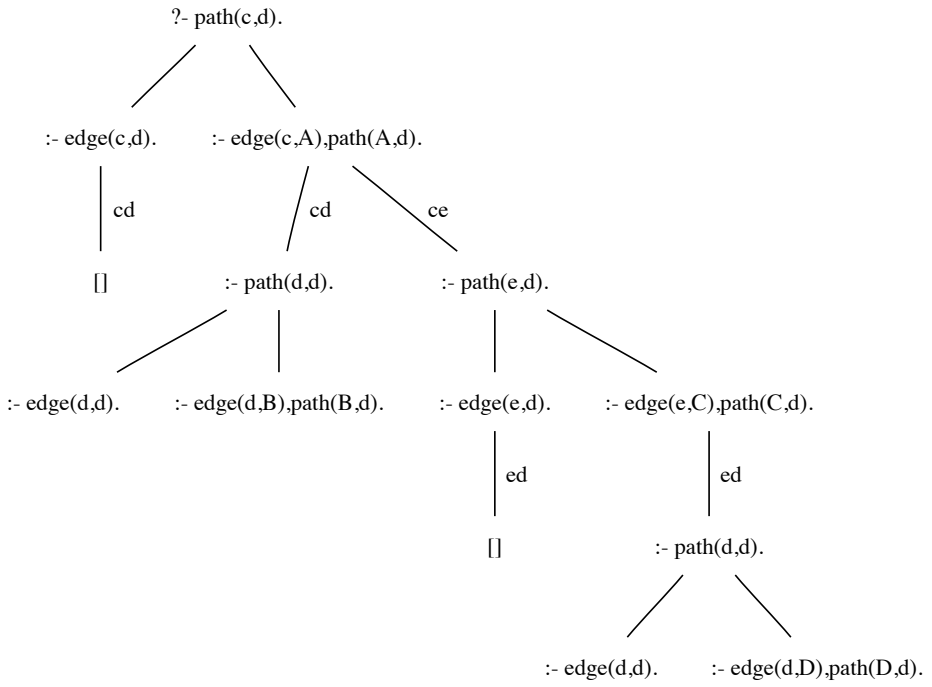



Figure 4.1: SLD-tree for query $\texttt{path}(\texttt{c},\texttt{d})$. in Example 3.1

where partial explanations are discarded if their probability drops below that of the best explanation found so far.

**Example 4.2** *Assuming that the successful leftmost branch of the SLD-tree in Figure 4.1 has already been considered, the derivations in the rightmost branch can be stopped at subgoal* $:-\ \texttt{path}(\texttt{e},\texttt{d})$*, as the partial explanation ce has probability* 0.8 *and thus cannot lead to a complete explanation with probability higher than* 0.9*, the current best value corresponding to explanation cd.*

Instead of traversing the entire SLD-tree to find all explanations, the algorithm uses iterative deepening with a probability threshold $\gamma$ to find the most likely one. Algorithm 4.3 extends Algorithm 4.2 to return a partial explanation if its probability falls below the current threshold $\gamma$ (lines 7-8). The iterative deepening search is detailed in Algorithm 4.4. Its central loop backtracks over the SLD-tree up to the current threshold, updating the current best explanation and its probability, and recording whether any stopped derivation occurred which could justify another

**Algorithm 4.1** Performing an SLD-resolution step and updating the current partial explanation and its probability.

```
1: function ResolutionStep(goal (g_0, g), probability p, explanation E)
2:   if g_0 = not(g_1) then
3:     select next probabilistic fact f matching g_1
4:     θ := mgu(g_1, f); g' := gθ
5:     if fθ ∈ E then
6:       backtrack to fact selection
7:     else if ¬fθ ∉ E then
8:       E := E ∧ ¬fθ
9:       p := p · (1 − prob(f))
10:    return (g', p, E)
11:  else
12:    select next clause c matching g_0
13:    θ := mgu(g_0, head(c)); g' := (body(c)θ, gθ)
14:    if c probabilistic fact then
15:      if ¬cθ ∈ E then
16:        backtrack to clause selection
17:      else if cθ ∉ E then
18:        E := E ∧ cθ
19:        p := p · prob(c)
20:    return (g', p, E)
```

**Algorithm 4.2** SLD-resolution in ProbLog.

```
1: function Resolve(query q)
2:   p := 1; E :=True
3:   repeat
4:     (q, p, E) :=ResolutionStep(q, p, E)
5:     if q = ∅ then
6:       return (success, E, p)
7:   until no further resolution possible
8:   return (fail,False, 0)
```

iteration. The algorithm stops after the first iteration where an explanation has been found, as all stopped derivations could only lead to explanations with lower probability. The minimum threshold $\epsilon$ avoids exploring infinite SLD-trees without solution.

Calculating the *success probability* $P_s$, on the other hand, is more complex. As sketched in Section 3.2, we first construct the DNF formula corresponding to the set of explanations and then calculate the probability of this DNF, which involves dealing with the disjoint-sum-problem as discussed in the next section. The first

**Algorithm 4.3** SLD-resolution in ProbLog using minimal probability $\gamma$.

```
1: function ResolveThreshold(query q, threshold γ)
2:   p := 1; E := True
3:   repeat
4:     (q, p, E) := ResolutionStep(q, p, E)
5:     if q = ∅ then
6:       return (success, E, p)
7:     else if p < γ then
8:       return (stop, E, p)
9:   until no further resolution possible
10:  return (fail, False, 0)
```

**Algorithm 4.4** Calculating the most likely explanation by iterative deepening search in the SLD-tree.

```
1: function BestProb(query q, thresholds γ and ε, constant β ∈ (0, 1))
2:   max = −1; best := False; continue := True
3:   while (γ > ε) ∧ continue do
4:     continue := False
5:     repeat
6:       (result, E, p) := ResolveThreshold(q, γ)
7:       if (result = success) ∧ (p > max) then
8:         max := p; best := E
9:       else
10:        if result = stop then
11:          continue := True
12:      backtrack to the remaining choice points of ResolveThreshold
13:    until ResolveThreshold has no choice points remaining
14:    if max > −1 then
15:      return (max, best)
16:    else
17:      γ := β · γ
18:  if γ ≤ ε then
19:    return (−1, stop)
20:  else
21:    return (0, unprovable)
```

step simply uses backtracking to obtain all explanations from Algorithm 4.2.

### 4.1.1 The Disjoint-Sum-Problem

As already stated in Section 3.2, the *disjoint-sum-problem* is the problem of transforming an arbitrary DNF formula into one with mutually exclusive conjunctions only. In the context of ProbLog, the latter type of DNF could be used to calculate its probability as a sum of products mirroring the structure of the DNF. This is closely related to the two-terminal network reliability problem, which asks for the probability of two nodes being connected in a communication network with probabilistically failing components. This type of problem has been shown to be #P-complete [Valiant, 1979].

**Example 4.3** *In the case of the two explanations $ce \wedge ed$ and $cd$ of Example 3.4, the set of complete interpretations where $cd \vee (ce \wedge ed)$ is true can be split into three disjoint sets: those interpretations where $cd$ is true and $ce \wedge ed$ is false, those where $cd$ is false and $ce \wedge ed$ is true, and those where both explanations are true. This last set contributes to the probabilities of both explanations, which are $P^T(cd) = 0.9$ and $P^T(ce \wedge ed) = 0.8 \cdot 0.5 = 0.4$. Summing the probabilities of the explanations would thus count the contribution of the third set twice, while it is only counted once in the success probability. Indeed, the difference between the sum of the probabilities of the two explanations and the success probability is $(0.4 + 0.9) - 0.94 = 0.36$, which is exactly the probability of the last set, $P(ce \wedge ed \wedge cd) = 0.4 \cdot 0.9$.*

Example 4.3 indicates one way of tackling the disjoint-sum-problem, namely the *inclusion-exclusion-principle*, which calculates the probability by taking into account all combinations of conjunctions:

$$P(E_1 \vee \ldots \vee E_n) = \sum_{1 \leq j \leq n} \sum_{\substack{S \subseteq \{1,\ldots,n\} \\ |S|=j}} (-1)^{j+1} \cdot P\left(\bigwedge_{i \in S} E_i\right) \tag{4.1}$$

While this approach is followed for instance in the implementation of pD, cf. Section 4.5, it clearly does not scale to larger DNF. An alternative solution strategy is to add *additional literals* to a conjunction, thereby explicitly excluding interpretations covered by another part of the formula from the corresponding part of the sum.

**Example 4.4** *In Example 4.3, extending $ce \wedge ed$ to $ce \wedge ed \wedge \neg cd$ reduces the second part of the sum to those interpretations not covered by the first:*

$$\begin{aligned} P_s^T(\texttt{path(c,d)}) &= P^T(cd \vee (ce \wedge ed)) \\ &= P^T(cd) + P^T(ce \wedge ed \wedge \neg cd) \\ &= 0.9 + 0.8 \cdot 0.5 \cdot (1 - 0.9) = 0.94 \end{aligned}$$

**Algorithm 4.5** Calculating the probability of a BDD.

```
1: function PROBABILITY(BDD node n)
2:   if n is the 1-terminal then
3:     return 1
4:   if  n is the 0-terminal then
5:     return 0
6:   let h and l be the high and low children of n
7:   prob(h) := PROBABILITY(h)
8:   prob(l) := PROBABILITY(l)
9:   return p_n · prob(h) + (1 − p_n) · prob(l)
```

However, as the number of explanations grows, disjoining them gets more involved, as the overlap between complete interpretations has to be considered for each pair of explanations: once the first two explanations are disjoint, the third one needs to be made disjoint with respect to both of them, the fourth one with respect to the first three, and so forth. A variety of methods based on this idea has been studied especially in the field of network reliability. Poole [2000] introduces one such technique for ICL.

For ProbLog, we typically are not interested in the resulting formula, but only in its probability. We therefore follow an alternative approach that avoids explicit manipulation of the formula, but instead encodes it as binary decision diagram (BDD, cf. Section 2.4). BDDs encode Boolean formulae in a way that implicitly solves the disjoint-sum-problem. Given such a BDD, the probability is calculated by traversing the BDD, in each node summing the probability of the high and low child, weighted by the probability of the node's variable being assigned true and false respectively, cf. Algorithm 4.5. Intermediate results are cached, and the algorithm has a time and space complexity linear in the size of the BDD.

**Example 4.5** *Figure 4.2 shows the BDD corresponding to the DNF formula* $cd \vee (ce \wedge ed)$ *of Example 4.3. The algorithm starts by assigning probabilities* 0 *and* 1 *to the* 0- *and* 1*-leaf respectively. The node labeled ed has probability* $0.5 \cdot 1 + 0.5 \cdot 0 = 0.5$*, node ce has probability* $0.8 \cdot 0.5 + 0.2 \cdot 0 = 0.4$*; finally, node cd, and thus the entire formula, has probability* $0.9 \cdot 1 + 0.1 \cdot 0.4 = 0.94$.

In Section 4.2.2, we will discuss a Monte Carlo approach addressing the disjoint-sum-problem.

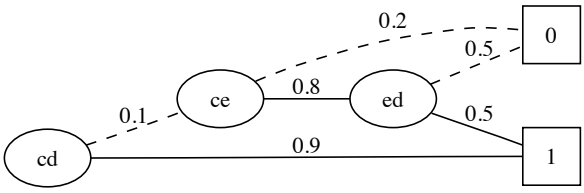



Figure 4.2: Binary decision diagram encoding the DNF formula $cd \vee (ce \wedge ed)$, corresponding to the two explanations of query `path(c,d)` in the example graph of Figure 3.1. An internal node labeled $xy$ represents the Boolean variable for the edge between $x$ and $y$, solid/dashed edges correspond to values true/false and are labeled with the probability that the variable takes this value.

## 4.2 Approximative Inference

As the size of the DNF formula grows with the number of explanations, its evaluation can become quite expensive, and ultimately infeasible. For instance, in the context of analyzing networks, even in small networks with a few dozen edges there are easily hundreds of thousands of possible paths between two nodes. ProbLog therefore includes several approximation methods that rely either on reducing the size of the DNF or on Monte Carlo techniques.

### 4.2.1 Using Less Explanations

We first discuss approximation techniques that reduce the size of the DNF by considering a subset of all possible explanations. We here exploit the fact that the DNF formula describing sets of explanations is *monotone*, meaning that adding more explanations will never decrease the probability of the formula being true. Thus, formulae describing subsets of the full set of explanations of a query will always give a lower bound on the query's success probability.

**Example 4.6** *In Example 3.1, the lower bound obtained from the shorter explanation would be* $P^T(cd) = 0.9$*, while that from the longer one would be* $P^T(ce \wedge ed) = 0.4$*.*

### Bounded Approximation

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

We observe that the probability of an explanation $l_1 \wedge \ldots \wedge l_n$, where the $l_i$ are positive or negative literals involving random variables $b_i$, will always be at most the probability of an arbitrary prefix $l_1 \wedge \ldots \wedge l_i, i \leq n$.

**Example 4.7** *In the graph example, the probability of the second explanation will be at most the probability of its first edge from c to e, i.e.,* $P^T(ce) = 0.8 \geq 0.4$.

As disjoining sets of explanations, i.e., including information on additional facts, can only decrease the contribution of single explanations, this upper bound carries over to a set of explanations or partial explanations, as long as prefixes for all possible explanations are included. Such sets can be obtained from an incomplete SLD-tree, i.e., an SLD-tree where branches are only extended up to a certain point.

These observations motivate ProbLog's *bounded approximation algorithm.* The algorithm relies on a probability threshold $\gamma$ to stop growing the SLD-tree and thus obtain DNF formulae for the two bounds[2]. The lower bound formula $D_1$ represents all explanations with a probability above the current threshold. The upper bound formula $D_2$ additionally includes all derivations that have been stopped due to reaching the threshold, as these still *may* succeed. Our goal is therefore to refine $D_1$ and $D_2$ in order to decrease $P^T(D_2) - P^T(D_1)$.

Bounded approximation as outlined in Algorithm 4.6 proceeds in an iterative-deepening manner similar to Algorithm 4.4, but collecting explanations in the two DNF formulae $D_1$ and $D_2$ instead of remembering the most likely explanation only. Initially, both $D_1$ and $D_2$ are set to FALSE, the neutral element with respect to disjunction, and the probability bounds are 0 and 1, as we have no full explanations yet, and the empty partial explanation holds in any model. After each iteration, BDDs for both formulae are constructed to calculate their probabilities using Algorithm 4.5, and iterative deepening stops once their difference falls below the stopping threshold $\delta$. It should be clear that $P^T(D_1)$ monotonically increases, as the number of explanations never decreases. On the other hand, as explained above, if $D_2$ changes from one iteration to the next, this is always because a partial explanation $E$ is either removed from $D_2$ and therefore no longer contributes to the probability, or it is replaced by explanations $E_1, \ldots, E_n$ that extend $E$ by additional literals, that is, $E_i = E \wedge S_i$ for conjunctions $S_i$, hence

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

**Algorithm 4.6** Bounded approximation using iterative deepening with probability thresholds.

```
1: function BOUNDS(query q, interval width δ, initial threshold γ, constant
   β ∈ (0,1))
2:   D1 := FALSE; P1 := 0; P2 := 1
3:   repeat
4:     D2 := FALSE
5:     repeat
6:       (result, E, p) := RESOLVETHRESHOLD(q, γ)
7:       if result = success then
8:         D1 := D1 ∨ E ; D2 := D2 ∨ E
9:       if result = stop then
10:        D2 := D2 ∨ E
11:      backtrack to the remaining choice points of RESOLVETHRESHOLD
12:    until RESOLVETHRESHOLD has no choice points remaining
13:    Construct BDDs B1 and B2 corresponding to D1 and D2
14:    P1 := PROBABILITY(root(B1))
15:    P2 := PROBABILITY(root(B2))
16:    γ := γ · β
17:  until P2 − P1 ≤ δ
18:  return [P1, P2]
```

$P^T(E_1 \vee \ldots \vee E_n) = P^T(E \wedge S_1 \vee \ldots \vee E \wedge S_n) = P^T(E \wedge (S_1 \vee \ldots \vee S_n))$. As explanations are partial interpretations of the probabilistic facts in the ProbLog program, each literal's random variable appears at most once in the conjunction representing an explanation, even if the corresponding subgoal is called multiple times during construction. We therefore know that the literals in the prefix $E$ cannot be in any suffix $S_i$, hence, given ProbLog's independence assumption, $P^T(E \wedge (S_1 \vee \ldots \vee S_n)) = P^T(E)P^T(S_1 \vee \ldots \vee S_n) \leq P^T(E)$. Therefore, $P(D_2)$ monotonically decreases.

**Example 4.8** *Consider a probability threshold* $\gamma = 0.9$ *for the SLD-tree in Figure 4.1. In this case,* $D_1$ *encodes the left success path while* $D_2$ *additionally encodes the path up to* `path(e,d)`*, i.e.,* $D_1 = cd$ *and* $D_2 = cd \vee ce$*, whereas the formula for the full SLD-tree is* $D = cd \vee (ce \wedge ed)$*. The lower bound thus is* $0.9$*, the upper bound (obtained by disjoining* $D_2$ *to* $cd \vee (ce \wedge \neg cd)$*) is* $0.98$*, whereas the true probability is* $0.94$*.*

### K-Best

Using a fixed number of explanations 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 as discussed in Chapter 7. We therefore introduce the $k$-probability $P_k^T(q)$, which approximates the success probability by using the $k$-best (that is, the $k$ most likely) explanations instead of all explanations when building the DNF formula used in Equation (3.25):

$$P_k^T(q) = P\left(\bigvee_{E \in Expl_k(q)} \bigwedge_{f_i \in E^1} b_i \wedge \bigwedge_{f_i \in E^0} \neg b_i\right) \tag{4.2}$$

where $Expl_k(q) = \{E \in Expl^T(q) | P^T(E) \geq P^T(E_k)\}$ with $E_k$ the $k$th element of $Expl^T(q)$ sorted by non-increasing probability. Setting $k = \infty$ leads to the success probability, whereas $k = 1$ corresponds to the explanation probability provided that there is a single best explanation. The branch-and-bound approach used to calculate the explanation probability can directly be generalized to finding the $k$-best explanations; cf. also [Poole, 1993a].

**Example 4.9** *Consider again our example graph, but this time with query* `path(a,d)`*. This query has four explanations* $ac \wedge cd$*,* $ab \wedge bc \wedge cd$*,* $ac \wedge ce \wedge ed$ *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* $P_1(\texttt{path(a,d)}) = 0.72$*. For* $k = 2$*, the overlap between the best two explanations has to be taken into account: the second explanation 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* $P_2(\texttt{path(a,d)}) = P((ac \wedge cd) \vee (\neg ac \wedge ab \wedge bc \wedge cd)) = 0.72 + (1 - 0.8) \cdot 0.378 = 0.7956$*. Similarly, we obtain* $P_3(\texttt{path(a,d)}) = 0.8276$ *and* $P_k(\texttt{path(a,d)}) = 0.83096$ *for* $k \geq 4$*.*

### 4.2.2 Monte Carlo Methods

In this section, we discuss two basic sampling methods for ProbLog inference. In both cases, the success probability is estimated as the fraction of "positives" among a large number of independently sampled possible worlds, that is, interpretations of the probabilistic facts. While the first approach directly uses the ProbLog program to sample interpretations and to check whether the query is entailed, the second approach first builds the DNF corresponding to the query as for exact inference, and then uses sampling to address the disjoint-sum-problem, that is, as an alternative to BDD construction.

#### Program Sampling

In [Kimmig et al., 2008], we proposed a first Monte Carlo method based on the basic distribution over interpretations. Here, we call this method *program sampling*

**Algorithm 4.7** Program sampling.

```
1: function ProgramSampling(query q, interval width δ, constant m)
2:   c = 0; i = 0; p = 0; Δ = 1
3:   while Δ > δ do
4:     Generate a sample I using Equation (3.11)
5:     if I ⊨ q then
6:       c := c + 1
7:     i := i + 1
8:     if i mod m == 0 then
9:       p := c/i
10:      Δ := 2 × sqrt((p·(1−p))/i)
11:  return p
```

to distinguish it from the second Monte Carlo method introduced in [Shterionov et al., 2010] and discussed below. Given a query $q$, program sampling repeats the following steps until convergence:

1. Sample an interpretation $I$ of $L^T$ from the distribution defined in Equation (3.11)
2. Consider $I$ a positive sample if $q$ is true in the canonical model extending $I$
3. Estimate the query probability $P$ as the fraction of positive samples

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

$$\delta \approx 2 \times \sqrt{\frac{P \cdot (1-P)}{N}} \tag{4.3}$$

Notice that confidence intervals do not directly correspond to the exact bounds used in bounded approximation. Still, we employ the same stopping criterion, that is, we run the Monte Carlo simulation until the width of the confidence interval is at most $\delta$. The algorithm is summarized in Algorithm 4.7.

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

query. Investigating similar probabilistic backtracking approaches for ProbLog is a promising future research direction.

### DNF Sampling

In this section, we introduce a second Monte Carlo method for ProbLog using the DNF to focus sampling on those interpretations where the query is true. It builds upon the Monte Carlo algorithm of Karp and Luby [1983]. The key idea is to first obtain the DNF as for exact inference, but to then use sampling to tackle the disjoint-sum-problem. To do so, each interpretation making the query true is assigned to exactly one of the explanations it extends. A sample then is a pair of an explanation and a complete interpretation extending this explanation, where those samples that respect the assignment are considered positive. The fraction of positive samples is an estimate of the ratio of the success probability to the sum of the probabilities of all explanations. We will first illustrate the idea by means of an example.

**Example 4.10** *Consider the DNF*

$$F = (a \wedge b \wedge c) \vee (b \wedge c \wedge d) \vee (b \wedge d \wedge e)$$

*with a probability of* 0.5 *for each random variable. Figure 4.3 shows the interpretations extending each of the three explanations. The sum of the explanations' probabilities is* $S(F) = 3 \cdot 0.125 = 0.375$*. However, the explanations are not mutually exclusive: all of them are true in world* $\{a, b, c, d, e\}$*, and two of them are true in worlds* $\{a, b, c, d\}$ *and* $\{b, c, d, e\}$*. In total, there are* 8 *different interpretations to be taken into account for the probability of* $F$*, which therefore is only* $P(F) = 8 \cdot 0.03125 = 0.25$*.*

*Most worlds extend only a single explanation and hence are associated with it. For the others, an arbitrary choice is made, e.g. for the first explanation they extend. This associates worlds* $\{a, b, c, d, e\}$ *and* $\{a, b, c, d\}$ *with* $a \wedge b \wedge c$*, and world* $\{b, c, d, e\}$ *with* $b \wedge c \wedge d$*. Samples are generated by first sampling an explanation* $E_i$ *with probability* $P(E_i)/S(F) = 1/3$*, and then extending* $E_i$ *by sampling truth values for remaining variables. For instance, we could obtain* $b \wedge c \wedge d$ *and world* $\{a, b, c, d\}$ *in this way, but as world* $\{a, b, c, d\}$ *is associated to* $a \wedge b \wedge c$*, this sample would be considered negative, whereas* $a \wedge b \wedge c$ *together with world* $\{a, b, c, d\}$ *would be considered positive. Informally speaking, the sampling procedure thus rejects half of the contribution of world* $\{a, b, c, d\}$*, thereby compensating for the two explanations it contains.*

The first phase of DNF sampling constructs the DNF as done in exact inference. From now on, we assume this DNF to be $D = E_1 \vee \ldots \vee E_n$. A key concept in

| | | | | |
|---|---|---|---|---|
| **a** | **b** | **c** | d | e |
| **a** | **b** | **c** | d | |
| **a** | **b** | **c** | | e |
| **a** | **b** | **c** | | |

| | | | | |
|---|---|---|---|---|
| a | **b** | **c** | **d** | e |
| a | **b** | **c** | **d** | |
| | **b** | **c** | **d** | e |
| | **b** | **c** | **d** | |

| | | | | |
|---|---|---|---|---|
| a | **b** | c | **d** | **e** |
| a | **b** | | **d** | **e** |
| | **b** | c | **d** | **e** |
| | **b** | | **d** | **e** |

Figure 4.3: Possible worlds where the conjunctions of the DNF of Example 4.10 are true. Each column corresponds to one conjunction, whose variables are marked in bold.

DNF sampling is the *sum of probabilities* for a DNF $E_1 \vee \ldots \vee E_n$, which is defined as

$$S_x^T(E_1 \vee \ldots \vee E_n) = \sum_{i=0}^{n} \prod_{f_j \in E_i^1} p_j \prod_{f_j \in E_i^0} (1 - p_j) \tag{4.4}$$

Note that if explanations are mutually exclusive, $S_x^T(D)$ is equal to the probability of $D$ being true, but it can be much higher in general. One way to remove overlap between explanations is to extend the corresponding formulae with additional terms. Here, we will use the following extension that assigns each complete interpretation to the first explanation it extends:

$$E_1 \vee \ldots \vee E_n = \bigvee_{i=1}^{n} \left( E_i \wedge \neg \bigvee_{j=1}^{i-1} E_j \right) \tag{4.5}$$

The success probability thus corresponds to the sum of the probabilities of these formulae. However, calculating those again involves the disjoint-sum-problem, while the sum of probabilities can be calculated easily. We therefore use sampling to instead estimate the ratio

$$\frac{P^T(E_1 \vee \ldots \vee E_n)}{S_x^T(E_1 \vee \ldots \vee E_n)} = \frac{\sum_{i=0}^{n} P^T\left(E_i \wedge \neg \bigvee_{j=1}^{i-1} E_j\right)}{\sum_{i=0}^{n} P^T(E_i)} \tag{4.6}$$

To do so, we sample from the set $U$ containing all pairs of explanations $E_i$ and their completions, cf. Equation (3.10), that is,

$$U(E_1 \vee \ldots \vee E_n) = \{(I, i) \mid I \in Compl^T(E_i)\} = \{(I, i) \mid I \models E_i\}. \tag{4.7}$$

We accept those samples that combine interpretations with the first explanation they extend, that is, those from

$$A(E_1 \vee \ldots \vee E_n) = \{(I, i) \mid I \models E_i \wedge \forall j < i : I \not\models E_j\} \tag{4.8}$$

$$= \{(I, i) \mid I \models (E_i \wedge \neg \bigvee_{j=1}^{i-1} E_j)\} \tag{4.9}$$

**Algorithm 4.8** DNF sampling, where $A$ and $S_x^T$ are given by Equations (4.8) and (4.4), respectively, and DNF backtracks over Algorithm 4.2 to generate the DNF.

```
1: function DnfSampling(query q, interval width δ, constant m)
2:    E_1 ∨ ... ∨ E_n := Dnf(q)
3:    S := S_x^T(E_1 ∨ ... ∨ E_n)
4:    c = 0; i = 0; p = 0; Δ = 1
5:    while Δ > δ do
6:       Sample E_i according to P^T(E_i)/S
7:       Sample I ∈ Compl^T(E_i) using Equation (3.11)
8:       if (I, i) ∈ A(E_1 ∨ ... ∨ E_n) then
9:          c := c + 1
10:      i := i + 1
11:      if i mod m == 0 then
12:         p := c/i · S
13:         Δ := 2 × sqrt((p·(1−p))/i)
14:   return p
```

DNF sampling as outlined in Algorithm 4.8 thus first constructs the DNF $D$ corresponding to the query's explanations and calculates the corresponding sum of probabilities $S_x^T(D)$. The sampling phase then uses the same convergence criteria as Algorithm 4.7. A sample is generated by sampling an explanation $E_i$ from $D$ using the normalized distribution $P^T(E_i)/S_x^T(D)$ and randomly extending it into an interpretation $I$ from $Compl^T(E_i)$ according to the distribution over the remaining probabilistic facts, that is,

$$P_U((I,i)) = \frac{P(E_i)}{S_x^T(E_1 \vee \ldots \vee E_n)} \cdot \prod_{f_j \in I^1 \setminus E_i^1} p_j \prod_{f_j \in I^0 \setminus E_i^0} (1 - p_j). \tag{4.10}$$

The sample is accepted if it belongs to $A$. Based on Equation (4.6), after $N$ samples, $N_{pos}$ of which are positive, the probability of formula $D$ is estimated as

$$P_{DNF}^T(q) = S_x^T(D) \cdot \frac{N_{pos}}{N} \tag{4.11}$$

Note that depending on the structure of the problem and the value of $S_x^T(D)$, estimates based on small numbers of samples may not be probabilities yet, that is, may be larger than one, especially if the actual probability is close to one. This is due to the fact that a sufficient number of samples is needed to identify overlap between conjunctions by means of sampling and to accordingly scale down the overestimate $S_x^T(D)$.

The probability $P_U((I,i))$ of sampling an interpretation $I$ based on the $i$th explanation is proportional to the probability $P^T(I)$ of sampling $I$ from the

ProbLog program. By construction of $A$, the probability of sampling a positive pair $(I, i)$ is

$$P_A((I,i)) = \frac{P^T(E_i \wedge \neg \bigvee_{j=1\ldots i-1} E_j)}{S_x^T(E_1 \vee \ldots \vee E_n)}. \tag{4.12}$$

As each interpretation is positive for exactly one explanation, the probability of accepting a sample is the sum of this probability over all explanations

$$P_A = \frac{P(E_1) + P(E_2 \wedge \neg E_1) + \ldots + P(E_n \wedge \neg(E_1 \vee \ldots \vee E_{n-1}))}{S_x^T(E_1 \vee \ldots \vee E_n))} \tag{4.13}$$

For a sufficiently large number $N$ of samples we thus expect the estimate of Equation (4.11) to be

$$\begin{aligned} P_{DNF} =& \frac{N \cdot P_A}{N} \cdot S_x^T(E_1 \vee \ldots \vee E_n) \\ =& P(E_1) + P(E_2 \wedge \neg E_1) + \ldots + P(E_n \wedge \neg(E_1 \vee \ldots \vee E_{n-1})), \end{aligned}$$

which indeed is the disjoint sum corresponding to Equation (4.5). As mentioned above, DNF sampling is an instance of the fully polynomial approximation scheme of Karp and Luby [1983], that is, the number of samples required for a given level of certainty is polynomial in the input length (the DNF in our case), and a corresponding stopping criterion could thus be used; for formal detail, we refer to [Karp and Luby, 1983]. For ease of comparison, we have chosen to use the same stopping criterion for both sampling approaches presented here.

## 4.3 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 4.4, with a typical ProbLog program, including ProbLog facts and background knowledge, at the top.

The implementation requires ProbLog programs to use the `problog` module. Each program consists of a set of labeled facts and of unlabeled background knowledge, a generic Prolog program. Labeled facts are preprocessed as described below. Notice that the implementation requires all queries to non-ground probabilistic facts to be ground on calling. Furthermore, while the semantics of ProbLog requires that probabilistic facts cannot unify with each other or with heads of background knowledge clauses, cf. Section 3.1, this is not enforced in the implementation.

In contrast to standard Prolog queries, where one is interested in answer substitutions, in ProbLog one is primarily interested in a probability. As discussed before, two common ProbLog queries ask for the most likely explanation and

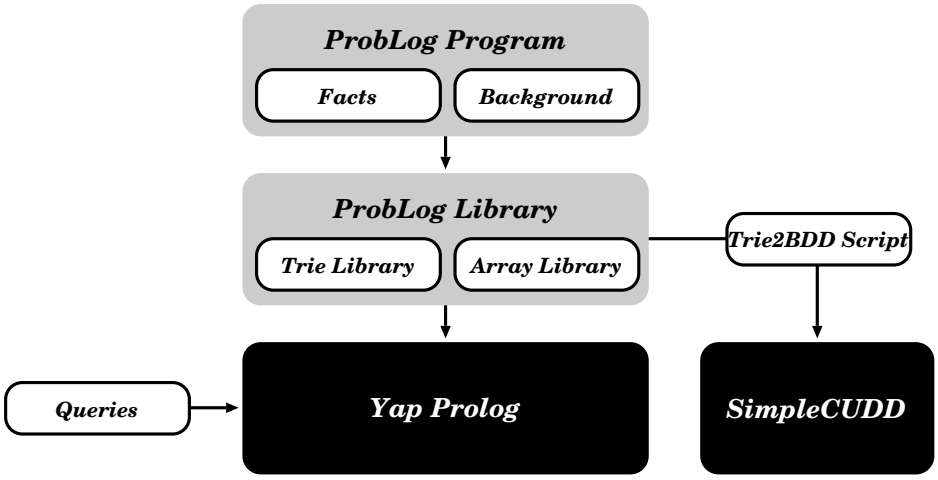



Figure 4.4: ProbLog implementation: A ProbLog program (top) requires the ProbLog library which in turn relies on functionality from the tries and array libraries. ProbLog queries (bottom-left) are sent to the YAP engine, and may require calling the BDD library CUDD via SimpleCUDD.

its probability, and the probability of whether a query would have an answer substitution. We have discussed two very different approaches to the problem:

- In exact inference, $k$-best and bounded approximation, the engine explicitly reasons about probabilities of proofs. The challenge is how to compute the probability of each individual proof, store a large number of proofs, and compute the probability of sets of proofs.
- In Monte Carlo, the probabilities of facts are used to sample from ProbLog programs. The challenge is how to compute a sample quickly, in a way that inference can be as efficient as possible.

ProbLog programs execute from a top-level query and are driven through a ProbLog query. The inference algorithms discussed above can be abstracted as follows:

- Initialise the inference algorithm;
- While probabilistic inference did not converge:
  - execute the query, instrumenting every ProbLog call in the current proof, cf. Algorithm 4.1. Instrumentation is required for recording the ProbLog facts required by a proof, but may also be used by the inference algorithm to stop proofs (e.g. if the current probability is lower than a bound);

  - process result (e.g. add current proof to set of proofs);
  - backtrack to alternative proofs;
- Proceed to the next step of the algorithm: this may be trivial or may require calling an external solver, such as a BDD tool, to compute a probability.

Notice that the current ProbLog implementation relies on the Prolog engine to efficiently execute goals. On the other hand, and in contrast to most other probabilistic language implementations, in ProbLog there is no clear separation between logical and probabilistic inference: in a fashion similar to constraint logic programming, probabilistic inference can drive logical inference.

From a Prolog implementation perspective, ProbLog poses a number of interesting challenges. First, labeled facts have to be efficiently compiled to allow mutual calls between the Prolog program and the ProbLog engine. Second, for exact inference, $k$-best and bounded approximation, sets of explanations have to be manipulated and transformed into BDDs for probability calculation. Finally, Monte Carlo simulation requires representing and manipulating samples. We discuss these issues next.

### 4.3.1 Labeled Facts

Following the approach of 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.

**Example 4.11** *As an example, the program:*

```
0.715 :: edge('PubMed_2196878','MIM_609065').
0.659 :: edge('PubMed_8764571','HGNC_5014').
```

*would be compiled as:*

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

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

Thus, the internal representation of each fact contains an identifier, the original arguments, and the logarithm of the probability[3]. The `grounding_id` procedure

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

will create and store a grounding specific identifier for each new grounding of a non-ground probabilistic fact encountered during proving, and retrieve it on repeated use. For ground probabilistic facts, it simply returns the identifier itself. The `add_to_explanation` procedure updates the data structure representing the current path through the search space, that is, a queue of identifiers ordered by first use, together with its probability. It thus implements the updates of $E$ and $p$ in Algorithm 4.1. Compared to the original meta-interpreter based implementation of [De Raedt et al., 2007b], the main benefit of source-to-source transformation is better scalability, namely by having a compact representation of the facts for the YAP engine [Santos Costa, 2007] and by allowing access to the YAP indexing mechanism [Santos Costa et al., 2007].

### 4.3.2 Explanations

Manipulating explanations is critical in ProbLog. We represent each explanation as a queue containing the identifier of each different ground probabilistic fact used in the explanation, ordered by first use. The implementation requires calls to non-ground probabilistic facts to be ground, and during proving maintains a table of groundings used within the current query together with their identifiers. Grounding identifiers are based on the fact's identifier extended with a grounding number, i.e. 5_1 and 5_2 would refer to different groundings of the non-ground fact with identifier 5. Extending this to negated probabilistic facts is straightforward: the negated identifier is included in the set in this case, and proving fails if both cases occur for the same fact. In our implementation, the queue is stored in a backtrackable global variable, which is updated by calling `add_to_explanation` with an identifier for the current ProbLog fact. We thus exploit Prolog's backtracking mechanism to avoid recomputation of shared explanation prefixes when exploring the space of proofs using Algorithm 4.2. Storing an explanation is simply a question of adding the value of the variable to a store.

As we have discussed above, the actual number of explanations can grow very quickly. ProbLog compactly represents an explanation as a list of numbers. To calculate or approximate the success probability, we would further like to have a scalable implementation of *sets* of explanations, such that we can compute the joint *probability* of large sets of explanations efficiently. Our representation for sets of explanations and our algorithm for encoding such sets as BDDs for probability calculation are discussed next.

### 4.3.3 Sets of Explanations

Storing and manipulating explanations is critical in ProbLog. When manipulating explanations, the key operation is often *insertion*: we would like to add an

explanation to an existing set of explanations. Some algorithms, such as exact inference or Monte Carlo, only manipulate complete explanations. Others, such as bounded approximation, require adding partial derivations too. The nature of the SLD-tree means that explanations tend to share both a prefix and a suffix. Partial explanations tend to share prefixes only. This suggests using *tries* to maintain the set of explanations. We use the YAP implementation of tries for this task, based itself on XSB Prolog's work on tries of terms [Ramakrishnan et al., 1999], which we briefly summarize here.

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

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

In order to minimize the number of nodes when storing explanations in a trie, we use Prolog lists to represent explanations. List elements in explanations are always either integers (for ground facts), two integers with an underscore in between (for groundings of facts), or the negation `not(i)` of such an identifier `i`.

**Example 4.12** *Figure 4.5 presents an example of a trie storing three explanations. Initially, the trie contains the root node only. Next, we store the explanation* $[3, 5\_1, 7, 5\_2]$ *containing ground facts* `3` *and* `7` *as well as two groundings of the fact* `5`*. Six nodes (corresponding to six tokens) are added to represent it (Figure 4.5(a)). The explanation* $[3, 5\_1, 9, 7, 5\_2]$ *is then stored which requires seven nodes. As it shares a common prefix with the previous explanation, we save the three initial nodes common to both representations (Figure 4.5(b)). The explanation* $[3, 4, 7]$ *is*

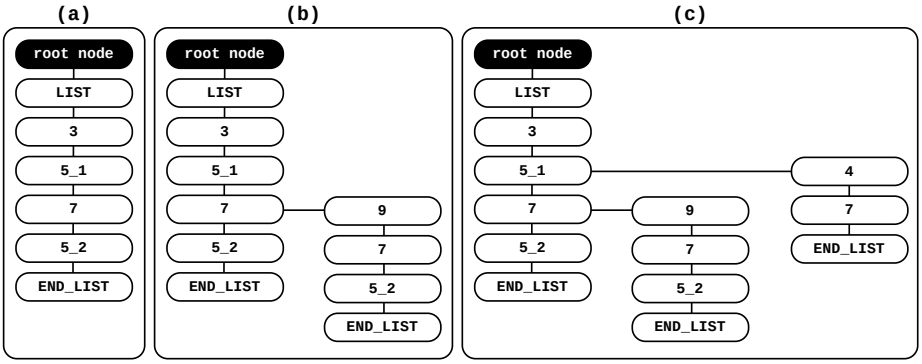



Figure 4.5: Using tries to store explanations. Initially, the trie contains the root node only. Next, we store the explanations: (a) $[3, 5\_1, 7, 5\_2]$; (b) $[3, 5\_1, 9, 7, 5\_2]$; and (c) $[3, 4, 7]$.

*stored next and we save again the two initial nodes common to all explanations (Figure 4.5(c)).*

To adapt exact inference for negation beyond probabilistic facts using the idea of negated subformulae as sketched in Section 3.3.3, the single trie is replaced by a hierarchy of tries [Kimmig et al., 2009]. Trie nodes are labeled either with a (possibly negated) fact identifier as before, or with a negated reference to another trie. Basically, on encountering a ground subgoal `not(Q)`, where `Q` is not a probabilistic fact, the current state of proving is suspended, and a new trie is used to solve `Q` as if it was an independent query. Once this trie is completed, the proof of the calling goal is resumed with a negated reference to `Q`'s trie added. To avoid repeated building of such subformulae, we use the same reference for reoccurring subgoals, thereby realising a simple restricted form of tabling. Note that such a trie no longer directly represents a DNF. However, as BDDs represent arbitrary propositional formulae, this is not a problem in practice. Using formulae that are not in DNF makes the first step conceptually simple, although further investigation is needed to obtain a clear idea of the price to be paid in the form of memory requirements. Mantadelis and Janssens [2010] follow a similar approach of nested tries for tabled ProbLog, where each tabled subgoal has its own trie which is referenced from other proofs requiring the subgoal.

**Example 4.13** *Figure 4.6 illustrates nested tries in the context of Example 3.10. Figure 4.6(a) shows the trie for query* `alarm`*. Figure 4.6(c) shows the trie for query* `johnCalls` *if nesting is only used for negated subgoals, Figure 4.6(b) additionally reuses the trie for the positive subgoal as in tabled ProbLog.*

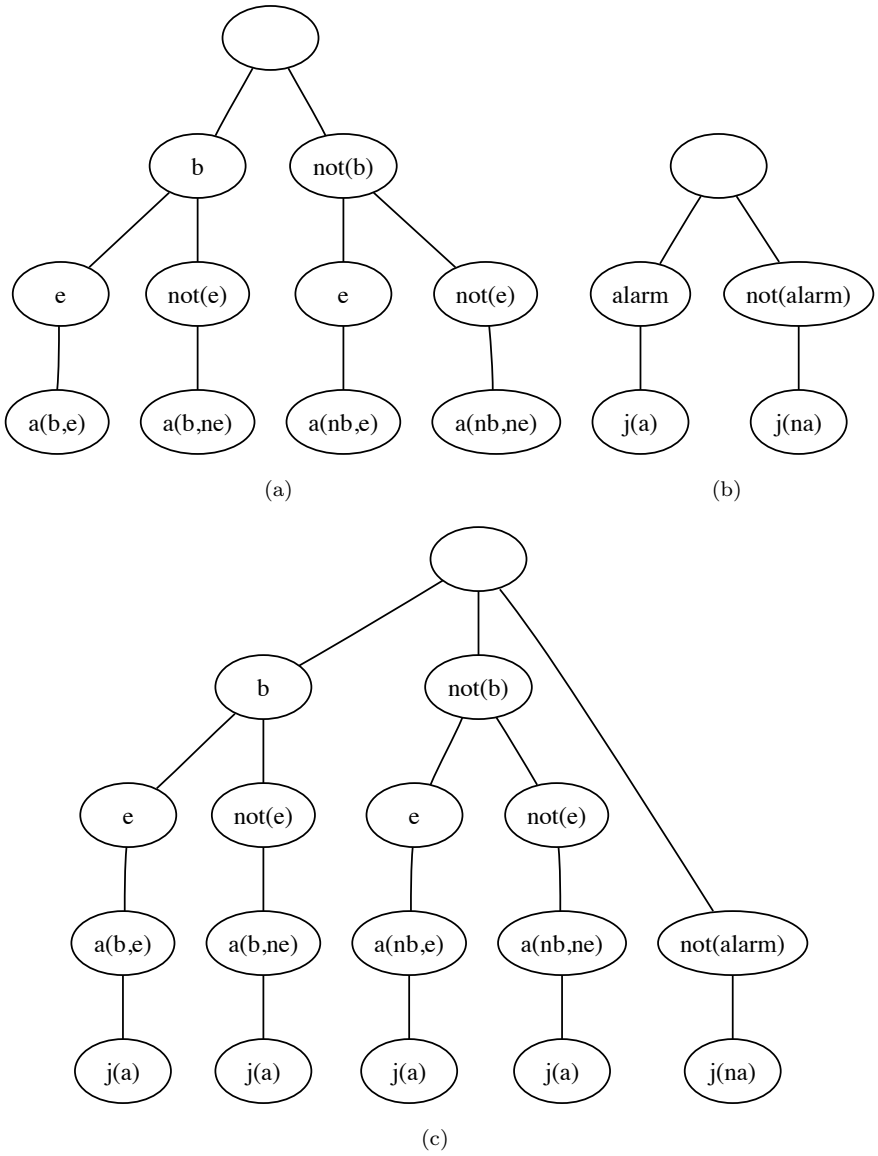



Figure 4.6: Nested tries for Example 4.13.

**Algorithm 4.9** Recursive node merging: generating a sequence of BDD definitions from a trie. REPLACE$(T, C, n_i)$ replaces each occurrence of $C$ in $T$ by $n_i$.

```
1: function RecursiveNodeMerging(trie T, index i)
2:   if ¬leaf(T) then
3:     S_∧ := {(C, P) | leaf C is the only child of P in T}
4:     for all (C, P) ∈ S_∧ do
5:       write n_i = P ∧ C
6:       T := replace(T, (C, P), n_i)
7:       i := i + 1
8:     S_∨ := {[C_1, ..., C_n] | leaves C_j are all the children of P in T, n > 1}
9:     for all [C_1, ..., C_n] ∈ S_∨ do
10:      write n_i = C_1 ∨ ... ∨ C_n
11:      T := replace(T, [C_1, ..., C_n], n_i)
12:      i := i + 1
13:    RecursiveNodeMerging(T, i)
```

### 4.3.4 From DNFs to BDDs

As outlined in Section 4.1.1, to efficiently compute the probability of a DNF formula representing a set of explanations, our implementation represents this formula as a binary decision diagram (BDD). We use SimpleCUDD [Mantadelis et al., 2008] as a wrapper tool for the state-of-the-art BDD package CUDD[4] to construct and evaluate BDDs. More precisely, the trie representation of the DNF is translated into a sequence of definitions, which is afterwards processed by SimpleCUDD to build the BDD using CUDD primitives. It is executed via Prolog's shell utility, and results are reported via shared files. In [Mantadelis et al., 2010], where this part of ProbLog inference is studied in more detail, these steps are called *preprocessing* and *BDD construction* respectively. Here, we will restrict the presentation to the basic preprocessing method, *recursive node merging*, as already given in [Kimmig et al., 2008, 2010]; for a comparison to alternative approaches and further optimizations we refer to [Mantadelis et al., 2010].

During preprocessing, it is crucial to exploit the structure sharing (prefixes and suffixes) already in the trie representation of a DNF formula, otherwise CUDD computation time becomes extremely long or memory overflows quickly. Since CUDD builds BDDs by joining smaller BDDs using logical operations, recursive node merging traverses the trie bottom-up to successively generate definitions for all its subtrees. Algorithm 4.9 gives the details of this procedure. In each iteration it applies two different operations that reduce the trie by merging nodes. The first operation (*depth reduction*, lines 3-7) creates the conjunction of a leaf node with its parent, provided that the leaf is the only child of the parent. The second operation

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

(*breadth reduction*, lines 8-12) creates the disjunction of all child nodes of a node, provided that these child nodes are all leaves. After writing the definition $n_i$ for a new subtree, the leaves of the trie (and their parents in the case of depth reduction) are scanned, replacing each occurrence of the subtree with a new node labeled $n_i$, thus avoiding repeated definitions. Because of the optimizations in CUDD, the final BDD can have a very different structure than the trie.

**Example 4.14** *The translation for query* `path(a,d)` *in our example graph is illustrated in Figure 4.7; it results in the following sequence of definitions:*

$$n1 = ce \wedge ed$$
$$n2 = cd \vee n1$$
$$n3 = ac \wedge n2$$
$$n4 = bc \wedge n2$$
$$n5 = ab \wedge n4$$
$$n6 = n3 \vee n5$$

The complexity of recursive node merging strongly depends on the amount of prefix sharing in the initial trie. As using the function REPLACE in Algorithm 4.9 is rather expensive due to repeated scanning of the trie, in [Mantadelis et al., 2010] we introduced a new data structure called *depth breadth trie* to store definitions of already processed subtries. Using this optimization, the complexity of recursive node merging is $O(N \cdot M)$, where $N$ and $M$ denote the number of explanations and the number of probabilistic facts or Boolean variables, respectively. In practice, due to sharing in the trie structure, recursive node merging is often significantly less complex. For more details, we refer to [Mantadelis et al., 2010].

To deal with the non-DNF formulae stemming from negation on derived subgoals, cf. Section 4.3.3, the preprocessing method needs to be adapted. We first determine a total order $T_1, \ldots, T_n$ of the tries such that each trie $T_i$ only contains references to other tries $T_j$ with $j < i$. We then run recursive node merging on the tries in this order using a consecutive sequence of references to obtain a single sequence of definitions for the entire formula.

### 4.3.5 Program Sampling

Program sampling is quite different from the approaches discussed before, as the two main steps are **(a)** generating a sample program and **(b)** performing standard refutation on the sample. Thus, instead of combining large numbers of explanations, we need to manipulate large numbers of different programs or samples.

Our first approach was to generate a complete sample and to check for an explanation. In order to accelerate the process, explanations were cached in

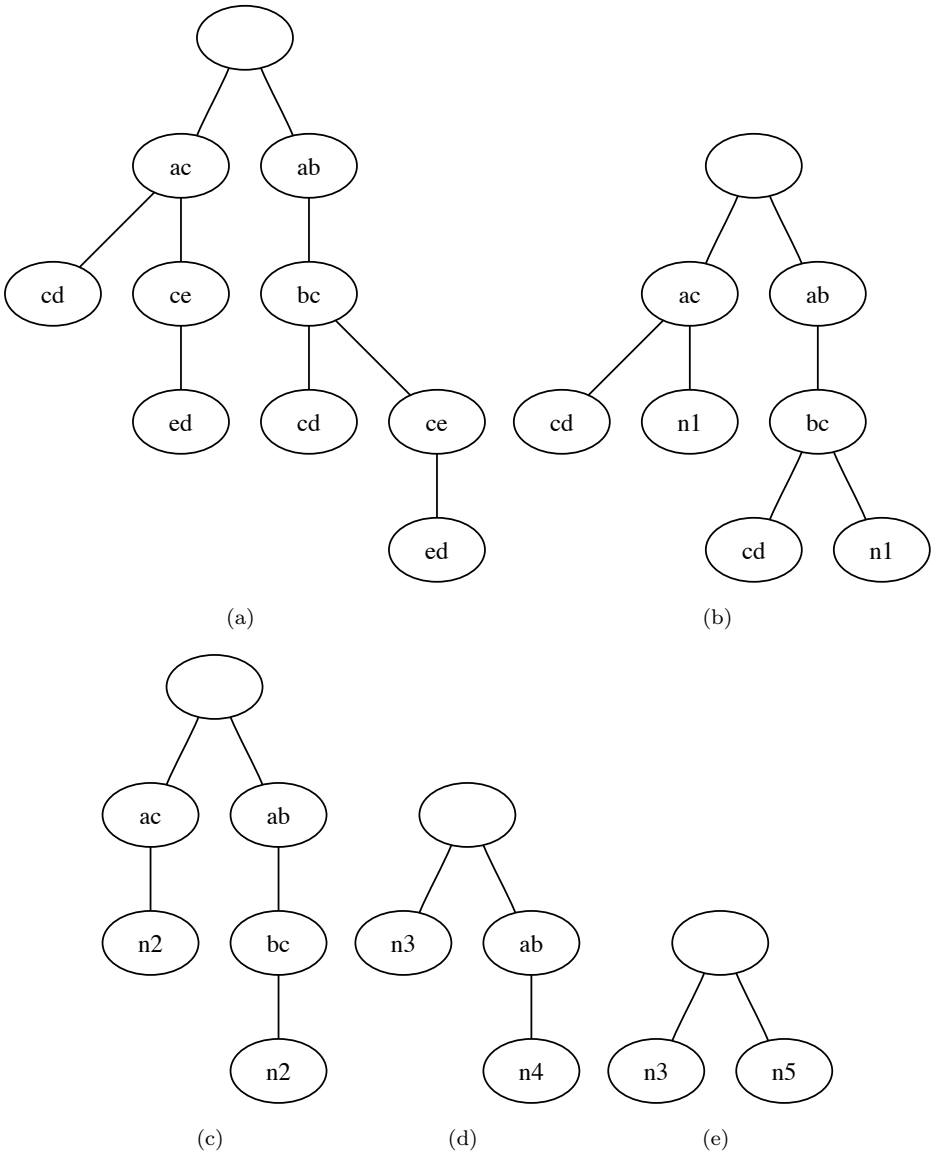



Figure 4.7: Recursive node merging for `path`(`a`, `d`)

a trie to skip inference on a new sample. If no explanations exist on a cache, we call the standard Prolog refutation procedure. Although this approach works rather well for small databases, it does not scale to larger databases where just generating

a new sample requires walking through millions of facts.

We observed that even in large programs explanations are often quite short, i.e., we only need to verify whether facts from a small fragment of the database are in the sample. This suggests that it may be a good idea to take advantage of the independence between facts and generate the sample *lazily*: our implementation of Algorithm 4.7 only verifies whether a fact is in the sample once it is needed in a proof. YAP represents samples compactly as a three-valued array with one field for each fact, where 0 means the fact was not yet sampled, 1 it was already sampled and belongs to the sample, 2 it was already sampled and does not belong to the sample. In this implementation:

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

Note that as fact identifiers are used to access the array, whose size needs to be fixed beforehand, the approach cannot directly be used for non-ground facts, where grounding-specific identifiers are generated on demand. The current implementation of program sampling therefore uses the internal database to store the result of sampling different groundings of such facts.

Program sampling can directly handle negation in ProbLog programs that have a total well-founded model for each interpretation of probabilistic facts (cf. Section 3.3.3) without need for adaptations, as it simply exploits Prolog's mechanisms for SLD-resolution including negation as failure.

In [Kimmig et al., 2009], we showed that using YAP's tabling mechanism can speed up program sampling execution in cases where subgoals have to be proven repeatedly.

### 4.3.6 DNF Sampling

Our implementation of DNF sampling as outlined in Algorithm 4.8 first builds the trie representing the DNF as for exact inference. This determines the order of explanations used to check whether a sampled interpretation-explanation pair $(I, i)$ is positive, that is, whether no $E_j$ with $j < i$ is true in $I$, cf. Equation (4.8). As for program sampling, the sampling phase follows a lazy approach, recording truth values in an array as either true, false, or undetermined. We first sample an explanation $E_i$ from the list of trie leaves. If $i = 1$, the sample is directly accepted,

else, we initialize the current sample by setting corresponding truth values for facts used by $E_i$ and check truth values of explanations $E_{i-1}, \ldots, E_1$ successively, sampling additional facts as needed. As soon as some $E_j$ is determined to be false in the current partial interpretation, we continue with the next explanation $E_{j-1}$. If an explanation $E_j$ with $j < i$ turns out to be true in the current partial interpretation, we know that the sample will not be accepted, and the algorithm terminates the current iteration. If $E_1$ is reached and determined to be false in the current interpretation, the sample is positive.

## 4.4 Experiments

We performed experiments with our implementation of ProbLog in the context of the biological network obtained from the Biomine project [Sevon et al., 2006] to answer the following questions:

**Q1** How do the approximation methods compare in terms of probability estimates and in terms of runtime?

**Q2** Does ProbLog inference scale to large databases?

We used two subgraphs extracted around genes known to be connected to the Alzheimer disease as well as the full network. Details of the datasets are given in Appendix A. For a first comparison of our algorithms on a small scale network where success probabilities can be calculated exactly, we used SMALL, a graph with around 150 edges. Scalability was evaluated using both MEDIUM (roughly 12,000 edges) and the entire BIOMINE network with roughly 1,000,000 nodes and 6,000,000 edges. In all experiments, we queried for the probability that two of the gene nodes with HGNC ids 620, 582, 983 are connected, that is, we used queries such as `path('HGNC_983','HGNC_620',Path)`. We used the following definition of an acyclic path in our background knowledge:

```
path(X,Y,A)   :- path(X,Y,[X],A).
path(X,X,A,A).
path(X,Y,A,R) :- X \== Y,
                 edge(X,Z),
                 absent(Z,A),
                 path(Z,Y,[Z|A],R).
```
(4.14)

As list operations to check for the absence of a node get expensive for long paths, we consider an alternative definition for use in program sampling. It provides cheaper testing by using the internal database of YAP to store nodes on the current

path under key `visited`, but cannot be used by inference methods relying on backtracking to find multiple proofs:

```
memopath(X,Y,A)    :- eraseall(visited),
                      memopath(X,Y,[X],A).
memopath(X,X,A,A).
memopath(X,Y,A,R)  :- X \== Y,                                   (4.15)
                      edge(X,Z),
                      recordzifnot(visited,Z,_),
                      memopath(Z,Y,[Z|A],R).
```

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

```
lenpath(N,X,Y,Path) :- lenpath(N,X,Y,[X],Path).
lenpath(N,X,X,A,A)  :- N >= 0.
lenpath(N,X,Y,A,P)  :- X\ == Y,
                       N > 0,                                    (4.16)
                       edge(X,Z),
                       absent(Z,A),
                       NN is N - 1,
                       lenpath(NN,Z,Y,[Z|A],P).
```

All experiments were performed on a Core 2 Duo 2.4 GHz 4 GB machine running Linux. All times reported are in `msec` and do not include the time to load the graph into Prolog. The latter takes 20, 200 and 78140 `msec` for SMALL, MEDIUM and BIOMINE respectively. Furthermore, as YAP indexes the database at query time, we query for the explanation probability of `path('HGNC_620','HGNC_582',Path)` before starting runtime measurements. This takes 0, 50 and 25900 `msec` for SMALL, MEDIUM and BIOMINE respectively. We report $T_P$, the time spent by ProbLog to search for explanations, as well as $T_B$, the time spent to execute BDD scripts (whenever meaningful). We also report the estimated probability $P$. For approximate inference using bounds, we report exact intervals for $P$, and also include the number $n$ of BDDs constructed. We set both the initial threshold and the shrinking factor to $0.5$. We computed $k$-probability for $k = 1, 2, \ldots, 1024$.[5] In the bounding algorithms, the error interval ranged between 10% and 1%. For the Monte Carlo methods, we also report the number $i$ of iterations, where each iteration generates $m = 1000$ samples.

**Small Sized Sample** We first compared our algorithms on SMALL. Table 4.1 shows the results for $k$-probability and exact inference. Note that nodes 620

[5]Note that $k = 1$ corresponds to the explanation probability.

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

Table 4.1: $k$-probability on SMALL.

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

Table 4.2: Inference using bounded approximation on SMALL; rows correspond to $\delta = 0.1, 0.05, 0.01$.

and 582 are close to each other, whereas node 983 is farther apart. Therefore, connections involving the latter are less likely. In this graph, we obtained good approximations using a small fraction of explanations (the queries have 13136, 155695 and 16048 explanations respectively). Our results also show a significant increase in runtimes as ProbLog explores more paths in the graph, both within the Prolog code and within the BDD code. The BDD runtimes can vary widely, we may actually have large runtimes for smaller BDDs, depending on BDD structure and the amount of variable reordering performed. However, using SimpleCUDD instead of the C++ interface used in [Kimmig et al., 2008] typically decreases BDD time by one or two orders of magnitude.

Table 4.2 gives corresponding results for bounded approximation. The algorithm converges quickly, as few explanations are needed and BDDs remain small. Note however that exact inference is competitive for this problem size. Moreover, we observe large speedups compared to the implementation with meta-interpreters used in [De Raedt et al., 2007b], where total runtimes to reach $\delta = 0.01$ for these queries were 46234, 206400 and 307966 `msec` respectively. Table 4.3 shows the performance of the program sampling estimator. On SMALL, program sampling is

| path | $983-620$ | | | $983-582$ | | | $620-582$ | | |
|---|---|---|---|---|---|---|---|---|---|
| $\delta$ | $i$ | $T_P$ | $P$ | $i$ | $T_P$ | $P$ | $i$ | $T_P$ | $P$ |
| 0.10 | 1 | 10 | 0.11 | 1 | 10 | 0.11 | 1 | 30 | 1.00 |
| 0.05 | 1 | 10 | 0.11 | 1 | 10 | 0.10 | 1 | 20 | 1.00 |
| 0.01 | 16 | 130 | 0.11 | 16 | 170 | 0.11 | 1 | 30 | 1.00 |

Table 4.3: Program sampling inference on SMALL.

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

Table 4.4: $k$-probability on MEDIUM.

the fastest approach. Already within the first 1000 samples a good approximation is obtained. When using the default stopping criterion for DNF sampling, roughly the same number of samples are generated, but runtimes are at least one order of magnitude higher as the full DNF needs to be constructed first. Given the DNF, generating 1000 samples takes about 770, 1900 and 230 `msec` for the three queries respectively. Furthermore, for larger $\delta$, probability estimates can be far off the true value (for instance 0.39 instead of 1.0), as 1000 samples are not sufficient to explore DNFs of this size.

As a first answer to question **Q1**, we note that, with the exception of DNF sampling, the approximation algorithms quickly produce close approximations on SMALL, with program sampling being fastest and exact inference being competitive among methods using BDDs. The first set of experiments thus confirms that the implementation on top of YAP-Prolog enables efficient probabilistic inference on small sized graphs.

**Medium Sized Sample** For graph MEDIUM with around 11000 edges, exact inference is no longer feasible. Table 4.4 again shows results for the $k$-probability. Comparing these results with the corresponding values from Table 4.1, we observe

| memo | $983-620$ | | | $983-582$ | | | $620-582$ | | |
|---|---|---|---|---|---|---|---|---|---|
| $\delta$ | $i$ | $T_P$ | $P$ | $i$ | $T_P$ | $P$ | $i$ | $T_P$ | $P$ |
| 0.10 | 1 | 1180 | 0.78 | 1 | 2130 | 0.76 | 1 | 1640 | 1.00 |
| 0.05 | 2 | 2320 | 0.77 | 2 | 4230 | 0.74 | 1 | 1640 | 1.00 |
| 0.01 | 29 | 33220 | 0.77 | 29 | 61140 | 0.77 | 1 | 1670 | 1.00 |

Table 4.5: Program sampling inference using `memopath/3` on MEDIUM.

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 runtimes. Approximate inference using bounds only reached loose bounds (with differences $> 0.2$) on queries involving node `'HGNC_983'`, as upper bound formulae with more than 10 million conjunctions were encountered, which could not be processed.

The program sampling estimator using the standard definition of `path/3` on MEDIUM did not complete the first 1000 samples within one hour. A detailed analysis shows that this is caused by some queries backtracking too heavily. Table 4.5 therefore reports results using the memorising version `memopath/3`. With this improved definition, program sampling performs well: it obtains a good approximation in a few seconds. Requiring tighter bounds however can increase runtimes significantly. DNF sampling on MEDIUM did not complete the resolution phase within 30 minutes.

Concerning **Q1**, this second set of experiments thus confirms that program sampling is fastest and most accurate among the approximation methods, though some programming effort is required to adapt the background knowledge for the larger database. As a first answer to **Q2**, we note that while inference techniques relying on the full DNF cannot be applied on MEDIUM, other approximation methods do scale to medium sized graphs.

**Biomine Database** The Biomine Database covers hundreds of thousands of entities and millions of links. On BIOMINE, we therefore restricted our experiments to the approximations given by $k$-probability and program sampling. Given the results on MEDIUM, we directly used `memopath/3` for program sampling. Tables 4.6 and 4.7 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 program sampling explode quickly even with the improved path definition.

Given that sampling a program that does not entail the query is extremely unlikely for the setting considered so far, we performed an additional experiment on BIOMINE,

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

Table 4.6: $k$-probability on BIOMINE; rows correspond to $k = 1, 2, 4, \ldots, 1024$.

| memo | $983-620$ | | | $983-582$ | | | $620-582$ | | |
|---|---|---|---|---|---|---|---|---|---|
| $\delta$ | $i$ | $T_P$ | $P$ | $i$ | $T_P$ | $P$ | $i$ | $T_P$ | $P$ |
| 0.10 | 1 | 100,700 | 1.00 | 1 | 1,656,660 | 1.00 | 1 | 1,696,420 | 1.00 |
| 0.05 | 1 | 100,230 | 1.00 | 1 | 1,671,880 | 1.00 | 1 | 1,690,830 | 1.00 |
| 0.01 | 1 | 93,120 | 1.00 | 1 | 1,710,200 | 1.00 | 1 | 1,637,320 | 1.00 |

Table 4.7: Program sampling inference using `memopath/3` on BIOMINE.

where we restrict the number of edges on the path connecting two nodes to a maximum of 2 or 3. Results are reported in Table 4.8. As none of the resulting queries have more than 50 explanations, exact inference is much faster than program sampling, which needs a higher number of samples to reliably estimate probabilities that are not close to 1. Table 4.9 shows the results of the same experiment for DNF sampling. While runtimes are significantly shorter than for program sampling, they clearly exceed those for exact inference, as for this problem size, building BDDs is very fast and the sampling phase thus causes large overhead.

Concerning question **Q2** on scalability, this last set of experiments shows that the $k$-probability and especially program sampling perform well for queries in large sized graphs. Furthermore, it illustrates that performance does not solely depend on the size of the database, but also on the set of explanations for a specific query.

We briefly summarize the experiments with respect to the different inference methods now.

- The *explanation probability* can efficiently be calculated even for large databases.

| len | $983-620$ | | | $983-582$ | | | $620-582$ | | |
|---|---|---|---|---|---|---|---|---|---|
| $\delta$ | $i$ | $T$ | $P$ | $i$ | $T$ | $P$ | $i$ | $T$ | $P$ |
| 0.10 | 1 | 21,400 | 0.04 | 1 | 18,720 | 0.11 | 1 | 19,150 | 0.58 |
| 0.05 | 1 | 19,770 | 0.05 | 1 | 20,980 | 0.10 | 2 | 35,100 | 0.55 |
| 0.01 | 6 | 112,740 | 0.04 | 16 | 307,520 | 0.11 | 40 | 764,700 | 0.55 |
| exact | - | 477 | 0.04 | - | 456 | 0.11 | - | 581 | 0.55 |
| 0.10 | 1 | 106,730 | 0.14 | 1 | 105,350 | 0.33 | 1 | 45,400 | 0.96 |
| 0.05 | 1 | 107,920 | 0.14 | 2 | 198,930 | 0.34 | 1 | 49,950 | 0.96 |
| 0.01 | 19 | 2,065,030 | 0.14 | 37 | 3,828,520 | 0.35 | 6 | 282,400 | 0.96 |
| exact | - | 9,413 | 0.14 | - | 9,485 | 0.35 | - | 15,806 | 0.96 |

Table 4.8: Program sampling inference for different values of $\delta$ and exact inference using `lenpath/4` with length at most 2 (top) or 3 (bottom) on BIOMINE. For exact inference, runtimes include both Prolog and BDD time.

| len | $983-620$ | | | $983-582$ | | | $620-582$ | | |
|---|---|---|---|---|---|---|---|---|---|
| $\delta$ | $i$ | $T$ | $P$ | $i$ | $T$ | $P$ | $i$ | $T$ | $P$ |
| 0.10 | 1 | 10,993 | 0.04 | 1 | 11,112 | 0.11 | 1 | 14,985 | 0.55 |
| 0.05 | 1 | 10,990 | 0.04 | 1 | 11,087 | 0.11 | 2 | 21,158 | 0.55 |
| 0.01 | 7 | 74,530 | 0.04 | 17 | 182,214 | 0.11 | 40 | 412,799 | 0.55 |
| 0.10 | 1 | 24,994 | 0.14 | 1 | 24,995 | 0.35 | 1 | 34,551 | 0.90 |
| 0.05 | 1 | 25,026 | 0.16 | 2 | 35,602 | 0.36 | 1 | 33,112 | 1.04 |
| 0.01 | 19 | 216,212 | 0.14 | 37 | 408,374 | 0.36 | 5 | 76,098 | 0.97 |

Table 4.9: DNF sampling inference for different values of $\delta$ using `lenpath/4` with length at most 2 (top) or 3 (bottom) on BIOMINE.

- The performance of *exact inference for the success probability* depends on the DNF; it is competitive on smaller databases, but also for queries with explanations from a restricted region of large databases.
- *Bounded Approximation* suffers from large DNFs for upper bounds. While it was the first practical inference technique for ProbLog, improvements of the implementation have made exact inference competitive since.
- The *k-probability* scales to large databases, but higher values of $k$ need to be considered with increasing query complexity, and the quality of approximation cannot directly be determined.
- *Program sampling* consistently provides the best performance in terms of approximation quality and runtime, though some programming effort can be needed to adapt the background knowledge for larger databases.
- *DNF sampling* is not competitive for the type of problem considered in these experiments; however, it might still be a valuable replacement of BDDs in the

context of tabled ProbLog, where the bottleneck of exact inference is often shifted from SLD-resolution to BDD construction [Mantadelis and Janssens, 2010].

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

## 4.5 Related Work

As discussed in Section 3.4, ProbLog is closely related to other probabilistic languages such as ICL, pD and PRISM, which are all based on Sato's distribution semantics. However, ProbLog is – to the best of the author's knowledge – the first implementation that tightly integrates Sato's original distribution semantics in a state-of-the-art Prolog system without making additional restrictions (such as the exclusive explanation assumption made in PHA and PRISM). Furthermore, ProbLog is the first probabilistic logic programming system using BDDs as a basic data structure for probability calculation, a principle that receives increased interest in the fields of probabilistic logic learning and probabilistic databases, cf. for instance [Riguzzi, 2007; Ishihata et al., 2008; Olteanu and Huang, 2008; Thon et al., 2008; Riguzzi, 2009].

As ProbLog, implementations of closely related languages typically use a two-step approach to inference, where explanations are collected in the first phase, and probabilities are calculated once all explanations are known.

Poole's ICL implementation AILog2 is a meta-interpreter implemented in SWI-Prolog for didactical purposes, where the disjoint-sum-problem is tackled using a symbolic disjoining technique [Poole, 2000]. Inspired on ProbLog, Riguzzi [2009] introduces an inference algorithm and its implementation for modularly acyclic ICL theories combining SLDNF-resolution and BDDs.

PRISM, built on top of B-Prolog, requires programs to be written such that alternative explanations for queries are mutually exclusive. PRISM uses a meta-interpreter to collect explanations in a hierarchical data structure called explanation graph. As explanations are mutually exclusive, the explanation graph directly mirrors the sum-of-products structure of probability calculation [Sato and Kameya, 2001]. These explanation graphs are also used to calculate most likely proofs using a Viterbi algorithm. To model failing derivations for parameter learning, the system uses a technique called First Order Compiler (FOC) to eliminate negation from PRISM programs. One of the keys to PRISM's efficiency is the use of tabling during

the first phase of exact inference. Mantadelis and Janssens [2009, 2010] introduce tabling for exact inference in ProbLog, replacing the single trie holding the DNF by a nested trie structure with single tries corresponding to tabled subgoals. The nested tries discussed for negated subgoals in Section 4.3.3 are a special case of their nested tries restricting subtries to negated goals.

Based on early work on ProbLog, Riguzzi [2007] introduces corresponding inference techniques for LPADs and CP-Logic, again using BDDs for probability calculation, but with a different encoding of probabilistic choices.

Fuhr's pD implementation HySpirit [Fuhr, 2000] uses a magic sets method for modularly stratified Datalog to construct so-called event expressions, that is, Boolean combinations of event keys corresponding to basic probabilistic events. The second step then uses the inclusion-exclusion principle, cf. Equation (4.1), to calculate the probability of such an expression, which is reported to scale to about ten conjuncts.

## 4.6 Conclusions

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

Although we focused on connectivity queries and Biomine here, similar problems are found across many domains; we believe that the techniques presented apply to a wide variety of queries and databases because ProbLog provides a clean separation between background knowledge and what is specific to the engine. As shown for program sampling inference, such an interface can be very useful to improve performance as it allows incremental refinement of background knowledge.

Thanks to the efficient implementation presented here, ProbLog can also serve as a vehicle for lifting traditional ILP approaches to probabilistic languages, and for developing new learning and mining algorithms and tools, as we will elaborate on a number of examples in Parts II and III of this thesis.

# Conclusions Part I

In this first part of the thesis, we have introduced ProbLog, a simple extension of the logic programming language Prolog with probabilistic facts representing independent random variables. Although ProbLog has originally been motivated by the need to combine probabilistic graphs with the deductive power of logic programming, it is a general probabilistic programming language and by no means restricted to the network setting.

We started in Chapter 3 by defining the language ProbLog and its semantics, which is an instance of Sato's distribution semantics. We then formalized the core of probabilistic inference in ProbLog, which encodes the set of explanations of a query as a DNF formula. After introducing additional language concepts to simplify modeling in ProbLog, we completed this first chapter with a discussion of ProbLog's relationship to probabilistic logic programs, the Independent Choice Logic, PRISM, probabilistic Datalog, and CP-Logic.

Chapter 4 then provided details on both exact and approximate inference algorithms for ProbLog. The key to exact inference in ProbLog is the use of binary decision diagrams to tackle the disjoint-sum-problem, that is, the problem of multiple explanations covering the same possible world. We introduced approximations based on either using restricted sets of explanations or sampling techniques and discussed our implementation of ProbLog combining YAP-Prolog with the BDD package CUDD. Finally, we reported on experiments in the context of the Biomine network, demonstrating that ProbLog can be used to query a probabilistic database with several millions of facts.

The ProbLog language and system as presented here form the core of an increasing body of work, ranging from technical improvements such as the inclusion of tabling [Mantadelis and Janssens, 2010] to extensions of the language with decision-theory [Van den Broeck et al., 2010] or continuous random variables [Gutmann et al., 2010a]. An important line of future work therefore lies in further developing the system to facilitate such extensions. From the perspective of probabilistic programming, MCMC-methods as used by other probabilistic programming languages as well as specialized algorithms for situations not requiring the full

power of BDD-based inference (such as grammars, where the disjoint-sum-problem does not occur) are promising directions for future work.

# Intermezzo

# Beyond probabilities

# Chapter 5

# $\omega$ProbLog*

In propositional logic, formulae in disjunctive normal form (DNF) are well suited for satisfiability checking, while binary decision diagrams (BDDs) [Bryant, 1986] facilitate model counting. Associating weights, such as for instance probabilities, costs or utilities, to propositional variables and defining weights of partial interpretations in terms of these basic weights makes it possible to use the same representations and algorithms to answer a variety of queries about models of logical formulae, ranging from finding partial models with minimal accumulated cost or maximal probability to identifying the models where the maximal cost associated to any positive literal is minimal or calculating the probability of the underlying formula being true. In ProbLog as introduced in Part I of this thesis, a logic program where some facts are labeled with probabilities is used to generate a DNF corresponding to all proofs or explanations of a query, and this DNF or an equivalent BDD is used to calculate explanation and success probabilities. While probabilities cover many interesting domains, for instance networks of uncertain relationships or probabilistic grammars, other types of weights are more natural in other domains. For instance, imagine a road network with edge labels denoting distance or expected travel time. In such a network, one might want to query for the shortest travel route between two given locations or the shortest round trip through a set of locations. To determine the best location for some new facility, one might query for the location for which the sum of distances to a number of given locations is smallest. Using weighted networks as an abstract representation of relationships with positive or negative value, a subgraph with maximum value connecting a given set of nodes could be extracted to reduce the size of the network for ease of inspection. In natural language processing, weights can be used to find the most likely derivation or to construct the derivation

*This chapter presents unpublished work. The author thanks Guy Van den Broeck for valuable comments.

forest of a sentence with respect to a probabilistic grammar [Goodman, 1999]. Similarly, constructing ProbLog's DNF representation of sets of explanations or the corresponding BDD (for a given variable order) can be seen as a task in a labeled program, where labels are propositional variables. Also in the context of ProbLog, using vectors of probabilities instead of single probabilities allows one to compare success probabilities for several sets of probability labels within a single query.

In this chapter, we introduce ωProbLog, a generalization of ProbLog to commutative weight semirings, where both proof-based and model-based weights can be calculated. The latter type of weights requires dealing with implicit information in the form of truth values of variables not appearing in an explanation, and thus often involves tackling the disjoint-sum-problem. We identify the semiring characteristics that require dealing with the disjoint-sum-problem for model-based weights and introduce corresponding inference algorithms for ωProbLog that generalize the ones of ProbLog. The Dyna system of Eisner et al. [2005] uses a similar approach of adding semiring weights to definite clause programs. However, given its origin in natural language processing, this system calculates proof- or derivation-based weights only and does not deal with model-based weights that require addressing the disjoint-sum-problem, such as ProbLog's success probability.

The chapter is organized as follows: based on a review of ProbLog inference in Section 5.1, we introduce ωProbLog and develop corresponding inference algorithms in Section 5.2. After discussing related work in Section 5.3, we conclude in Section 5.4.

## 5.1 A Generalized View on ProbLog Inference

Inference in ProbLog addresses questions about the proofs or explanations of a query $q$ in program $T$, or about the complete interpretations or possible worlds where the query is true. In the light of the distribution over interpretations defined by a ProbLog program, questions about proofs can be thought of as ignoring the probabilities of facts used neither positively nor negatively in the proof, while those are taken into account when querying about interpretations. While ProbLog inference algorithms for the explanation probability $P_x^T(q)$ (the probability of the most likely explanation of $q$), the sum of probabilities $S_x^T(q)$ and the success probability $P_s^T(q)$ (the probability of $q$ being true in a random interpretation) have been introduced in Chapter 4, in this section, we present a slightly different view on these algorithms to highlight their commonalities as well as their differences. We additionally consider the MAP (maximum a posteriori) probability $P_{MAP}^T(q)$, that is, the probability of the most likely complete interpretation or possible world in which the query is true, another type of query considered frequently in probabilistic models. The definitions of these values illustrate their underlying common structure, using either maximization or summation to aggregate over

explanations or interpretations, respectively:

$$P_x^T(q) = \max_{E \in Expl^T(q)} \left( \prod_{f_i \in E^1} p_i \prod_{f_i \in E^0} (1 - p_i) \right) \tag{5.1}$$

$$S_x^T(q) = \sum_{E \in Expl^T(q)} \left( \prod_{f_i \in E^1} p_i \prod_{f_i \in E^0} (1 - p_i) \right) \tag{5.2}$$

$$P_{MAP}^T(q) = \max_{I \models_T q} \left( \prod_{f_i \in I^1} p_i \prod_{f_i \in I^0} (1 - p_i) \right) \tag{5.3}$$

$$P_s^T(q) = \sum_{I \models_T q} \left( \prod_{f_i \in I^1} p_i \prod_{f_i \in I^0} (1 - p_i) \right) \tag{5.4}$$

While the success probability is defined in terms of the set of complete interpretations supporting the query, represented by the DNF formula

$$D_s^T(q) = \bigvee_{I \models_T q} \left( \bigwedge_{f_i \in I^1} b_i \wedge \bigwedge_{f_i \in I^0} \neg b_i \right), \tag{5.5}$$

it is calculated in ProbLog based on the set of all explanations of the query, represented as the logically equivalent[1] DNF formula

$$D_x^T(q) = \bigvee_{E \in Expl^T(q)} \left( \bigwedge_{f_i \in E^1} b_i \wedge \bigwedge_{f_i \in E^0} \neg b_i \right). \tag{5.6}$$

This *explanation DNF* can be obtained by backtracking over Algorithm 4.2 to enumerate explanations, where logically equivalent explanations are filtered out.

All four definitions as well as evaluating the truth value of the underlying DNF formulae can be seen as calculations in semirings, an observation that underlies the more general definitions and algorithms provided in Section 5.2. These semirings are listed in Table 5.1 together with a number of characteristics of the corresponding definitions; the latter will be discussed in more detail in Section 5.2. Let us now discuss algorithms to compute (5.1)–(5.4) based on $D_x^T(q)$, where the aim is to illustrate common principles, not to provide optimized algorithms.

The definition of the explanation probability in Equation (5.1) exactly mirrors the structure of $D_x^T(q)$, where negation is replaced by inverting the probability,

[1] We refer to Section 3.2 for more details on the relationship of these encodings.

| | Semiring and complement | | | | | | | $\oplus$ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Query | $\Omega$ | $\oplus$ | $\otimes$ | $\omega^0$ | $\omega^1$ | $\overline{\omega}$ | Int. | idp. | cp. | Alg. |
| $D_x(q)$ | $\{0,1\}$ | $\vee$ | $\wedge$ | 0 | 1 | $\neg\omega$ | p | yes | yes | 5.2 |
| $D_s(q)$ | $\{0,1\}$ | $\vee$ | $\wedge$ | 0 | 1 | $\neg\omega$ | c | yes | yes | 5.2 |
| $P_x(q)$ | $\mathbb{R}_{\geq 0}$ | max | $\prod$ | 0 | 1 | $1-\omega$ | p | yes | no | 5.2 |
| $S_x(q)$ | $\mathbb{R}_{\geq 0}$ | $\sum$ | $\prod$ | 0 | 1 | $1-\omega$ | p | no | yes | 5.2 |
| $P_{MAP}(q)$ | $\mathbb{R}_{\geq 0}$ | max | $\prod$ | 0 | 1 | $1-\omega$ | c | yes | no | 5.3 |
| $P_s(q)$ | $\mathbb{R}_{\geq 0}$ | $\sum$ | $\prod$ | 0 | 1 | $1-\omega$ | c | no | yes | 5.4 |

Table 5.1: Semirings for evaluating truth values as well as probabilistic queries (over p(artial) or c(omplete) interpretations), properties of semiring addition (idempotence and complementarity, cf. Equation (5.13)), and corresponding ωProbLog algorithm.

conjunction by multiplication, and disjunction by maximization. Consequently, it can be calculated by one pass over the DNF, calculating the probability of the current conjunction and updating the maximum seen so far if needed. The sum of probabilities in (5.2) can be calculated with the same algorithm using addition instead of maximization. To calculate explanation probabilities in ProbLog, we implemented an optimized algorithm that avoids explicit DNF construction and uses best first search based on probabilities of partial derivations, cf. Algorithm 4.4. Note that as summation is not idempotent, explicit DNF construction could not be avoided when calculating $S_x^T(q)$, as repeated occurrences of explanations need to be filtered out.

The MAP probability of Equation (5.3) and the success probability of Equation (5.4) perform aggregation over all complete interpretations as given by $D_s^T(q)$, which is impractical for all but the tiniest programs. However, the explanation DNF $D_x^T(q)$ can easily be extended into the equivalent DNF

$$D_{xs}^T(q) = \bigvee_{E \in Expl^T(q)} \left( \bigvee_{I \in Compl^T(E)} \left( \bigwedge_{f_i \in I^1} b_i \wedge \bigwedge_{f_i \in I^0} \neg b_i \right) \right), \tag{5.7}$$

where each explanation $E$ is replaced by the disjunction over its set of completions $Compl^T(E)$, cf. Equation (3.10) on page 34. In contrast to the logically equivalent $D_s^T(q)$, $D_{xs}^T(q)$ may contain the same interpretation multiple times, as interpretations can extend multiple explanations, which is known as the disjoint-sum-problem, cf. Section 4.1.1. The algorithms for calculating $P_{MAP}$ and $P_s$ exploit this idea, but without explicitly constructing the fully expanded formula.

Using the algorithm sketched for the explanation probability, $P_{MAP}^T(q)$ could directly be calculated on either $D_s^T(q)$ or $D_{xs}^T(q)$. However, the grouping of interpretations in $D_{xs}^T(q)$ can be exploited to instead base this calculation on $D_x^T(q)$ by maximizing over the maximum of each group. As multiplication distributes over

**Algorithm 5.1** ProbLog MAP inference: calculating the probability of the most likely interpretation satisfying a DNF (restricted to variables of DNF literals $l_{ij}$).

```
1:  function MAP(DNF D_x^T(q) = ⋁_{i=1}^{n} ⋀_{j=1}^{m_i} l_ij)
2:      maxp := 0; var = ∅
3:      for i = 1, ..., n do
4:          p_i := 1; var_i = ∅
5:          for j = 1, ..., m_i do
6:              p_i := p_i · p(l_ij)
7:              var_i := var_i ∪ {var(l_ij)}
8:          p_i := p_i · ∏_{f_k ∈ var \ var_i} max(p_k, 1 − p_k)
9:          maxp := maxp · ∏_{f_k ∈ var_i \ var} max(p_k, 1 − p_k)
10:         maxp := max(maxp, p_i)
11:         var := var ∪ var_i
12:     return maxp
```

maximization, the most likely interpretation extending a given explanation $E$ is the one that sets each variable not occurring in the explanation to its most likely truth value, and its probability is thus given by:

$$\max_{I \in Compl^T(E)} P(I) = P(E) \cdot \prod_{f_i \in E^?} \max(p_i, (1 - p_i)) \tag{5.8}$$

The basic algorithm thus considers one explanation at a time, comparing the probability of the most likely interpretation extending it to the probability of the most likely interpretation extending any of the previous explanations, using Equation (5.8) to perform the maximization corresponding to the inner disjunction of Equation (5.7). As maximization is idempotent, repeated occurrences of the same interpretation as extensions of different explanations do not influence the result. A variant of this algorithm is detailed in Algorithm 5.1. It incorporates one simple optimization: to avoid setting a potentially large number of variables to their more likely truth value, it maintains the set of variables that appeared in the DNF as processed so far and only performs maximization with respect to interpretations of this set. This is possible as each unseen variable will be set to its more likely value in the most likely interpretation extending such a partial interpretation. Lines 8 and 9 therefore update the probabilities to be compared in line 10 to take into account the same set of variables. Note that this does only take into account variables occurring in the DNF; if there are other variables, the same type of correction has to be applied to the result for each of them.

While the definition of the success probability in Equation (5.4) only differs from that of the MAP probability in using summation instead of maximization, Algorithm 5.1 cannot directly be adapted for its calculation, as summation is not idempotent and repeated occurrences of the same interpretation due to different

explanations need to be filtered out. This cannot be done by simply maintaining the set of variables seen, but would require one to disjoin each new explanation from the entire formula processed so far. As this quickly becomes impractical, ProbLog instead uses binary decision diagrams (BDDs, [Bryant, 1986], cf. Section 2.4) to address the disjoint-sum-problem. Basically, a BDD repeatedly splits a DNF according to the truth value of a single variable $n$, leading to two new DNFs $h$ and $l$ (corresponding to high and low children in the BDD), exploiting the following recursive equation to calculate probabilities:

$$P((n \wedge h) \vee (\neg n \wedge l)) = p_n \cdot P(h) + (1 - p_n) \cdot P(l) \tag{5.9}$$

For more details, we refer to the discussion of Algorithm 4.5 in Section 4.1.1. *Reduced* BDDs as used in ProbLog omit nodes whose low and high children are isomorphic, meaning that those nodes are also left out from the calculation. Such nodes would require one to calculate $p_n \cdot P(h) + (1-p_n) \cdot P(h)$, which by distributivity equals $(p_n + (1 - p_n)) \cdot P(h) = 1 \cdot P(h) = P(h)$ and thus can safely be dropped.

## 5.2 From ProbLog to ωProbLog

The inference tasks for ProbLog discussed in the previous section are all rooted in the same framework that assigns probabilities to (partial) interpretations by multiplying probabilities of basic facts. However, similar questions could be asked in different contexts, where basic facts are labeled with weights that are not probabilities, or where labels are combined differently to obtain weights of partial interpretations. For instance, the number of satisfying interpretations could be determined in this way, or one could ask for proofs or interpretations of lowest cost if costs are associated to positive and negative literals. This general idea is formalized in ωProbLog using commutative semirings. ωProbLog is related to DTProbLog [Van den Broeck et al., 2010] in that it extends ProbLog beyond the use of probability labels. However, while DTProbLog uses different types of labelled facts to integrate probabilistic information and decision theory, ωProbLog uses a single type of general weight labels and also tackles different types of problems.

A *semiring* is a structure $(\Omega, \oplus, \otimes, \omega^0, \omega^1)$ where *addition* $\oplus$ and *multiplication* $\otimes$ are binary operations over the set $\Omega$ such that both operators are associative, $\oplus$ is commutative, $\otimes$ distributes over $\oplus$, $\omega^0 \in \Omega$ is the neutral element with respect to $\oplus$, $\omega^1 \in \Omega$ that of $\otimes$, and for all $\omega \in \Omega$, $\omega^0 \otimes \omega = \omega \otimes \omega^0 = \omega^0$. In a *commutative semiring*, $\otimes$ is commutative as well.

A *ωProbLog program* $T$ consists of a commutative semiring $(\Omega, \oplus, \otimes, \omega^0, \omega^1)$, a finite set of *labeled ground facts* $\{\omega_1 :: f_1, \ldots, \omega_n :: f_n\}$ with $\omega_i \in \Omega$, a set of *background knowledge clauses* as in ProbLog, cf. Equation (3.1) (p. 31), and a complement function that assigns weight $\overline{\omega}$ to the negation of a labeled fact with weight $\omega$. The

weight of a (partial) interpretation $J$ is defined as the multiplication of the weights of its literals:

$$W^T(J) = \bigotimes_{f_i \in J^1} \omega_i \bigotimes_{f_i \in J^0} \overline{\omega}_i \tag{5.10}$$

Each ωProbLog program $T$ defines two different weights of a query $q$, the *explanation weight*

$$W_x^T(q) = \bigoplus_{E \in Expl^T(q)} \left( \bigotimes_{f_i \in E^1} \omega_i \bigotimes_{f_i \in E^0} \overline{\omega}_i \right), \tag{5.11}$$

and the *interpretation weight*

$$W_s^T(q) = \bigoplus_{I \models_T q} \left( \bigotimes_{f_i \in I^1} \omega_i \bigotimes_{f_i \in I^0} \overline{\omega}_i \right). \tag{5.12}$$

As both multiplication and addition are commutative and associative in a commutative semiring, the following property holds for all ωProbLog programs.

**Property 5.1** *The explanation and interpretation weight are independent of the order of both literals and conjunctions.*

An addition operator $\oplus$ is *idempotent* if $\omega \oplus \omega = \omega$ for all $\omega \in \Omega$. We will call an addition operator $\oplus$ *complementary* (w.r.t. a given complement function) if the following (one of the complement axioms of Boolean algebras) holds for all $\omega \in \Omega$:

$$\omega \oplus \overline{\omega} = \omega^1 \tag{5.13}$$

**Property 5.2** *For any commutative ring $(\Omega, \oplus, \otimes, \omega^0, \omega^1)$ with additive inverse $-\omega$, addition $\oplus$ is complementary for the complement function $\overline{\omega} = -\omega \oplus \omega^1$.*

While the ring setting is covered by ωProbLog, combining a semiring with an arbitrary complement function is more flexible.

**Property 5.3** *The weight of the set $Compl(E)$ of all completions of an explanation $E$ can be calculated as*

$$\bigoplus_{I \in Compl^T(E)} W^T(I) = W^T(E) \otimes \bigotimes_{f_i \in E^?} (\omega_i \oplus \overline{\omega}_i).$$

This is due to distributivity of multiplication over addition in semirings.

**Property 5.4** *If semiring addition $\oplus$ is complementary, the weight of a given explanation $E$ equals the sum of the weights of all its completions, that is*

$$W^T(E) = \bigoplus_{I \in Compl^T(E)} W^T(I)$$

This is a direct consequence of Property 5.3.

**Property 5.5** *If the addition operator $\oplus$ is idempotent and complementary, the explanation weight and the interpretation weight coincide.*

Given Property 5.4, $W_x^T(q)$ can be rewritten as

$$\bigoplus_{E \in Expl^T(q)} W^T(E) = \bigoplus_{E \in Expl^T(q)} \bigoplus_{I \in Compl^T(E)} W^T(I),$$

which, given commutativity and associativity of $\oplus$, equals $W_s^T(q)$ for idempotent $\oplus$. An instance of such a semiring is that of truth values as given for both $D_x^T(q)$ and $D_s^T(q)$ in Table 5.1.

**Example 5.1** *Consider the following weighted variant of Example 3.1*

$$1 :: \texttt{edge(a,c)}. \quad -2 :: \texttt{edge(a,b)}. \quad 2 :: \texttt{edge(c,e)}.$$
$$3 :: \texttt{edge(b,c)}. \quad 3 :: \texttt{edge(c,d)}. \quad 1 :: \texttt{edge(e,d)}.$$

*together with background knowledge clauses defining a spanning tree predicate* `st/0`*, semiring* $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$ *and complement function* $\overline{\omega} = 0$*. The set of all spanning trees is given by*

$$\begin{aligned} D_x(\texttt{st}) = {} & (ab \wedge ac \wedge cd \wedge ce) \vee (ab \wedge ac \wedge ce \wedge ed) \vee (ab \wedge ac \wedge cd \wedge ed) \\ & \vee (ac \wedge bc \wedge cd \wedge ce) \vee (ac \wedge bc \wedge ce \wedge ed) \vee (ac \wedge bc \wedge cd \wedge ed) \\ & \vee (ab \wedge bc \wedge cd \wedge ce) \vee (ab \wedge bc \wedge ce \wedge ed) \vee (ab \wedge bc \wedge cd \wedge ed) \end{aligned}$$

$W_x(\texttt{st})$ *corresponds to the weight of the minimum spanning tree,* $W_s(\texttt{st})$ *to the minimum weight of any subgraph connecting all nodes.*

As for ProbLog, inference in ωProbLog first constructs the explanation DNF $D_x^T(q)$ as defined in Equation (5.6), which is then used in the second phase to calculate the weight of interest. In fact, ProbLog inference can be seen as inference in ωProbLog using the semirings and complement functions summarized in Table 5.1. The third

**Algorithm 5.2** Calculating the explanation weight $W_x^T(q)$ for query $q$ and ωProbLog program $T$ given the explanation DNF $D_x^T(q)$.

```
1: function ExplanationWeight(DNF D_x^T(q) = ⋁_{i=1}^{n} ⋀_{j=1}^{m_i} l_ij)
2:     weight := ω^0
3:     for i = 1, ..., n do
4:         ω_i := ω^1
5:         for j = 1, ..., m_i do
6:             ω_i := ω_i ⊗ ω(l_ij)
7:         weight := weight ⊕ ω_i
8:     return weight
```

and fourth column of this table list characteristics that determine the choice of inference algorithm for the given type of query, while the last column refers to the corresponding algorithm used in the second phase of ωProbLog inference. These algorithms generalize the ones sketched for ProbLog above; we will now discuss them in turn.

Algorithm 5.2 formalizes the calculation of $W_x^T(q)$, generalizing the DNF algorithm for the explanation probability. It simply iterates over all conjunctions, multiplying literal weights for each conjunction and summing the results.

**Example 5.2** *In Example 5.1, the weight of the first explanation $ab \wedge ac \wedge cd \wedge ce$ is $-2+1+3+2=4$, that of $ab \wedge ac \wedge ce \wedge ed$ is $-2+1+2+1=2$. The intermediate weight after seeing two explanations thus is 2, which will be confirmed to be the overall minimum when processing the rest of the DNF.*

Algorithm 5.3 generalizes Algorithm 5.1 for MAP inference in ProbLog to calculate the interpretation weight $W_s^T(q)$ on $D_x^T(q)$ for idempotent semiring addition $\oplus$. The weight updates in lines 8 and 9 exploit Property 5.4, analogously to Equation (5.8). Again, the result needs to be corrected in the same way to take into account variables not appearing in the DNF.

**Example 5.3** *Consider again Example 5.2, but this time to calculate $W_s(\texttt{st})$. To do so, one could explicitly expand $D_x(\texttt{st})$ to $D_{xs}(\texttt{st})$ as in the following fragment restricted to the first two explanations:*

$$\begin{aligned} D_{xs}(\texttt{st}) = {} & (ab \wedge ac \wedge cd \wedge ce \wedge \neg bc \wedge \neg ed) \vee (ab \wedge ac \wedge cd \wedge ce \wedge \neg bc \wedge ed) \\ & \vee (ab \wedge ac \wedge cd \wedge ce \wedge bc \wedge \neg ed) \vee (ab \wedge ac \wedge cd \wedge ce \wedge bc \wedge ed) \\ & \vee (ab \wedge ac \wedge ce \wedge ed \wedge \neg bc \wedge \neg cd) \vee (ab \wedge ac \wedge ce \wedge ed \wedge \neg bc \wedge cd) \\ & \vee (ab \wedge ac \wedge ce \wedge ed \wedge bc \wedge \neg cd) \vee (ab \wedge ac \wedge ce \wedge ed \wedge bc \wedge cd) \vee \ldots \end{aligned}$$

**Algorithm 5.3** Calculating the interpretation weight $W_s^T(q)$ for query $q$ and ωProbLog program $T$ with idempotent addition operator $\oplus$ given the explanation DNF $D_x^T(q)$ (restricted to variables of DNF literals $l_{ij}$).

**function** InterpretationWeight(DNF $D_x^T(q) = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} l_{ij}$)
  $weight := \omega^0; var = \emptyset$
  **for** $i = 1, \ldots, n$ **do**
    $\omega_i := \omega^1; var_i = \emptyset$
    **for** $j = 1, \ldots, m_i$ **do**
      $\omega_i := \omega_i \otimes \omega(l_{ij})$
      $var_i := var_i \cup \{var(l_{ij})\}$
    $\omega_i := \omega_i \bigotimes_{f_k \in var \setminus var_i} (\omega_k \oplus \overline{\omega}_k)$
    $weight := weight \bigotimes_{f_k \in var_i \setminus var} (\omega_k \oplus \overline{\omega}_k)$
    $weight := weight \oplus \omega_i$
    $var := var \cup var_i$
  return $weight$

**Algorithm 5.4** Calculating the interpretation weight $W_s^T(q)$ for query $q$ and ωProbLog program $T$ with complementary addition operator $\oplus$ given a BDD representation of the explanation DNF $D_x^T(q)$.

**function** InterpretationWeightBdd(BDD node $n$)
  **if** $n$ is the 1-terminal **then**
    return $\omega^1$
  **if** $n$ is the 0-terminal **then**
    return $\omega^0$
  let $h$ and $l$ be the high and low children of $n$
  $weight(h)$ := InterpretationWeightBdd($h$)
  $weight(l)$ := InterpretationWeightBdd($l$)
  return $(\omega_n \otimes weight(h)) \oplus (\overline{\omega}_n \otimes weight(l))$

*Algorithm 5.3 instead updates the weights of these explanations by additionally considering ed and cd, respectively, leading to values* $4 + \min(1, 0) = 4$ *and* $2 + \min(3, 0) = 2$ *and thus to an intermediate result of 2 based on variable set* $\{ab, ac, cd, ce, ed\}$*. bc will only be taken into account when processing the fourth explanation.*

If semiring addition is not idempotent, calculating $W_s^T(q)$ based on $D_x^T(q)$ involves solving the disjoint-sum-problem and is therefore done on the BDD encoding of $D_x^T(q)$. The algorithm as given in Algorithm 5.4 generalizes Algorithm 4.5 for ProbLog's success probability.

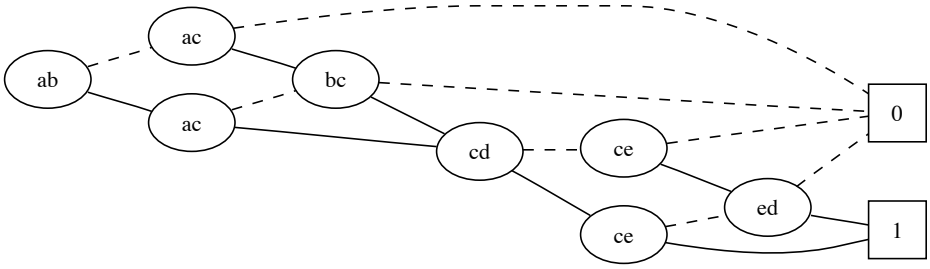



Figure 5.1: BDD used to calculate $W_s(\texttt{st})$ in Examples 5.4 and 5.5.

**Property 5.6** *If semiring addition is complementary, the interpretation weight $W_s^T(q)$ can be calculated on the BDD representation of $D_x^T(q)$.*

Calculating the weight on the BDD instead of on the disjunction of all interpretations making the query true exploits associativity and commutativity of both semiring operations (as variables need to be brought in the same order on all paths through the BDD, and paths are sorted according to variable values) as well as distributivity (to obtain a tree-shaped representation of the formula). Merging redundant subtrees does not affect the calculation, as all ingoing edges are maintained. However, dropping redundant nodes is only safe for complementary addition operators, as only in this case the omitted update $(\omega_n \otimes W(h)) \oplus (\overline{\omega}_n \otimes W(h))$ equals $W(h)$.

**Example 5.4** *If we set all fact weights $\omega$ as well as their complements $\overline{\omega}$ to* 0.5 *in the program of Example 5.1 and use semiring $(\mathbb{R}_{\geq 0}, +, \cdot, 0, 1)$, $W_s(\texttt{st})$ is the fraction of interpretations making the query true. Figure 5.1 shows the corresponding BDD, on which Algorithm 5.4 will calculate the interpretation weight. Note that this corresponds to calculating the success probability in a ProbLog program where all facts have probability* 0.5*.*

Based on Property 5.3, Algorithm 5.5 generalizes Algorithm 5.4 to non-complementary addition operators. As in Algorithm 5.3, lines 9 and 10 extend the results for the two subtrees to take into account the same set of variables before they are combined in line 11. Again, the same type of correction is required for variables not appearing in the BDD. The algorithm could be used instead of Algorithm 5.3, but for idempotent $\oplus$, it introduces the unnecessary extra cost of building the BDD.

**Example 5.5** *Reconsider Example 5.4, but with semiring $(\mathbb{N}_0, +, \cdot, 0, 1)$ and with all fact weights $\omega$ as well as their complements $\overline{\omega}$ being* 1*. In this case, $W_s(\texttt{st})$*

**Algorithm 5.5** Calculating the interpretation weight $W_s^T(q)$ for query $q$ and ωProbLog program $T$ given a BDD representation of the explanation DNF $D_x^T(q)$ (restricted to BDD variables).

**function** InterpretationWeightBddGeneral(BDD node $n$)
  **if** $n$ is the 1-terminal **then**
    return $(\omega^1, \emptyset)$
  **if** $n$ is the 0-terminal **then**
    return $(\omega^0, \emptyset)$
  let $h$ and $l$ be the high and low children of $n$
  $(weight(h), var(h)) :=$ InterpretationWeightBddGeneral($h$)
  $(weight(l), var(l)) :=$ InterpretationWeightBddGeneral($l$)
  $extended(h) := weight(h) \otimes \bigotimes_{f_i \in var(l) \setminus var(h)} (\omega_i \oplus \overline{\omega}_i)$
  $extended(l) := weight(l) \otimes \bigotimes_{f_i \in var(h) \setminus var(l)} (\omega_i \oplus \overline{\omega}_i)$
  $weight(n) := (\omega_n \otimes extended(h)) \oplus (\overline{\omega}_n \otimes extended(l))$
  return $(weight(n), \{n\} \cup var(h) \cup var(l))$

| | Semiring, labels and complement | | | | | | | | ⊕ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Ex. | $\Omega$ | $\oplus$ | $\otimes$ | $\omega^0$ | $\omega^1$ | $\omega \in$ | $\overline{\omega}$ | Int. | idp. | cp. | Alg. |
| 5.1 | $\mathbb{R} \cup \{\infty\}$ | min | $+$ | $\infty$ | 0 | $\mathbb{R}$ | 0 | p | yes | no | 5.2 |
| 5.1 | $\mathbb{R} \cup \{\infty\}$ | min | $+$ | $\infty$ | 0 | $\mathbb{R}$ | 0 | c | yes | no | 5.3 |
| 5.4 | $\mathbb{R}_{\geq 0}$ | $+$ | $\cdot$ | 0 | 1 | $\{0.5\}$ | 0.5 | c | no | yes | 5.4 |
| 5.5 | $\mathbb{N}$ | $+$ | $\cdot$ | 0 | 1 | $\{1\}$ | 1 | c | no | no | 5.5 |

Table 5.2: Semirings used in Examples 5.1–5.5.

*corresponds to the number of different connected subgraphs and thus performs model counting. The BDD is the same as in the previous example; however, as addition is not complementary here, Algorithm 5.5 needs to be used to calculate* $W_s(\texttt{st})$. *For instance, at the lower node labeled ce, we need to take into account that its high child has not considered ed, while the low child has. The weight* 1 *of the high child thus needs to be extended to* $1 \cdot (1 + 1) = 2$, *resulting in weight* $1 \cdot 2 + 1 \cdot 1 = 3$ *based on variables* $\{ce, ed\}$ *for the current node.*

Table 5.2 summarizes the example instances of ωProbLog used in Examples 5.1–5.5 in analogy to Table 5.1. We conclude this section with some examples of ωProbLog programs with more complex weight domains.

**Example 5.6** *Consider the path predicate and the graph of Example 3.1 with the following labels*

$(0.8, 1) :: \texttt{edge(a, c)}.$ $(0.7, 1) :: \texttt{edge(a, b)}.$ $(0.8, 1) :: \texttt{edge(c, e)}.$
$(0.6, 1) :: \texttt{edge(b, c)}.$ $(0.9, 1) :: \texttt{edge(c, d)}.$ $(0.5, 1) :: \texttt{edge(e, d)}.$

*semiring* $(\mathbb{R}_{\geq 0} \times \mathbb{N}, \oplus_a, \otimes_a, (0,0), (1,1))$ *and complement function* $\overline{(p,c)} = (1-p, c)$*, where the binary operators are defined as follows:*

$$(p_1, c_1) \oplus_a (p_2, c_2) = (p_1 + p_2, c_1 + c_2)$$

$$(p_1, c_1) \otimes_a (p_2, c_2) = (p_1 \cdot p_2, c_1 \cdot c_2)$$

*In this case, query weights simultaneously sum probabilities and count the number of explanations or models. The result can thus be used to calculate average probabilities in an additional step. Such averages cannot be calculated directly, as they correspond to n-ary operations that cannot be broken down to use an associative binary addition operator. Note that while* $W_s^T(q)$ *can be calculated using Algorithm 5.5 in this example, this would not be possible if interpretation weights were defined as sum, as the resulting multiplication* $(p_1, c_1) \otimes'_a (p_2, c_2) = (p_1 + p_2, c_1 \cdot c_2)$ *would no longer distribute over summation* $\oplus_a$*, thus violating the semiring conditions. However, one could still use* $D_s^T(q)$ *to calculate the value.*

**Example 5.7** *We extend the ProbLog path example to explicitly include the set of partial interpretations leading to the explanation or MAP probability:*

$(0.8, \{\{\texttt{ac}\}\}) :: \texttt{edge}(\texttt{a},\texttt{c}).$ $(0.7, \{\{\texttt{ab}\}\}) :: \texttt{edge}(\texttt{a},\texttt{b}).$
$(0.8, \{\{\texttt{ce}\}\}) :: \texttt{edge}(\texttt{c},\texttt{e}).$ $(0.6, \{\{\texttt{bc}\}\}) :: \texttt{edge}(\texttt{b},\texttt{c}).$
$(0.9, \{\{\texttt{cd}\}\}) :: \texttt{edge}(\texttt{c},\texttt{d}).$ $(0.5, \{\{\texttt{ed}\}\}) :: \texttt{edge}(\texttt{e},\texttt{d}).$

*The second argument of the weight is the set of interpretations where the fact is true, which consists of the single partial interpretation containing a unique propositional variable. We denote the set of literals using the variables occurring in fact labels by* $\mathcal{L}$*. The complement function is defined as* $\overline{(p, \{\{v\}\})} = (1-p, \{\{\neg v\}\})$*. The binary operators of the semiring* $(\mathbb{R}_{\geq 0} \times 2^{2^{\mathcal{L}}}, \oplus_{MAP}, \otimes_{MAP}, (0, \emptyset), (1, \{\emptyset\}))$ *are defined as follows:*

$$(p_1, S_1) \otimes_{MAP} (p_2, S_2) = (p_1 \cdot p_2, \{i_1 \cup i_2 \mid i_1 \in S_1, i_2 \in S_2\}) \tag{5.14}$$

$$(p_1, S_1) \oplus_{MAP} (p_2, S_2) = \begin{cases} (p_1, S_1) & \text{if } p_1 > p_2 \\ (p_2, S_2) & \text{if } p_1 < p_2 \\ (p_1, S_1 \cup S_2) & \text{if } p_1 = p_2 \end{cases} \tag{5.15}$$

*On the first argument denoting the probability, the operators thus correspond to those of calculating the explanation or MAP probability, while the second argument maintains corresponding sets of interpretations. Note that multiplication is only well-defined if the resulting interpretations are consistent; however, this is the case when evaluating on DNF formulae.*

*Given* $D_x(\texttt{path}(\texttt{a},\texttt{c})) = ac \vee (ab \wedge bc)$*, we obtain* $W_x(\texttt{path}(\texttt{a},\texttt{c})) = (0.8, \{\{ac\}\})$ *as* $W(ac) = (0.8, \{\{ac\}\})$*,* $W(ab \wedge bc) = (0.7 \cdot 0.6, \{\{ab, bc\}\})$*, and* $0.8 >$

0.42. *When evaluating* $W_x(\texttt{path}(\texttt{a},\texttt{c}))$*, both explanations are extended into* $(0.336, \{\{ab, ac, bc\}\})$*, which is the most likely interpretation of relevant facts supporting the query. Taking remaining facts into account as well, we obtain* $(0.12096, \{\{ab, ac, bc, cd, ce, ed\}, \{ab, ac, bc, cd, ce, \neg ed\}\})$*, as there are two equally likely interpretations with MAP probability.*

**Example 5.8** *Consider the semiring* $(\mathcal{BDD}(\mathcal{V}), \vee, \wedge, 0, 1)$ *with complement function* $\overline{\omega} = \neg\ \omega$*, where* $\mathcal{BDD}(\mathcal{V})$ *is the set of reduced ordered BDDs over variables* $\mathcal{V} = \{ab, ac, bc, cd, ce, ed\}$ *(in this order),* 0 *and* 1 *are BDD terminal nodes and the operators the usual BDD operators. We label facts with BDDs corresponding to logical variables:*

$\texttt{ac} :: \texttt{edge}(\texttt{a},\texttt{c}).$ $\quad \texttt{ab} :: \texttt{edge}(\texttt{a},\texttt{b}).$ $\quad \texttt{ce} :: \texttt{edge}(\texttt{c},\texttt{e}).$
$\texttt{bc} :: \texttt{edge}(\texttt{b},\texttt{c}).$ $\quad \texttt{cd} :: \texttt{edge}(\texttt{c},\texttt{d}).$ $\quad \texttt{ed} :: \texttt{edge}(\texttt{e},\texttt{d}).$

*Using background knowledge defining a spanning tree predicate* `st/0` *as in Example 5.1,* $W_x^T(\texttt{st})$ *is the BDD shown in Figure 5.1.*

**Example 5.9** *Labelling basic facts with polynomials, e.g.*

$\texttt{x} :: \texttt{edge}(\texttt{a},\texttt{c}).$ $\quad (\texttt{x}-3) :: \texttt{edge}(\texttt{a},\texttt{b}).$ $\quad (-\texttt{x}+1) :: \texttt{edge}(\texttt{c},\texttt{e}).$
$2\texttt{x} :: \texttt{edge}(\texttt{b},\texttt{c}).$ $\quad \texttt{x}^2 :: \texttt{edge}(\texttt{c},\texttt{d}).$ $\quad -\texttt{x}^3 :: \texttt{edge}(\texttt{e},\texttt{d}).$

*and using the corresponding ring of polynomials* $(\mathcal{P}(x), +, \cdot, 0, 1)$ *and complement function* $\overline{\omega} = -\omega$*, we can use ωProbLog to construct polynomials describing the query weights in terms of the input parameter* $x$*. For instance,* $W_x^T(\texttt{path}(\texttt{a},\texttt{c})) = x + (x-3)\cdot 2x = 2x^2 - 5x$*.*

## 5.3 Related Work

As ωProbLog, the *weighted logic programs* of Eisner and Blatz [2007] associate weights to basic facts and use a logic program to derive weights of other atoms. However, the weight of a derived atom is obtained by aggregating the weights of the bodies of all ground clauses whose head is the atom, where body weights in turn aggregate weights of their atoms using a second type of aggregation operator. Different types of aggregation over clauses can be used within a program, as long as unifiable heads all use the same type. The *semiring-weighted dynamic programs* employed by the Dyna system of Eisner et al. [2005] (called Dyna programs in the following) are a special case of this framework. The Dyna system originates from work in natural language processing and is closely related to the general framework of *semiring parsing* introduced by Goodman [1999]. Dyna programs use "Horn equations", that is, range-restricted definite clauses where conjunction in the

body is replaced by the semiring's multiplication operator $\otimes$ and the consequence operator `:-` is replaced by the same update operator $\oplus =$ in all clauses. Values for axioms (atoms not appearing in clause heads) are asserted as part of the input of a Dyna program. Furthermore, clauses can have side conditions, which essentially are conjunctions of logical atoms without assigned semiring value whose truth values decide on the applicability of a clause. To allow for uniform treatment in practice, these atoms are assigned the semiring's $\omega^0$ or $\omega^1$ element depending on their truth values. In contrast to $\omega$ProbLog, Dyna programs do not use negated atoms in clause bodies, and do not require multiplication to be commutative as the evaluation order is fixed by the program. Apart from this, the key difference between Dyna programs and $\omega$ProbLog programs is that Dyna directly uses the program structure to calculate weights, while $\omega$ProbLog uses the explanation DNF, which corresponds to a reduced form of the program structure. More specifically, the DNF filters repeated occurrences of the same basic literal in a proof as well as repeated occurrences of the same explanation (independent of fact order in the underlying proof), while all these contribute to weights in Dyna programs. At the level of literals in proofs, this is closely related to repeated occurrences of the same probabilistic fact being considered different random events in PRISM [Sato and Kameya, 2001], but a single one in ProbLog, cf. Sections 3.3.2 and 3.4.4. Furthermore, $\omega$ProbLog offers a second type of weight defined in terms of interpretations, which is not possible in Dyna programs unless the program explicitly generates complete interpretations as proofs. As ProbLog's success probability falls into this latter category, it cannot directly be calculated in Dyna. In contrast, PRISM programs directly correspond to Dyna programs calculating the probability of an atom as the sum of the probabilities of all its mutually exclusive derivations.

Dyna's basic inference algorithm is an iterative forward reasoning algorithm that maintains an agenda of atoms to be processed (initially the ones whose value was asserted) and a chart of current values associated to atoms. Changes of values for each atom are propagated along all groundings of clauses having the atom in the body, until no more changes need propagation or a user-defined convergence criterion is met. The use of forward reasoning is motivated by the need to deal with phenomena such as unary rule cycles and $\epsilon$-productions, which are known to be problematic in backward reasoning. However, in principle, the backward reasoning algorithm which constructs the explanation DNF in $\omega$ProbLog could be adapted to this type of weight calculation by dropping any filtering operations and directly performing semiring operations in parallel with logical operations during resolution.

Wachter et al. [2007] have identified ordered BDDs as the most appropriate representation language for local computations in semiring valuation algebras, where the main inference task is to eliminate a set of variables from a semiring product whose factors can have overlapping domains. An example of such a

product is the joint probability in Bayesian networks, where factors correspond to conditional probability distributions. For optimization tasks in semiring valuation algebras, Pouly et al. [2007] follow a similar approach using propositional DAGs, another graphical representation of Boolean functions. While ωProbLog shares the underlying compilation idea of these approaches, it addresses a different inference task and also provides a framework to define concrete problem instances by means of logic programs.

## 5.4 Conclusions

We have introduced ωProbLog, a generalization of ProbLog where basic facts are labeled with weights from an arbitrary commutative semiring and a complement function associates weights to corresponding negated literals. Defining weights of partial interpretations as multiplication of literal weights, two types of weights are defined for derived queries, based on summing over either all explanations of the query, or over all complete interpretations where the query is true. We have generalized ProbLog inference algorithms to calculate both types of weights based on the explanation DNF, using binary decision diagrams to tackle the disjoint-sum-problem when needed.

While ωProbLog is closely related to the semiring-weighted dynamic programs of the Dyna system [Eisner et al., 2005], both approaches cover cases that cannot be represented in the other one. ωProbLog defines two types of weights based on the set of explanations or the set of interpretations, respectively, while Dyna defines a single type of weight based on the set of proofs. This difference implies that Dyna can use semirings with non-idempotent operators to take into account duplicate proof weights, which is not possible in ωProbLog, but cannot calculate ProbLog success probabilities, which is possible in ωProbLog.

The work presented in this chapter opens several directions for future work. While we have provided some first example problems that can be addressed within ωProbLog, a more systematic overview should be attempted. Implementing the basic inference algorithms discussed here and experimenting with various types of semirings will be another step towards fully exploring the potential of the framework. Furthermore, extending the algorithms to work with nested representations as obtained in tabled ProbLog [Mantadelis and Janssens, 2010] and to pruning techniques that avoid building the full explanation DNF needs to be investigated.

# Part II

# BDD-based Learning

# Outline Part II

In the first part of the thesis, we have introduced ProbLog as a tool for modeling and reasoning in probabilistic databases. However, manually inspecting large probabilistic databases or assigning probabilities can be tedious, while providing example queries of interest is often easier. In this part of the thesis, we therefore discuss two different **Machine Learning techniques** targeted at improving ProbLog programs with respect to a set of example queries. Both methods directly exploit the binary decision diagrams generated by ProbLog's inference engine to efficiently evaluate the effect of possible changes, but differ in the type of changes they consider. Both methods will be experimentally evaluated in the context of the Biomine network.

In **Chapter 6**, we study how to condense a large ProbLog database to a subset covering the most relevant information for the current context. We introduce the task of **Theory Compression** as the reduction of a ProbLog database to at most $k$ probabilistic facts – without changing their parameters – based on queries that should or should not have a high success probability, which is a form of theory revision with a single operator that deletes probabilistic facts. We develop a greedy compression algorithm that removes one fact at a time, choosing the one with maximal improvement of the likelihood of examples.

While theory compression is restricted to deleting facts or setting their (known) probabilities to 0, in **Chapter 7**, we consider the task of automatically inferring labels for probabilistic facts. To this aim, we introduce a novel setting for **Parameter Learning** in probabilistic databases. Probabilistic databases such as ProbLog differ from the generative models widely used in parameter learning, as they do not define a distribution over possible example queries, but a distribution over truth values for each query. We therefore consider training examples labeled with their desired probability and develop a gradient descent method to minimize the mean squared error. The approach integrates learning from entailment and learning from proofs, as proofs can conveniently be regarded as conjunctive queries in ProbLog.

# Chapter 6

# Theory Compression*

In this chapter, we introduce the task of compressing a ProbLog theory using a set of positive and negative examples, and develop 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 well 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 motivated by the biological application as outlined in Section 2.5. 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. A biologist may for instance 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. The idea thus 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 [Wrobel et al., 1996] where the only operation allowed is the deletion of facts. The analogy explains why we have formalized the theory

*This chapter presents joint work with Kate Revoredo, Kristian Kersting, Hannu Toivonen and Luc De Raedt published in [De Raedt et al., 2006, 2007a] and, in most detail, in [De Raedt et al., 2008b]. Main contributions of the author are the ProbLog-related part of the implementation as well as parts of the experiments.

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.

We proceed as follows: in Section 6.1, the task of probabilistic theory compression is defined; an algorithm for tackling the compression problem is presented in Section 6.2. Experiments that evaluate the effectiveness of the approach are presented in Section 6.3. Finally, in Sections 6.4 and 6.5, we touch upon related work and conclude.

## 6.1 Compressing ProbLog Theories

Before introducing the ProbLog theory compression problem, it is helpful to consider the corresponding problem in a purely logical setting[1]. Assume that, as in traditional theory revision [Wrobel et al., 1996; Richards and Mooney, 1995], 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 Biomine network, cf. Section 2.5, 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. We are interested in finding a small theory that maximizes the likelihood of a set $X = P \cup N$ of positive and negative examples. Here, small means that the number of probabilistic facts 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 $P^T(x)$ that any given example $x$ is positive. (This is subtly different from specifying the distribution of (positive) examples.) We define the *likelihood of an example* as

$$LL^T(x) = \begin{cases} P^T(x) & \text{if } x \in P \\ 1 - P^T(x) & \text{if } x \in N. \end{cases} \tag{6.1}$$

[1] This can – of course – be modeled within ProbLog by setting all probability labels to 1.

The examples are assumed to be mutually independent, so the total likelihood is obtained as a simple product:

$$LL^T(X) = \prod_{x \in X} LL^T(x) \tag{6.2}$$

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, in order to avoid overfitting, to effectively handle noisy data, and to obtain smaller theories, we slightly redefine $P^T(x)$ in Equation (6.1) to allow misclassifications at a high cost:

$$\hat{P}^T(x) = \max\big(\min[1-\epsilon, P^T(x)], \epsilon\big) \tag{6.3}$$

for some constant $\epsilon > 0$ specified by the user. This avoids the possibility that the likelihood function becomes 0 due to a positive example not covered by the theory at all or a negative example covered with probability 1.

The *ProbLog Theory Compression Problem* can therefore be formalized as follows:

**Task 6.1 (ProbLog Theory Compression)**

**Given**
- *a ProbLog theory* $T = F^T \cup BK$,
- *sets* $P$ *and* $N$ *of positive and negative examples in the form of independent and identically-distributed ground facts, and*
- *a constant* $k \in \mathbb{N}$,

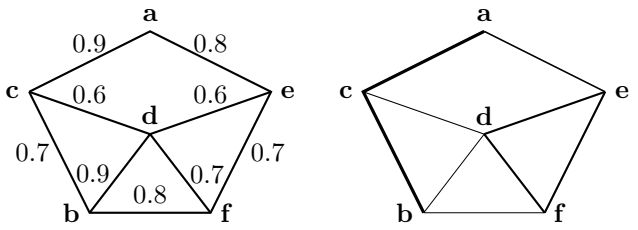
**find** *a theory* $R \subseteq T$ *of size at most* $k$ *(i.e.* $|F^R| \leq k$*) that has a maximum likelihood w.r.t. the examples* $X = P \cup N$*, i.e.,*

$$R = \arg \max_{R \subseteq T \wedge |F^R| \leq k} LL^R(X) \tag{6.4}$$

**Example 6.1** *Within bibliographic data analysis, the similarity structure among items can improve information retrieval results. Consider a collection of papers* $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}\}$ *and some pairwise similarities* `similar(a,c)`*, e.g., based on keyword analysis. Uncertainty in the data can elegantly be represented by the attached probabilities:*

```
0.9 :: similar(a,c).   0.9 :: similar(b,d).   0.6 :: similar(d,e).
0.8 :: similar(a,e).   0.8 :: similar(b,f).   0.7 :: similar(d,f).
0.7 :: similar(b,c).   0.6 :: similar(c,d).   0.7 :: similar(e,f).
```

*The corresponding graph is depicted in Figure 6.1(a). We only list pairs of similar facts in one order and encode symmetry in the background knowledge. Two items*




(a) (b)

| edge | $L^T(E)$ |
|---|---|
| - | 0.014 |
| (b,d) | 0.127 |
| (b,f) | 0.276 |
| (c,d) | 0.377 |
| (a,e) | 0.627 |
| (d,e) | 0.627 |
| (d,f) | 0.627 |
| (e,f) | 0.627 |
| (a,c) | 0.005 |
| (b,c) | 0.005 |

(c)

Figure 6.1: Illustration of Examples 6.1 and 6.2: (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.

`X` *and* `Y` *are* `related(X,Y)` *if they are similar (such as* `a` *and* `c`*) or if* `X` *is similar to some item* `Z` *which is related to* `Y`*:*

```
sim(X,Y)     :- similar(X,Y).
sim(X,Y)     :- similar(Y,X).
related(X,Y) :- sim(X,Y).
related(X,Y) :- sim(X,Z),related(Z,Y).
```

*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.*

## 6.2 The ProbLog Theory Compression Algorithm

The ProbLog compression algorithm removes one fact at a time from the theory, and chooses the fact greedily to be the one whose removal results in the largest likelihood.

**Example 6.2** *Reconsider the* `related` *theory from Example 6.1, Figure 6.1(a), where 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*

**Algorithm 6.1** ProbLog theory compression. LOWBDD is akin to BOUNDS (cf. Algorithm 4.6), but returns the last lower bound BDD.

```
1: function COMPRESS(ProbLog program T = {p_1 :: f_1, ..., p_n :: f_n} ∪ BK,
   Examples E, constants k and ε)
2:   for all e ∈ E do
3:     BDD(e) := LOWBDD(e, T, δ)
4:   R := {p_i :: f_i | b_i occurs in a BDD(e)}
5:   BDD(E) := ⋃_{e∈E} {BDD(e)}
6:   improves := TRUE
7:   while (|R| > k or improves) and R ≠ ∅ do
8:     ll := LIKELIHOOD(R, BDD(E), ε)
9:     f := arg max_{f∈R} LIKELIHOOD(R − {f}, BDD(E), ε)
10:    improves := (ll ≤ LIKELIHOOD(R − {f}, BDD(E), ε))
11:    if improves or |R| > k then
12:      R := R − {f}
13:  return R ∪ BK
```

*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 theory will be compressed to just the two edges* `similar(a,c)` *and* `similar(c,b)`*, which is the smallest theory with maximal likelihood* 0.627*. The sequence of deletions is illustrated in Figures 6.1(b) and 6.1(c).*

The ProbLog compression algorithm as given in Algorithm 6.1 works in two phases. First, it constructs BDDs for explanations of the examples using the standard ProbLog inference engine. These BDDs then play a key role in the second step where facts are greedily removed, as they make it very efficient to test the effect of removing a fact.

More precisely, the algorithm starts by calling ProbLog's inference engine using $e$ as a query. In principle, any BDD-based method could be used. Here, we use bounded approximation, cf. Section 4.2, which computes the BDDs for lower and upper bounds. The compression algorithm only employs the lower bound BDDs, since they are simpler and, hence, more efficient to use. All facts used in at least one explanation occurring in the (lower bound) BDD of some example constitute the set $R$ of possible revision points. All other probabilistic facts do not contribute 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 the theory, rather than the size of the resulting theory, then all these other facts should be left intact.

After the set $R$ of revision points has been determined — and the other facts 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 fact whose deletion results in the best likelihood score, and then deletes it. This process is continued until both $|R| \leq k$ *and* deleting further facts 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 bounded approximation) one can easily evaluate (in the revision phase) conditional probabilities of the form $P^T(q|b'_1 \wedge \cdots \wedge b'_k)$, where the $b'_i$s are possibly negated Booleans representing the truth-values of facts. To compute the answer using the BDD-based probability calculation of Algorithm 4.5, 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 facts, the $b'_i$s will be negative, so one has to set $p'_i = 0$ for all $b'_i$s.

**Example 6.3** *To illustrate fact deletion on a small BDD, we restrict our graph to the nodes* $\{\mathbf{b}, \mathbf{c}, \mathbf{d}\}$*. Figure 6.2 shows the effect of deleting* $\texttt{similar}(\texttt{c}, \texttt{d})$ *on the probability of query* `related(d,b)`*. On the original BDD, the probability is* $P^T(\texttt{related}(\texttt{d}, \texttt{b})) = 0.9 + 0.1 \cdot 0.7 \cdot 0.6 = 0.942$*. After deleting the fact by setting its probability to* 0*, the probability is lowered to* $P^T(\texttt{related}(\texttt{d}, \texttt{b}) | \neg \texttt{similar}(\texttt{c}, \texttt{d})) = 0.9 + 0.1 \cdot 0 \cdot 0.6 = 0.9$*.*

## 6.3 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 facts 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.

### 6.3.1 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.

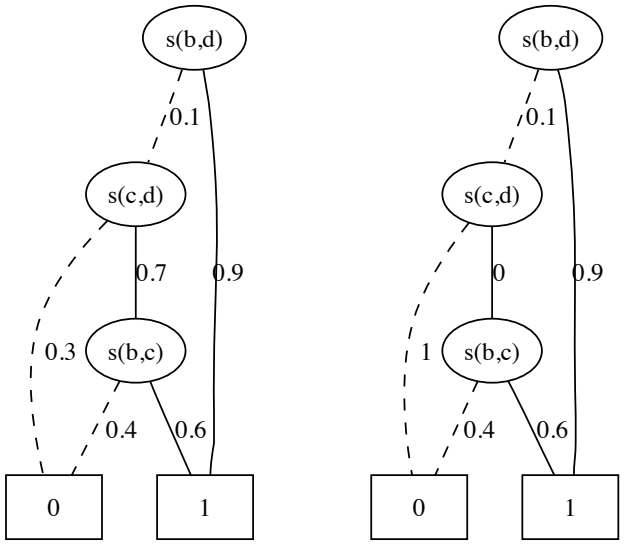



Figure 6.2: BDDs illustrating Example 6.3, (a) before and (b) after deleting `similar(c, d)`.

*Real Data:* To obtain a realistic test setting with natural example queries, we base positive examples on the random Alzheimer genes used to extract Biomine datasets, cf. Appendix A. All our tests use SMALL (144 edges). Three pairs of genes (from HGNC ids 620, 582, 983) are used as positive examples, in the form `path(gene_620, gene_582)` unless otherwise stated. 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.

The larger graph MEDIUM (11530 edges) 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 $\delta = 0.2$.

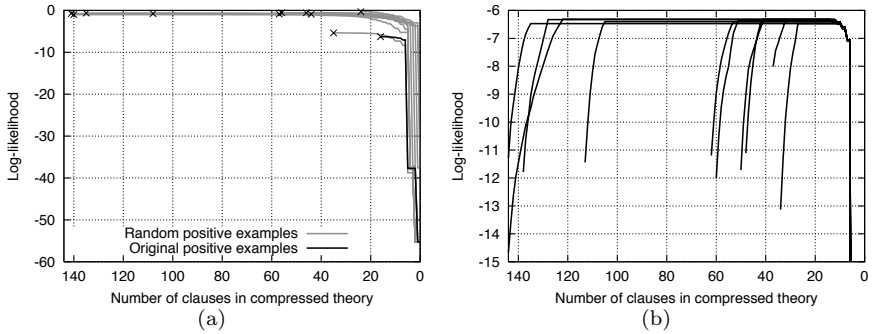



Figure 6.3: Evolvement of log-likelihood for 10 test runs with positive examples only (a) and with both positive and negative examples (b). Different starting points of lines reflect the number of facts in the BDDs used in theory compression.

### 6.3.2 Implementation

We implemented the inference and compression algorithms in Prolog (Yap-5.1.0). The experiments presented in this chapter were performed with the initial implementation of ProbLog, which used a meta-interpreter instead of the source-to-source transformation discussed in Section 4.3.1, stored sets of explanations in trie-like nested terms instead of YAP's tries, and used the C++ interface of CUDD instead of SimpleCUDD. The compiled BDD results were saved for use in the revision phase.

### 6.3.3 Quality of ProbLog Theory Compression

The quality of compressed ProbLog theories is hard to judge using a single objective. We therefore carried out a number of experiments investigating different aspects of quality, summarized by the following set of questions:

**Q1** How does the likelihood evolve during compression?

**Q2** How appropriately are edges of known positive and negative effects handled?

**Q3** How does compression affect unknown test examples?

We will now address these questions in turn.

**Q1: How does the likelihood evolve during compression?** Figure 6.3(a) shows how the log-likelihood evolves during compression in the real data, using positive examples only, where all revision points are eventually removed. In one

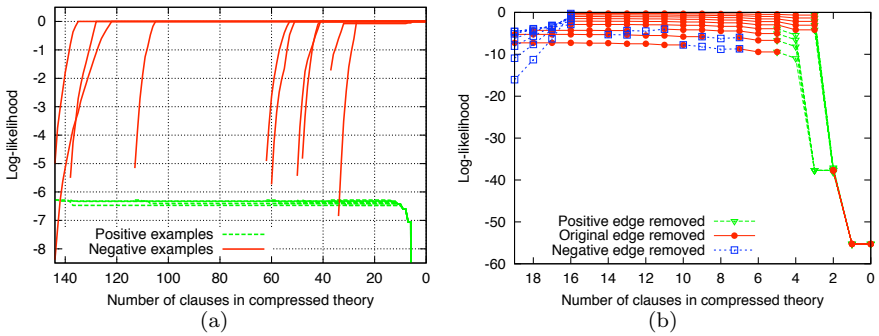



Figure 6.4: Evolvement of log-likelihood for 10 test runs with both positive and negative examples (a). Evolvement of log-likelihood for settings with artificially implanted edges with negative and positive effects (b). In (b), linestyle indicates the type of edge that was removed in the corresponding 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 .

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 facts, compressing to less than 20 facts has only a minor effect on the likelihood. The radical drops in the end occur once 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 6.3(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 6.4(a)) explains this: facts that affect the negative examples mostly are removed first (resulting in an improvement of the likelihood). Only if no other alternatives exist, facts 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 compression process, an issue to be studied shortly in a cross-validation setting.

**Q2: 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 $p$.

We note that different values of $p$ lead to different sets of revision points. This depends on how many explanations are needed to reach the probability interval $\delta$. To obtain comparable results, we computed in a first step the revision points using $p = 0.5$ and interval width 0.2. All facts not appearing as a revision point were excluded from the experiments. On the resulting theory, we then re-ran the compression algorithm for $p = 0.1, 0.2, \ldots, 0.9$ using $\delta = 0.0$, i.e., using exact inference.

Figure 6.4(b) shows the log-likelihood of the examples as a function of the number of facts remaining in the theory during compression; the types of removed edges are coded by linestyle. 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 method is to recognize the artificial edges. Their influence drops together with $p$, but negative edges are always removed before positive ones.

**Q3: 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: facts 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

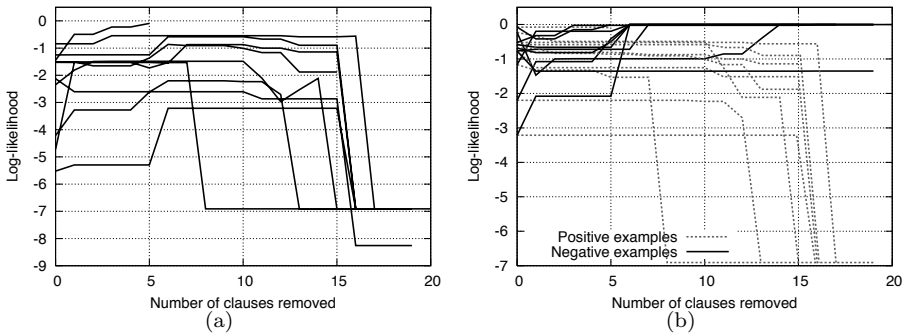



Figure 6.5: Evolvement of log-likelihoods in test sets for 10 random runs: total log-likelihood (a) and log-likelihoods of positive and negative test examples separately (b).

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.

We compressed each of the ProbLog theories based on the training set only; Figure 6.5 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 6.3(b) and 6.4(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, using 3-fold cross-validation for each, is given in Figure 6.6(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

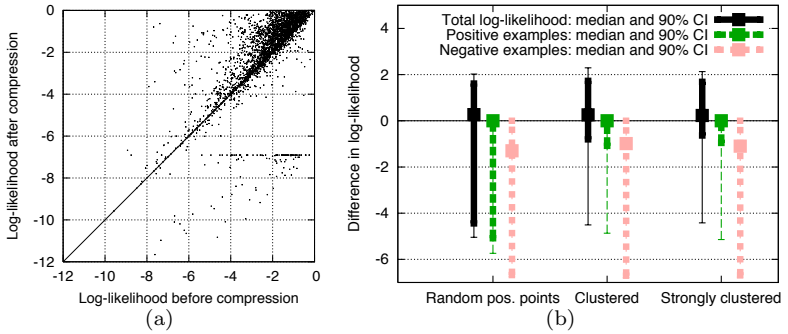



Figure 6.6: Log-likelihoods of the test sets before and after compression (a). Distributions of differences of log-likelihoods in test examples before and after compression, for three different densities of positive nodes (b) (thick line: 90% confidence interval; thin line: 95% confidence interval; for negative test examples, differences in log(1−likelihood) are reported to make results comparable with positive examples).

use of probability $\epsilon = 0.001$ for the example, and probabilities $\leq 0.001$ 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 6.6(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 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,

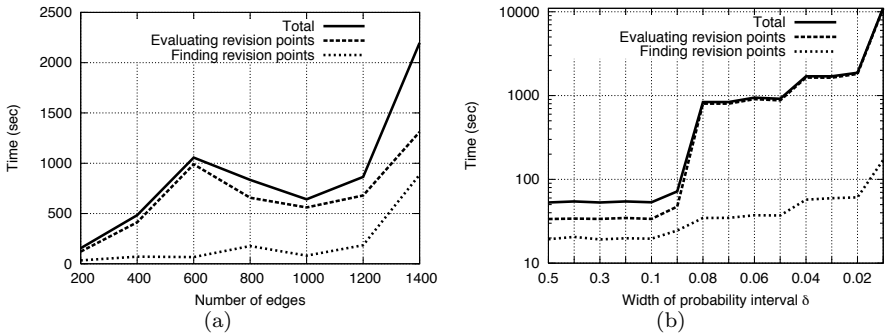



Figure 6.7: Runtime as a function of graph size (number of edges) (a) and as a function of interval width $\delta$ (b). Note that in (b) the y axis is in log-scale and the x axis is discretized in 0.1 steps for [0.5,0.1] and in 0.01 steps in [0.1,0.01].

however, come at the expense of very high runtimes? This question will be investigated in the following subsection.

### 6.3.4 Complexity of ProbLog Theory Compression

We now investigate scalability and runtimes of our compression approach, focusing on the following questions:

**Q4** How do the methods scale up to large graphs?

**Q5** What is the effect of the probability approximation interval $\delta$?

**Q6** What are the crucial technical factors for runtimes?

**Q4: 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 $G_1 \subset G_2 \subset \ldots$ 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.

Runtimes are given in Figure 6.7(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 runtimes 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 explanations and remove the need of deep search for obtaining tight bounds.

**Q5: What is the effect of the probability approximation interval $\delta$?** Smaller values of $\delta$ obviously are more demanding. To study the relationship, we ran the method on the smaller biological graph with the three positive examples using varying $\delta$ values. Figure 6.7(b) shows the total runtime as well as its decomposition to two phases, finding revision points and removing them. The interval width $\delta$ has a major effect on runtime; in particular, the complexity of the revision step is greatly affected by $\delta$.

**Q6: What are the crucial technical factors for runtimes?**

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

Given an interval $\delta$, resources needed for running bounded approximation to obtain the revision points are hard to predict, as those are extremely dependent on the sets of explanations and especially stopped derivations encountered until the stopping criterion is reached. This is not only influenced by the structure of the theory and $\delta$, but also by the presentation of examples. (In one case, reversing the order of nodes in some path atoms decreased runtime 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 fact at a time.

Although in Figure 6.7(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 $\delta$, we have little influence on 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 runtimes (with the cost of less predictable probability intervals). One option would be to use the $k$-probability as done in parameter learning, cf. Chapter 7.

## 6.4 Related Work

The problem of theory compression as introduced in this chapter is closely related to the traditional theory revision problem studied in inductive logic programming [Wrobel et al., 1996]. 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. Furthermore, it is grounded in a sound probabilistic framework. This framework bears some relationships to the PTR approach by Koppel et al. [1994] in that possible revisions in PTR are annotated with weights or probabilities. Still, PTR interprets the theory in a purely logical fashion to classify examples. It only uses the weights as bias during the revision process in which it also updates them using examples in a kind of propagation algorithm. Once the weights become close to 0, clauses are deleted. ProbLog compression is also somewhat related to Zelle and Mooney's work on Chill [Zelle and Mooney, 1994] in that it specializes an overly general theory but differs again in the use of a probabilistic framework. In the context of probabilistic logic languages, PFORTE [Paes et al., 2005] is a theory revision system using BLPs [Kersting and De Raedt, 2008] that follows a hill-climbing approach similar to the one used here, but with a wider choice of revision operators.

Several related approaches to reduce the size of a probabilistic network have been developed in the context of Biomine. One line of work aims at finding the most reliable subgraph connecting a set of query nodes. The algorithm introduced in [Hintsanen, 2007] for the case of two query nodes is similar to ProbLog theory compression in that it also removes edges in order of relevance. However, relevance for all edges is estimated using Monte Carlo simulation in a first phase, and values are not updated during edge removal. Methods that construct reliable subgraphs for two or more query nodes by adding edges to an initially empty subgraph instead of deleting them from the full graph are presented in [Hintsanen and Toivonen, 2008] and [Kasari et al., 2010]. Finally, the general framework to simplify networks by removing edges introduced in [Toivonen et al., 2010] differs from the approaches discussed so far in that it is not based on examples, but prunes edges that do not affect the quality of the best path between any pair of nodes in the network.

## 6.5 Conclusions

Using ProbLog, we have introduced a novel framework for theory compression in large probabilistic databases. We defined a new type of theory compression problem, namely finding a small subset of a given program that maximizes the likelihood w.r.t. a set of positive and negative examples. A solution to the ProbLog theory compression problem has then been developed. As for ProbLog inference, BDDs play a central role in making theory compression efficient. Theory compression as

discussed here is a restricted form of theory revision with a single operator that deletes facts by setting their probabilities to 0. However, the setting and algorithm could easily be extended to other operators, such as turning probabilistic facts into logical ones by setting their probabilities to 1. Finally, we have shown that ProbLog theory compression is not only theoretically interesting, but is also applicable on various realistic problems in a biological link discovery domain.

# Chapter 7

# Parameter Learning*

In the past few years, the statistical relational learning community has devoted a lot of attention to learning both the structure and parameters of probabilistic logics, cf. [Getoor and Taskar, 2007; De Raedt et al., 2008a], but so far seems to have devoted little attention to the learning of probabilistic database formalisms. Probabilistic databases such as ProbLog and the formalism of Dalvi and Suciu [2004] associate probabilities to facts, indicating the probabilities with which these facts hold. This information is then used to define and compute the success probability of queries or derived facts or tuples, which are defined using background knowledge (in the form of predicate definitions). As one example, imagine a life scientist mining a large network of biological entities in an interactive querying session, such as the Biomine network of Sevon et al. [2006], cf. Section 2.5. Interesting questions can then be asked about the probability of the existence of a connection between two nodes, or the most reliable path between them.

The key contribution of the present chapter is the introduction of a novel setting for learning the parameters of a probabilistic database from examples together with their target probability. The task then is to find those parameters that minimize the least squared error w.r.t. these examples. The examples themselves can either be queries or proofs, that is, explanations in terms of ProbLog. This learning setting is then incorporated in the probabilistic database ProbLog, though it can easily be integrated in other probabilistic databases as well. This yields the second key contribution of the chapter, namely an effective learning algorithm. It performs gradient-based optimization utilizing advanced data-structures for

*This chapter presents joint work with Bernd Gutmann, Kristian Kersting and Luc De Raedt published in [Gutmann et al., 2008]. The author has contributed to the development of the learning setting and the algorithms. As the implementation and experiments, the extensions of this work appearing in the technical report [Gutmann et al., 2010b] are mostly contributions of Bernd Gutmann. We therefore do not discuss these extensions here.

efficiently computing the gradient. This efficient computation of the gradient allows us to estimate a ProbLog program from a large real-world network of biological entities in our experiments, which can then be used for example by a life scientist in interactive querying sessions.

We proceed as follows. After reviewing related work in Section 7.1, we formally introduce the parameter estimation problem for probabilistic databases in Section 7.2. Section 7.4 presents our least-squares approach for solving it, based on the gradient derived in Section 7.3. Before concluding, we present the results of an extensive set of experiments on a real-world data set in Section 7.5.

## 7.1 Related Work

The distinguishing features of the new learning setting introduced in this chapter are the use of probabilistic databases, the integration of learning from entailment and learning from proofs, and the fact that examples are labeled with their desired probabilities.

This learning setting is in line with the general theory of probabilistic logic learning [De Raedt and Kersting, 2003] and inductive logic programming. From an inductive logic programming perspective, a query corresponds to a formula that is entailed by the database, and hence, learning from queries corresponds to the well-known learning from entailment setting. On the other hand, a proof does not only show *what* was proven but also *how* this was realized. An analogy with a probabilistic context-free grammar is useful here. One can learn the parameters of such a grammar starting from sentences belonging to the grammar (learning from entailment / from queries), or alternatively, one could learn them from parse-trees (learning from proofs), cf. the work on tree-bank grammars [Charniak, 1996; De Raedt et al., 2005]. The former setting is typically a lot harder than the latter one because one query may have multiple proofs, thus introducing hidden parameters into the learning setting, which are not present when learning from parse-trees. In this chapter, both types of examples can be combined, and to the best of our knowledge, it is the first time within relational learning and inductive logic programming that learning from proofs is integrated with learning from entailment.

However, statistical relational learning approaches usually assume a generative model defining a distribution over training examples. For stochastic logic programs (SLPs) [Cussens, 2001] (and probabilistic context-free grammars) as well as for PRISM [Sato and Kameya, 2001], the learning procedure assumes that ground atoms for a *single* predicate (or in the grammar case, sentences belonging to the language) are sampled and that the sum of the probabilities of all different atoms obtainable in this way is at most 1. Recently, Chen et al. [2008] proposed a learning setting where examples are labeled with probabilities in the context of SLPs. The

probabilities associated with examples, however, are viewed as specifying the degree of being sampled from some distribution given by a generative model, which does not hold in our case. Furthermore, they only provide an algorithm for learning from facts and not from both queries and proofs as we do. Probabilistic relational models (PRMs) [Friedman et al., 1999] and Bayesian logic programs (BLPs) [Kersting and De Raedt, 2008] are relational extensions of Bayesian networks using entity relationship models or logic programming respectively. In both frameworks, possible worlds, i.e. interpretations, are sampled, and the sum of the probabilities of such worlds is 1. Consider now learning in the context of probabilistic networks. It is unclear how different paths could be sampled and, clearly, the sum of the probabilities of such paths need not be equal to 1. Probabilistic databases define a generative model at the level of interpretations. However, as an interpretation states the truth-value of all ground atoms in an example, this is a challenging setting. When considering substructures or paths in a network, the sheer number of them makes explicitly listing interpretations virtually impossible. These difficulties explain – in part – why so far only few learning techniques for probabilistic databases have been developed. As we aim at applying our learning approach to biological network mining, we focus on learning from entailment and from proofs in this chapter. However, Gutmann et al. [2010c] recently introduced an algorithm for learning ProbLog parameters from partial interpretations, where BDDs are used to compactly encode all interpretations extending a partial interpretation without explicitly generating all true atoms of each such interpretation.

Within the probabilistic database community, parameter estimation has received surprisingly few attention. Nottelmann and Fuhr [2001] consider learning probabilistic Datalog rules in a similar setting also based on the distribution semantics. However, their setting and approach also significantly differ from ours. First, a single probabilistic target predicate only is estimated whereas we consider estimating the probabilities attached to definitions of multiple predicates. Second, their approach employs the training probabilities only to generate training examples labeled with 0/1 randomly according to the observed probabilities whereas we use the observed probabilities directly. Finally, while our algorithm follows a principled gradient approach employing (in principle) all combinations of proofs or explanations, they follow a two-step bootstrapping approach first estimating parameters as empirical frequencies among matching rules and then selecting the subset of rules with the lowest expected quadratic loss on an hold-out validation set. Gupta and Sarawagi [2006] also consider a closely related learning setting but only extract probabilistic facts from data.

Finally, the new setting and algorithm are a natural and interesting addition to the existing learning algorithms for ProbLog. It is most closely related to the theory compression setting discussed in Chapter 6, where the task is to remove all but the $k$ best facts from the database (that is to set the probability of such facts to 0), which realizes an elementary form of theory revision. The present task extends the

compression setting in that parameters of *all* facts can now be *tuned* starting from evidence. This realizes a more general form of theory revision [Wrobel et al., 1996], albeit that only the parameters are changed and not the structure.

## 7.2 Parameter Learning in Probabilistic Databases

Within probabilistic logical and relational learning [De Raedt and Kersting, 2003; De Raedt, 2008], the task of parameter estimation can be defined as follows:

**Task 7.1 (Parameter Estimation)**

**Given**
- *a set of examples $X$,*
- *a probabilistic database or probabilistic logic theory $\mathcal{D}$,*
- *a probabilistic coverage relation $P^{\mathcal{D}}(e)$ that denotes the probability that the database $\mathcal{D}$ covers the example $x \in X$, and*
- *a scoring function score,*

**find** *parameters of $\mathcal{D}$ such that the scoring is optimal.*

The key difference with logical learning approaches is that the coverage relation becomes probabilistic. Furthermore, we explicitly target probabilistic examples, that is, the examples themselves will have associated probabilities. The reason is that such examples naturally arise in various applications. For instance, text extraction algorithms return the confidence, experimental data is often averaged over several runs, and so forth. As one illustration consider populating a probabilistic database of genes from MEDLINE[1] abstracts using off-the-shelf information extraction tools, where one might extract from a paper that gene $a$ is located in region $b$ and interacting with gene $c$ with a particular probability denoting the degree of belief; cf. [Gupta and Sarawagi, 2006]. This requires one to deal with probabilistic examples such as $0.6 : \texttt{locatedIn}(\texttt{a}, \texttt{b})$ and $0.7 : \texttt{interacting}(\texttt{a}, \texttt{c})$. Also in the context of the life sciences, Chen et al. [2008] report on the use of such probabilistic examples, where the probabilities indicate the percentage of successes in an experiment that is repeated several times.

Let us now investigate how we can integrate those two ideas, that is, the notion of a probabilistic example and learning from entailment and proofs, within the ProbLog formalism. When learning from entailment, examples are atoms or clauses that are logically entailed by a theory. Transforming this setting to ProbLog leads to examples that are logical queries, and given that we want to work with

[1] `http://medline.cos.com/`

probabilistic examples, these queries will have associated target probabilities. When learning from proofs in ProbLog, a proof corresponds to an explanation, that is, a conjunction of facts, cf. Equation (3.14), page 35, again with associated target probabilities. It is easy to integrate both learning settings in ProbLog because the logical form of the example will be translated to a DNF formula and it is this last form that will be employed by the learning algorithm anyway. The key difference between learning from entailment and learning from proofs in ProbLog is that the DNF formula is a conjunction when learning from proofs and a more general DNF formula when learning from queries.

**Example 7.1** *In Example 3.1, using the query* $\texttt{path}(\texttt{a},\texttt{c})$ *as example results in* $ac \vee (ab \wedge bc)$*, whereas the explanation* $\texttt{edge}(\texttt{a},\texttt{b}), \texttt{edge}(\texttt{b},\texttt{c})$ *results in* $ab \wedge bc$ *only.*

To the best of our knowledge, this is the first time that learning from proofs and learning from entailment are integrated in one setting.

By now we are able to formally define the learning setting addressed in this chapter:

**Task 7.2 (Parameter Learning in ProbLog)**

**Given**
- *a ProbLog program* $T$ *and*
- *a set of training examples* $\{q_i, \tilde{p}_i\}_{i=1}^{M}$*,* $M > 0$*, where each* $q_i \in \mathcal{Q}$ *is a query or proof and* $\tilde{p}_i$ *is the* $k$*-probability of* $q_i$*,*

**find** *a function* $h \in \mathcal{H}$ *with low approximation error on the training examples as well as on unseen examples, where* $\mathcal{H} = \{h : \mathcal{Q} \to [0,1] | h(.) = P_k^{T'}(.)\}$ *comprises all parameter assignments* $T'$ *for* $T$*.*

Note that in this definition we have chosen the $k$-probability as probabilistic coverage relation as this allows for maximal flexibility. The definition also leaves the question open how to measure a "low approximation error". In this chapter, we propose to use the mean squared error as *error function*

$$MSE(T) = \frac{1}{M} \sum\nolimits_{1 \le i \le M} \left(P_s^T(q_i) - \tilde{p}_i\right)^2 . \tag{7.1}$$

Our setting is related to the one considered by Kwoh and Gillies [1996] in that we learn from examples where not everything is observable and that we assume a distribution over the result (that is, whether a query fails or succeeds) and the examples are independent of one another. While in our case, all the observations are binary, Kwoh and Gillies use training examples which contain several variables. They show that minimizing the squared error for this type of problem corresponds

to finding a maximum likelihood hypothesis, provided that each training example $(q_i, \tilde{p}_i)$ is disturbed by an error term. The actual distribution of this error is such that the observed query probability is still in the interval $[0, 1]$.

Gradient descent is a standard way of minimizing a given error function. The tunable parameters are initialized randomly. Then, as long as the error does not converge, the gradient of the error function is calculated, scaled by the learning rate $\eta$, and subtracted from the current parameters. In the following sections, we derive the gradient of the MSE and show how it can be computed efficiently.

## 7.3 Gradient of the Mean Squared Error

Applying the sum and chain rule to Equation (7.1) yields the partial derivative

$$\frac{\partial MSE(T)}{\partial p_j} = \frac{2}{M} \sum_{1 \leq i \leq M} \underbrace{\left(P_s^T(q_i) - \tilde{p}_i\right)}_{\text{Part 1}} \cdot \underbrace{\frac{\partial\ P_s^T(q_i)}{\partial p_j}}_{\text{Part 2}} . \tag{7.2}$$

Part 1 can be calculated by a ProbLog inference call computing the success probability of Equation (3.16) (p. 35). It does not depend on $j$ and has to be calculated only once in every iteration of a gradient descent algorithm. Part 2 can be calculated as

$$\frac{\partial P_s^T(q_i)}{\partial p_j} = \sum_{I \models q_i} \delta_{jI} \prod_{\substack{f_x \in I^1 \\ x \neq j}} p_x \prod_{\substack{f_x \in I^0 \\ x \neq j}} (1 - p_x) \,, \tag{7.3}$$

where $\delta_{jI} := 1$ if $f_j \in I^1$ and $\delta_{jI} := -1$ if $f_j \in I^0$. It is derived by first deriving the gradient $\partial P^T(I)/\partial p_j$ for a fixed interpretation $I$ of $L^T$, which is straight-forward, and then summing over all interpretations $I$ where $q_i$ can be proven.

To ensure that all $p_j$ stay probabilities during gradient descent, we reparameterize the search space and express each $p_j \in \,]0, 1[$ in terms of the sigmoid function[2] $p_j = \sigma(a_j) := 1/(1 + \exp(-a_j))$ applied to $a_j \in \mathbb{R}$. This technique has been used for Bayesian networks and in particular for sigmoid belief networks [Saul et al., 1996]. We derive the partial derivative $\partial P_s^T(q_i)/\partial a_j$ in the same way as (7.3) but we have to apply the chain rule once more due to the $\sigma$ function

$$\sigma(a_j) \cdot (1 - \sigma(a_j)) \cdot \sum_{I \models q_i} \delta_{jI} \prod_{\substack{f_x \in I^1 \\ x \neq j}} \sigma(a_x) \prod_{\substack{f_x \in I^0 \\ x \neq j}} (1 - \sigma(a_x)). \tag{7.4}$$

[2]The sigmoid function can induce plateaus which might slow down a gradient-based search. However, it is unlikely that a plateau will spread out over several dimensions and we did not observe such a behavior in our experiments. If it happens though, one can take standard counter measures like simulated annealing or random restarts.

**Algorithm 7.1** Evaluating the gradient of a query efficiently by traversing the corresponding BDD, calculating partial sums, and adding only relevant ones.

```
1: function GRADIENT(BDD b, variable v_j)
2:    (val, seen) = GRADIENTEVAL(root(b), v_j)
3:    if seen = 1 then
4:       return val · σ(a_j) · (1 − σ(a_j))
5:    else
6:       return 0
7: function GRADIENTEVAL(node n, variable v_j)
8:    if n is the 1-terminal then
9:       return (1, 0)
10:   if n is the 0-terminal then
11:      return (0, 0)
12:   Let h and l be the high and low children of n
13:   (val(h), seen(h)) = GRADIENTEVAL(h, v_j)
14:   (val(l), seen(l)) = GRADIENTEVAL(l, v_j)
15:   if variable(n) = v_j then
16:      return (val(h) − val(l), 1)
17:   else if seen(h) = seen(l) then
18:      return (σ(a_n) · val(h) + (1 − σ(a_n)) · val(l), seen(h)))
19:   else if seen(h) = 1 then
20:      return (σ(a_n) · val(h), 1)
21:   else if seen(l) = 1 then
22:      return ((1 − σ(a_n)) · val(l), 1)
```

We also have to replace every $p_j$ in Equation (3.16) by $\sigma(p_j)$. Going over all interpretations $I$ in the last equation is infeasible. In the following section, we discuss how to calculate the gradient on BDDs instead and introduce the gradient descent algorithm for ProbLog.

## 7.4 Parameter Learning Using BDDs

We now extend Algorithm 4.5 for probability calculation on BDDs to the computation of the gradient (7.3). Both algorithms have a time and space complexity of O(number of nodes in the BDD) if intermediate results are cached.

Let us first consider a full decision tree instead of a BDD. Each branch in the tree represents a full interpretation, and thus a product $n_1 \cdot n_2 \cdot \ldots \cdot n_i$, where the $n_i$ are the probabilities associated to the corresponding variable assignment of nodes on the branch. The gradient of such a branch $b$ with respect to $n_j$ is $g_b = n_1 \cdot n_2 \cdot \ldots n_{j-1} \cdot n_{j+1} \cdot \ldots \cdot n_i$ if $n_j$ is true, and $-g_b$ if $n_j$ is false in $b$. As

**Algorithm 7.2** Gradient descent parameter learning. kBestBDD returns the $k$-probability and the corresponding BDD.

```
1: function GradientDescent(program L^T ∪ BK, training set {(q_j, p̃_j) 1 ≤
   j ≤ M}, constants η and k)
2:   initialize all a_j randomly
3:   while not converged do
4:     T := {σ(a_j) :: c_j | c_j ∈ L^T} ∪ BK
5:     Δa := 0
6:     for 1 ≤ i ≤ M do
7:       (p_i, BDD_i) := kBestBDD(q_i,T,k)
8:       y := (2/M) · (p_i − p̃_i)
9:       for 1 ≤ j ≤ n do
10:        Δa_j := Δa_j + y· Gradient(BDD_i, node_j)
11:    a := a − η · Δa
12:  return {σ(a_j) :: c_j | c_j ∈ L^T}
```

all branches in a full decision tree are mutually exclusive, the gradient w.r.t. $n_j$ can be obtained by simply summing the gradients of all branches ending in a leaf labeled 1. This literally corresponds to Equation (7.3). In BDDs however, isomorphic sub-parts are merged, and obsolete parts are left out. This implies that some paths from the root to the 1-terminal may not contain $n_j$, therefore having a gradient of 0. So, when calculating the gradient on the BDD, we have to keep track of whether $n_j$ appeared on a path or not. Given that the variable order is the same on all paths, we can easily propagate this information in our bottom-up algorithm, as described in Algorithm 7.1. Specifically, GradientEval$(n, n_j)$ calculates the gradient w.r.t. $n_j$ in the sub-BDD rooted at $n$. It returns two values: the gradient on the sub-BDD and a Boolean indicating whether or not the target node $n_j$ appears in the sub-BDD. If at some node $n$ the indicator values for the two children differ, we know that $n_j$ does not appear above the current node, and we can drop the partial result from the child with indicator 0. The indicator variable is also used on the top level: Gradient returns the value calculated by the bottom-up algorithm if $n_j$ occurred in the BDD and 0 otherwise.

The learning algorithm as shown in Algorithm 7.2 combines the BDD-based gradient calculation with a standard gradient descent search. Starting from parameters $\mathbf{a} = a_1, \ldots, a_n$ initialized randomly, the gradient $\Delta\mathbf{a} = \Delta a_1, \ldots, \Delta a_n$ is calculated, parameters are updated by subtracting the gradient, and updating is repeated until convergence. When using the $k$-probability with finite $k$, the set of $k$ best proofs may change due to parameter updates. After each update, we therefore recompute the set of proofs and the corresponding BDD.

## 7.5 Experiments

We set up experiments to investigate the following questions:

**Q1** Does our approach reduce the mean squared error on training and test data?

**Q2** Is our approach able to recover the original parameters?

Answering these first questions serves as a sanity check for the algorithm and our implementation.

**Q3** Is it necessary to update the set of $k$ best proofs in each iteration?

As building BDDs for all examples is expensive, building BDDs once and using them during the entire learning process can save significant amounts of resources and time. We are therefore interested in the effects this strategy has on the results.

**Q4** Can we obtain good results approximating $P_s$ by $P_k$ for finite (small) $k$?

Given that using BDDs to calculate $P_s$ is infeasible for huge sets of proofs, as they occur in our application, where we easily get hundreds of thousands of proofs, we are interested in fast, reliable approximations.

**Q5** Do the results improve if parts of the training examples are given as proof?

Here we are interested in exploring the effects of providing more information in the form of proofs, which is one of the main distinguishing features of our algorithm.

To answer these questions, we use the Biomine subgraphs ALZHEIMER1 and ASTHMA1, cf. Appendix A. From these graphs we generated 3 sorts of training sets:

1. We sampled 500 random node pairs $(a, b)$ from each graph and estimated the query probability for `path(a,b)` using $P_5$, the probability of the 5 best proofs. These two sets are used to answer **Q1**, **Q2**, and **Q3**.
2. We sampled 200 random node pairs $(a, b)$ from ASTHMA1 and estimated $P_s(\texttt{path}(\texttt{a}, \texttt{b}))$ using the lower bound of bounded approximation (Algorithm 4.6) with interval width $\delta = 0.01$. This set is used to answer **Q4**.
3. We sampled 300 random node pairs $(a, b)$ and calculated $P_x(\texttt{path}(\texttt{a}, \texttt{b}))$, the probability of the best path between $a$ and $b$. We then build several sets where different fractions of the examples where given as the best explanation of `path(a,b)`, instead of the query itself, and used them to answer **Q5**.

To assess results, we use the root mean squared error on the test data $\sqrt{MSE_{\text{test}}}$, cf. Equation (7.1), and the mean absolute difference $MAD_{\text{facts}}$ between learned $p_j$ and original fact probabilities $p_j^{\text{true}}$:

$$MAD_{\text{facts}} := n^{-1} \sum_{j=1}^{n} |p_j - p_j^{\text{true}}|. \tag{7.5}$$

We always sampled the initial fact parameters uniformly in the interval $[-0.5, 0.5]$. Applying the sigmoid function yields probability values with mean $0.5 \pm 0.07$. The datasets used had fact probabilities in this range and we therefore got lower initial errors than by completely random initialization. In general, one can utilize prior knowledge to initialize the parameters. We performed 10-fold cross-validation in all experiments. The learning rate $\eta$ was always set to the number of training examples, as this has been found to speed up convergence. The algorithm was implemented in Prolog (Yap-5.1.3) using CUDD for BDD operations.

**Q1, Q2: Sanity Check** We attached probabilities to queries in the training set based on the best $k = 5$ proofs. The same approximation was used in the gradient descent algorithm, where the set of proofs to build the BDD was determined anew in every iteration as stated in Algorithm 7.2. We repeated the experiment using a total of 100, 300, and 500 examples, which we each split in ten folds for cross-validation. We thus used 90, 270, and 450 training examples. The more training examples are used, the more time each iteration takes. In the same amount of time, the algorithm therefore performs less iterations when using more training examples. The right column of Figure 7.1 shows the root mean squared error on the test data during learning. The gradient descent algorithm reduces the MSE on both training and test data, with significant differences in all cases (two-tailed t-test, $\alpha = 0.05$). These results affirmatively answer **Q1**.

The $MAD_{\text{facts}}$ error is reduced as can be seen in the right column of Figure 7.2. Again, all differences are significant (two-tailed t-test, $\alpha = 0.05$). Using more training examples results in faster error reduction. This answers **Q2** affirmatively. It should be noted however that in other domains, especially with limited or noisy training examples, minimizing the MSE might not reduce $MAD_{\text{facts}}$, as the MSE is a non-convex non-concave function with local minima.

**Q3: Error made if the best proofs are not updated** We repeated the same series of experiments, but without updating the set of proofs used for constructing the BDDs. The evolvement of $\sqrt{MSE_{\text{test}}}$ as well as of $MAD_{\text{Facts}}$ is plotted in the left column of Figures 7.1 and 7.2 respectively.

The plots for ASTHMA1 are hardly distinguishable and there is indeed no significant difference (two-tailed t-test, $\alpha = 0.05$). However, the runtime decreases by orders of magnitude, since searching for proofs and building BDDs are expensive operations

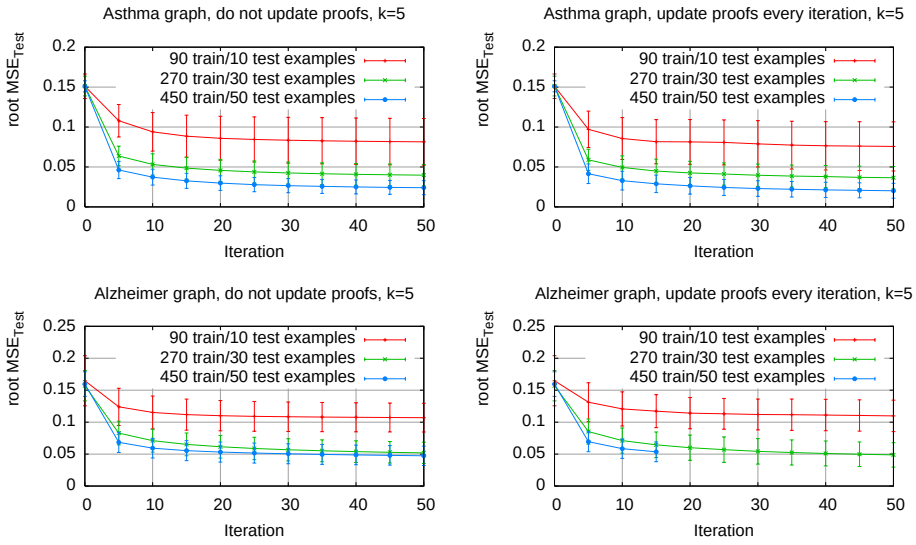



Figure 7.1: $\sqrt{MSE_{\text{test}}}$ for ASTHMA1 and ALZHEIMER1 using the 5 best proofs ($k = 5$); if the BDDs and proofs are not updated (left column); if they are updated every iteration (right column) (**Q2** and **Q3**)

which had to be done only once in the current experiments. Not updating the BDDs results in a speedup of 10 for ALZHEIMER1. For ALZHEIMER1 there is no significant difference for the $MSE_{\text{test}}$ (two-tailed t-test, $\alpha = 0.05$), but $MAD_{\text{facts}}$ is reduced a little slower (in terms of iterations) when the BDDs are kept constant. However, in terms of time this is not the case. These results indicate that BDDs can safely be kept fixed during learning in this domain, which affirmatively answers **Q3**.

**Q4: Less proofs, more speed, and still the right results?** In the next experiment, we studied the influence of the number $k$ of best proofs used during learning on the results. We considered ASTHMA1 with the second dataset, where training example probabilities are lower bounds obtained from bounded approximation with interval width 0.01. During learning, $P_k$ was employed to approximate probabilities.

We ran the learning algorithm on this dataset and used different values of $k$ ranging from 10 to 5000. We thus learned parameters using an underestimate of the true function, as $k$ best proofs may ignore a potentially large number of proofs used originally. Figure 7.3 shows the results for this experiment after 50 iterations of gradient descent. As can be seen, the average absolute error per fact ($MAD_{\text{facts}}$) goes down slightly with higher $k$. The difference is statistically significant for

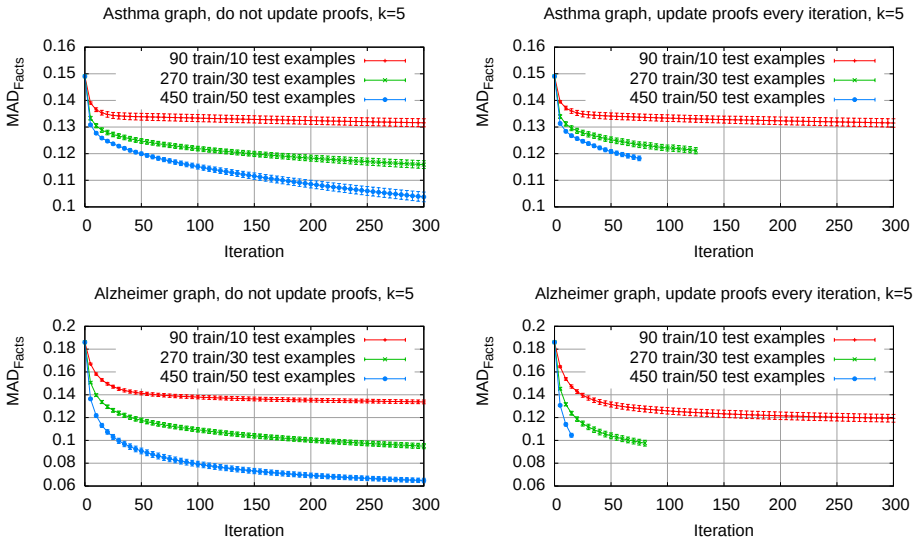



Figure 7.2: $MAD_{\text{facts}}$ for ASTHMA1 and ALZHEIMER1 using the 5 best proofs ($k = 5$); if the BDDs and proofs are not updated (left column); if they are updated every iteration (right column) (**Q2** and **Q3**)

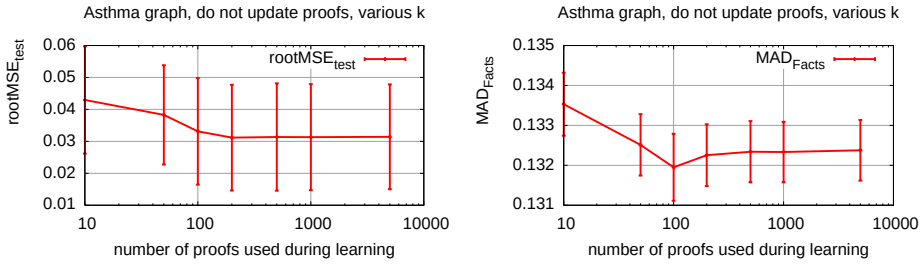



Figure 7.3: $MAD_{\text{facts}}$ and $\sqrt{MSE_{\text{test}}}$ after 50 iterations for different $k$ (number of best proofs used) on ASTHMA1 where training examples carry $P_s$ probabilities (**Q4**)

$k = 10$ and $k = 100$ (two-tailed t-test, $\alpha = 0.05$), but using more than 200 proofs has no significant influence on the error. The MSE also decreases significantly (two-tailed t-test, $\alpha = 0.05$) comparing the values for $k = 10$ and $k = 200$, but using more proofs has no significant influence. It takes more time to search for more proofs and to build the corresponding BDDs. These results indicate that using only 100 proofs is a sufficient approximation in this domain and affirmatively answer **Q4**.

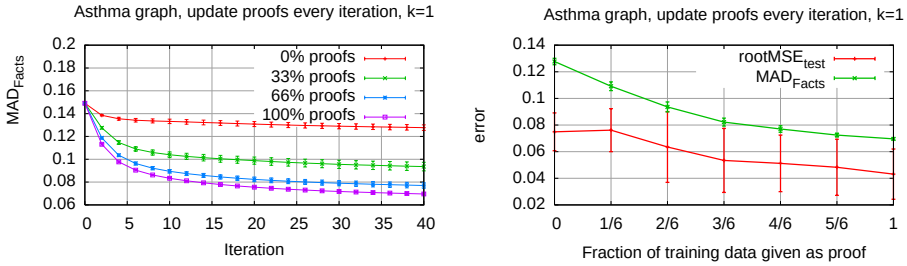



Figure 7.4: $MAD_{\text{facts}}$ and $\sqrt{MSE_{\text{test}}}$ after 40 iterations on ASTHMA1 if different fractions of the data are given as proof (**Q5**)

**Q5: Learning from Proofs and Queries** To investigate the effect of using both proofs and queries as examples, we computed the most likely explanation and its probability for 300 examples per graph. For each example, we either used the query or the most likely explanation, both with the explanation probability. Learning used $k = 1$. We used proofs for $0, 50, \ldots, 300$ examples and queries for the remaining ones, and performed stratified 10-fold cross-validation, that is the ratio of examples given as queries and as proofs was the same in every fold. We updated BDDs in every iteration. Figure 7.4 shows the results of this experiment. The curve on the left side indicates that the error per fact ($MAD_{\text{facts}}$) goes down faster in terms of iterations when increasing the fraction of proofs. Furthermore, the plot on the right side shows that the root MSE on the test set decreases. These results answer **Q5** affirmatively.

## 7.6 Conclusions

We have introduced a setting for learning the parameters of probabilistic databases that unifies learning from entailment and learning from proofs. We instantiated this general setting for ProbLog and developed a gradient-based algorithm to learn ProbLog parameters. The effectiveness of the method has been demonstrated on real biological datasets. Interesting directions for future research include conjugate gradient techniques and regularization-based cost functions. Those enable domain experts to successively refine probabilities of a database by specifying training examples.

# Conclusions Part II

This part of the thesis has studied the use of machine learning techniques to improve ProbLog programs based on example queries.

In Chapter 6, we introduced the task of theory compression, a form of theory revision restricted to the deletion of probabilistic facts, which aims at reducing a ProbLog database to a size that allows for inspection by a human expert while focusing on the part of the database most relevant for a set of example queries. We presented a greedy algorithm for solving this task that relies on ProbLog's BDD representation of explanation sets for efficient scoring of candidate deletions, which subsequently has been evaluated in the context of both real and artificial network data, demonstrating its practical applicability on various realistic problems in a biological link discovery domain. Extending theory compression with additional revision operators to realize a more general form of theory revision for probabilistic theories is a promising direction for future work.

In Chapter 7, we introduced a new parameter learning setting for probabilistic databases. As such databases do not define a distribution over all ground atoms, but distributions over the truth values of individual ground atoms, example queries in this setting are labeled with their desired probability. We presented a gradient descent method to minimize the mean squared error which exploits BDDs to calculate the gradient. The approach has been experimentally validated on real biological datasets. Interesting directions for future research include more sophisticated approaches to parameter learning in the new setting, such as conjugate gradient techniques and regularization-based cost functions.

# Part III

# Reasoning by Analogy

# Outline Part III

This part is devoted to the task of **Reasoning by Analogy**. The notion of analogy employed here is based on explanations in a relational language. In a probabilistic logic setting, such explanations can be used to rank examples by probability, and hence, analogy. While obtaining such a ranking corresponds to basic probabilistic inference in ProbLog, we contribute two new methods to learn explanations in probabilistic settings with or without domain knowledge, both based on well-known relational learning techniques, thereby further illustrating the use of ProbLog as a framework for lifting traditional ILP tasks to the probabilistic setting.

We introduce **Probabilistic Explanation Based Learning** (PEBL), where the problem of multiple explanations as encountered in classical explanation based learning is resolved by choosing the most likely explanation. PEBL deductively constructs explanations by generalizing the logical structure of proofs of example queries in a domain theory defining a target predicate.

**Probabilistic Local Query Mining** extends existing multi-relational data mining techniques to probabilistic databases. It thus follows an inductive approach, where the language of explanations is defined by means of a language bias and the search is structured using a refinement operator. Furthermore, negative examples can be incorporated in the score to find correlated patterns.

# Chapter 8

# Inductive and Deductive Probabilistic Explanation Based Learning*

Reasoning by analogy uses common properties of entities, such as shared structure or participation in the same relations, to transfer information between them. It is a strategy adopted in many contexts, reaching from everyday decisions such as which movie to watch (based on similar ones seen before) or whether to take an umbrella (based on days with similar weather conditions) to more complex problems such as diagnosing a patient (based on patients with similar symptoms) or deciding on whether to grant a loan (based on similar customers). The properties and relations underlying analogical reasoning can conveniently be represented in first order logic languages, allowing one to formally capture the notion of analogy. Furthermore, extending such representations towards probabilistic logic languages makes it possible to quantify the degree of analogy in terms of probabilities. In this chapter, we use ProbLog as a framework for finding analogous examples based on both relational and probabilistic information, though other probabilistic logic languages could be used as well. Imagine for instance a life scientist exploring the biological database described in Section 2.5, studying genes related to a specific disease. He might know some genes that are relevant and be interested in finding genes that are connected to the disease in analogous ways. In terms of the network representation, such connections could be expressed as simple paths using certain combinations of edge types, but more complex graph structures are possible as well. Ranking these different types of connections according to their probabilities

*This chapter builds on [Kimmig et al., 2007; Kimmig and De Raedt, 2008, 2009]

on the given examples gives a clear criterion for choosing the concept definition to be used for measuring analogy of unseen examples.

Reasoning by analogy in the context of probabilistic logics thus relies on a relational concept definition or *explanation* that can be used to rank possible examples. Using a probabilistic framework affects both the construction of such explanations and their use for ranking examples, as the coverage relation – between potential explanations and given examples, or between the chosen explanation and potential answer tuples – is no longer a hard 0/1-decision, but a gradual probabilistic score, which can be used to rank candidates. The explanations themselves are purely logical, and the probabilistic information resides in the database only, similar in spirit to ProbLog's separation of probabilistic facts and background knowledge.

While ranking candidate examples given an explanation corresponds to regular ProbLog inference, the key task to be studied in this chapter is that of learning explanations. More specifically, we consider two types of explanations, either based on general domain knowledge, or on database information only. If domain knowledge is available, it can be used to deductively construct explanations for given examples, as has been done in explanation based learning [Mitchell et al., 1986; DeJong, 2004], where a proof of a given example is abstracted to obtain an explanation that can be applied to similar examples. In the absence of domain knowledge, one can resort to inductive techniques that construct explanations solely based on the database, such as relational query mining [Dehaspe et al., 1998; De Raedt and Ramon, 2004]. In this chapter, we extend both techniques to use probabilistic scoring functions to learn best explanations, thus realizing a deductive as well as an inductive approach to explanation based learning in a probabilistic setting.

This chapter is organized as follows: we introduce the general setting in more detail in Section 8.1, followed by a summary of the underlying non-probabilistic techniques of explanation based learning (Section 8.2.1) and query mining (Section 8.2.2). Their probabilistic counterparts are introduced in Section 8.3. Section 8.4 discusses the implementations used in Section 8.5 for experiments in the context of biological link mining. Finally, Section 8.6 touches upon related work, and Section 8.7 concludes.

## 8.1 Explanation-based Analogy in Probabilistic Logic

The key notion of this chapter is that of an *explanation* in a relational language which specifies *analogy* between examples.[1] Combining such an explanation with

[1] In this chapter, we use the term *proof* for ProbLog's explanations (conjunctions of ground probabilistic facts supporting a query, cf. Equation (3.14), p. 35), and *explanation* for the explanation clauses used to reason by analogy (cf. Equation (8.1)).

probabilistic information makes it possible to rank analogous examples. We use ProbLog as underlying language here, though other probabilistic relational languages could be used as well. More specifically, an *explanation clause* (or *explanation* for short) is a conjunctive query of the form

$$q(\mathbf{X}) \quad :- \quad l_1, \ldots, l_m \tag{8.1}$$

where $q/n$ is a predicate not appearing in a given ProbLog program $T$, the $l_i$ are positive literals using predicates from $T$ and the variable vector $\mathbf{X} = (X_1, \ldots, X_n)$ represents example tuples. Such an explanation is said to *cover* a ground tuple $q(c_1, \ldots, c_n)$ if the success probability of $q(c_1, \ldots, c_n)$ in $T$ extended by the explanation is non-zero. All tuples in $T$ covered by a given explanation $q$ are considered *analogous*. Furthermore, covered tuples can be ranked by probability.

**Example 8.1** *Within bibliographic data analysis, the similarity structure among items can improve information retrieval results. Consider a collection of papers* $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}$ *and some pairwise similarities* $\mathtt{alike}(\mathtt{a}, \mathtt{c})$*, e.g., based on key word analysis, cf. also Figure 3.1.*

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

*The explanation*

$$\begin{array}{lll} \mathtt{q}(\mathtt{X}_1, \mathtt{X}_2) & :- & \mathtt{alike}(\mathtt{X}_1, \mathtt{X}_3), \mathtt{alike}(\mathtt{X}_3, \mathtt{X}_4), \\ & & \mathtt{alike}(\mathtt{X}_4, \mathtt{X}_5), \mathtt{alike}(\mathtt{X}_5, \mathtt{X}_2). \end{array}$$

*covers tuple* $(\mathbf{a}, \mathbf{d})$ *with probability* $0.7 \cdot 0.6 \cdot 0.8 \cdot 0.5 = 0.168$.

The main task studied in this chapter can be formalized as follows:

**Task 8.1 (Explanation Learning)**

**Given**
- *a ProbLog program* $T$*,*
- *a language* $\mathcal{L}$ *of queries of the form* (8.1) *over the vocabulary of* $T$*,*
- *one or more training examples in the form of ground atoms* $\mathtt{id}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$*, and*
- *a function* $\psi^T_{\mathtt{id}/\mathtt{n}}$ *scoring queries on the examples,*

**find**
- *an explanation* $q \in \mathcal{L}$ *that maximizes* $\psi^T_{\mathtt{id}/\mathtt{n}}(q)$*.*

In this chapter, we propose two different approaches to explanation learning. While both extend well-known machine learning techniques that generate explanations in a purely relational setting to probabilistic logic languages, they are based on different types of input information. *Probabilistic explanation based learning* requires background knowledge defining the example predicate `id/n`, which is used to generate explanations *deductively* according to the structure of the most likely proof of an example $\texttt{id}(\texttt{c}_1, \ldots, \texttt{c}_\texttt{n})$. *Probabilistic query mining*, on the other hand, does not rely on such background knowledge and generates explanations *inductively* by successive refinement. Independent of the concrete setting used for explanation learning, the resulting explanation together with a ProbLog program over the same vocabulary defines a ranking of analogous examples.

## 8.2 Background: Constructing Explanations

This section reviews explanation based learning and query mining, two well-known techniques for learning explanations in relational, non-probabilistic domains. We conclude with a summary of their commonalities as well as their differences in Section 8.2.3.

### 8.2.1 Explanation Based Learning

When searching for explanations, it is intuitively appealing to use as much domain knowledge as possible in order to focus on plausible explanations. This idea forms the basis of explanation based learning (EBL), which was a popular research theme within the field of machine learning during the 80s. EBL is concerned with finding plausible, abstracted explanations for particular examples using a logical domain theory; we refer to [DeJong, 2004] for an overview and introduction. Traditional explanation based learning was largely studied within first order logic representations and explanations were built using deduction [Hirsh, 1987; Van Harmelen and Bundy, 1988], though there was sometimes also an abductive component [Cohen, 1992]. To make them applicable to other, analogous examples, concrete explanations or proofs of given examples were variabelized. The resulting abstracted explanations were then typically turned into rules that would be added to the theory, often with the aim of speeding up further inference or extending an imperfect domain theory.

EBL as conveniently formalized for Prolog [Hirsh, 1987; Van Harmelen and Bundy, 1988] computes an abstracted explanation from a concrete proof of an example. Explanations use only so-called *operational* predicates, i.e. predicates that capture essential characteristics of the domain of interest and should be easy to prove. More formally, this leads to the following task description:

**Task 8.2 (Explanation Based Learning (EBL))**

**Given**
- *an example* $\texttt{id}(\texttt{t}_1, \ldots, \texttt{t}_n)$
- *a pure Prolog program* $T$ *including a definition of the goal concept* `id/n`
- *a set of operational predicates occurring in the program*

**find** *an abstracted explanation in the form of a conjunction of literals with operational predicates based on a proof of the example.*

Explanation based learning starts from a definite clause theory $T$, that is a pure Prolog program, and an example in the form of a ground atom $id(t_1, ..., t_n)$. It then constructs a refutation proof of the example using SLD-resolution, cf. Section 2.1. Given such a proof for the example $id(t_1, ..., t_n)$, explanation based learning constructs an abstracted explanation, starting from the variabelized goal, i.e. $id(X_1, ..., X_n)$ with different variables $X_i$, and then performing the same SLD-resolution steps as in the proof for the example. The only difference is that in the general proof atoms $q(s_1, ..., s_r)$ for operational predicates $q$ in a goal $?-g_1, ..., g_i, q(s_1, ..., s_r), g_{i+1}, ..., g_n$ are not resolved away. Also, the proof procedure stops once the goal contains only atoms for operational predicates. The resulting goal provides an *abstracted explanation* for the example. In terms of proof trees, explanation based learning cuts off branches below operational predicates. It is easy to implement the explanation based proof procedure as a meta-interpreter in Prolog [Van Harmelen and Bundy, 1988; Hirsh, 1987].

**Example 8.2** *We extend Example 8.1 (ignoring probability labels for now) with background knowledge defining that two items* `X` *and* `Y` *are* `related(X,Y)` *if they are alike (such as* `a` *and* `c`*) or if* `X` *and some item* `Z` *which is related to* `Y` *are alike.*

```
related(X,Y) :- alike(X,Y).
related(X,Y) :- alike(X,Z), related(Z,Y).
```

*We declare* `alike/2` *to be the only operational predicate, and use* `related(b,e)` *as training example. EBL proves this goal using two instances of the operational predicate, namely* `alike(b,c)` *and* `alike(c,e)`*, and then produces the explanation* `alike(X,Z), alike(Z,Y)` *for the abstract example* `related(X,Y)`*. The successful branches of the SLD-trees for* `related(b,e)` *and the abstract example* `related(X,Y)` *are depicted in Figure 8.1; corresponding proof trees are in Figure 8.2. The result can be turned into an explanation of the form* (8.1) *by adding a suitable clause head:*

```
exp_related(X,Y) :- alike(X,Z), alike(Z,Y).
```

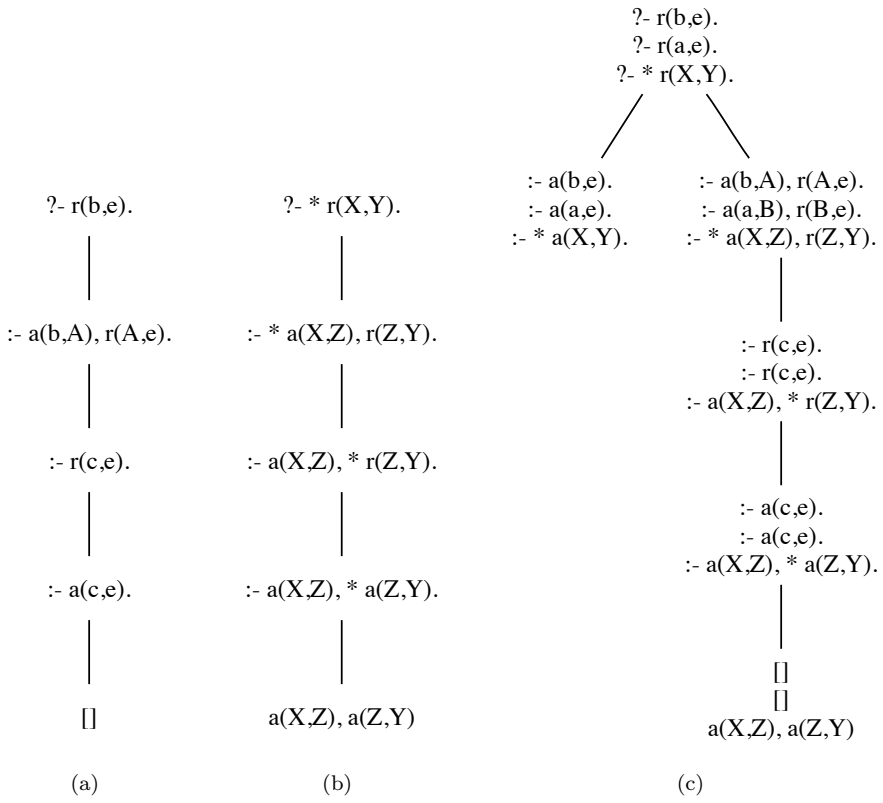



Figure 8.1: (a) The successful branch of the SLD-tree for `related(b,e)` in Example 8.2. (b) The corresponding branch for general goal `related(X,Y)`, where the asterisk separates the explanation constructed so far and the remaining query. (c) A partial SLD-tree for Example 8.12, where each node contains the current status for the two training examples as well as the general version.

### 8.2.2 Query Mining

In the absence of domain knowledge, alternative strategies have to be adopted to construct explanations based on the database only, for instance by finding patterns in the data. The traditional *local pattern mining* task is that of identifying those elements in a language of patterns that satisfy the constraints imposed by a *selection predicate* w.r.t. a given database [Mannila and Toivonen, 1997]. Numerous works have been devoted to local pattern mining since the introduction of item-set mining, cf. [Agrawal et al., 1996]. They can be distinguished alongst three main dimensions. The first is concerned with the type of pattern considered, that is

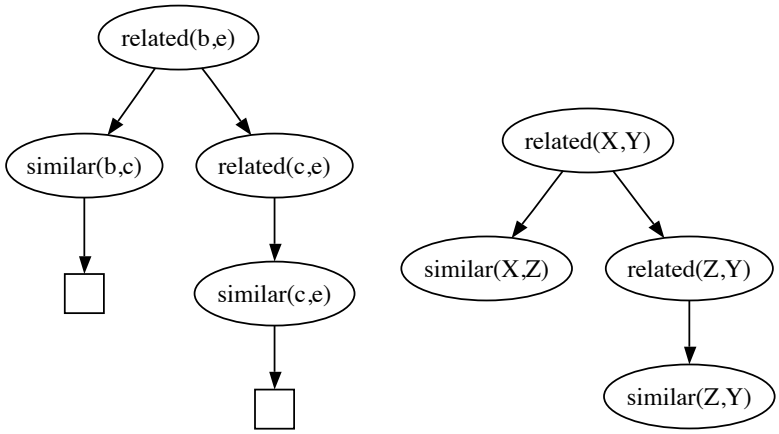



Figure 8.2: (a) Proof tree for `related`(`b`, `e`) in Example 8.2, and (b) for the abstract example `related`(`X`, `Y`), stopped at operational predicates.

whether one mines for item-sets, episodes and strings, graphs or relational patterns. The second is concerned with the nature of the selection predicate applied [Ng et al., 1998], such as whether one employs frequency, confidence, significance, lift, closedness, freeness, and so forth. Finally, the third distinguishes two types of desired answer sets: all patterns covered by the selection predicate in the case of frequent pattern mining [Agrawal et al., 1996], or the best $k$ patterns in the case of correlated pattern mining [Morishita and Sese, 2000].

Query mining upgrades traditional local pattern mining to the representations of multi-relational databases [Dehaspe et al., 1998]; it is also known under the name of *query flocks* [Tsur et al., 1998]. Queries in a relational database form a very general type of pattern that can emulate many other pattern domains, such as item-sets, sequences, trees and, as in our motivating application, graphs.

*Query Mining* aims at finding all queries satisfying a selection predicate $\phi$. It can be formulated as follows, cf. [Dehaspe et al., 1998; De Raedt and Ramon, 2004]:

**Task 8.3 (Query Mining)**

**Given**
- *a database $\mathcal{D}$ including the designated relation id,*
- *a language $\mathcal{L}$ containing queries of the form* (8.1) *over the vocabulary of $\mathcal{D}$, and*
- *a selection predicate $\phi$,*

**find** *all queries* $q \in \mathcal{L}$ *such that* $\phi^{\mathcal{D}}(q) = true$.

In contrast to EBL as defined in Task 8.2, query mining does not rely on background knowledge to deduce explanations, but induces them by combining literals that can be matched against the database, which can be seen as a way of hypothesizing explanations by freely combining individual properties. To exclude undesired combinations, for instance including isolated literals not related to the rest of the query, additional syntactic or semantic restrictions, called *bias*, can be imposed on the form of queries by explicitly specifying the language $\mathcal{L}$, cf. [Dehaspe et al., 1998; Tsur et al., 1998; De Raedt and Ramon, 2004].

To structure and prune the search, query mining typically relies on a generality relation $\preceq$ between patterns and requires the selection predicate to be anti-monotonic with respect to this relation. A predicate $\phi$ is *anti-monotonic* w.r.t. a generality relation $\preceq$ if and only if for all pairs of queries $q_1$ and $q_2$, if $q_1 \preceq q_2$ then $\phi(q_1) \geq \phi(q_2)$. The generality relation employed in this work is *OI*-subsumption [Esposito et al., 2000], as the corresponding notion of subgraph isomorphism is favorable within the intended application in network mining. More formally, a conjunctive query $q_1$ represented as a set of literals *OI-subsumes* a conjunctive query $q_2$, notation $q_1 \preceq q_2$, if and only if there exists a substitution $\theta = \{V_1/t_1, \cdots, V_n/t_n\}$ such that $q_1\theta \subseteq q_2$ and the $t_i$ are different constants not occurring in $q_1$.

**Example 8.3** *For queries* $Q1$ *to* $Q3$ *below, query* $Q1$ *OI-subsumes* $Q2$*, but neither* $Q1$ *nor* $Q2$ *OI-subsumes* $Q3$*: for* $Q1$*,* `Y` *and* `Z` *cannot both be mapped to* `c`*; for* $Q2$*,* `Y` *cannot be mapped to* `c` *as* `c` *already occurs in* $Q2$*.*

$(Q1)$ `q(X) :- alike(X,Y),alike(Y,Z).`

$(Q2)$ `q(X) :- alike(X,Y),alike(Y,c).`

$(Q3)$ `q(X) :- alike(X,c),alike(c,c).`

The most prominent selection predicate is minimum frequency, that is, selected queries have to cover a minimum number of tuples in the designated relation.

**Example 8.4** *Let* $\mathsf{COUNT}^{\mathcal{D}}(\mathsf{q}(*))$ *denote the number of different ground instances* $id(\mathbf{X})$ *for which* $q(\mathbf{X})$ *is provable in* $\mathcal{D}$*. Minimum frequency*

$$\phi^{\mathcal{D}}(q) = \mathsf{COUNT}^{\mathcal{D}}(\mathsf{q}(*)) \geq t \tag{8.2}$$

*is anti-monotonic with respect to OI-subsumption: if* $q_1 \preceq q_2$*, that is,* $q_1\theta \subseteq q_2$*,* $q_1$ *covers at least all the instances covered by* $q_2$*, and thus* $\mathsf{COUNT}^{\mathcal{D}}(\mathsf{q}_1(*)) \geq \mathsf{COUNT}^{\mathcal{D}}(\mathsf{q}_2(*))$.

Within the intended application in network mining, the pattern language can be restricted to connected subgraphs by requiring linkage of variables. A query $q(\mathbf{X}) : -l_1, ..., l_n$ is *linked* if and only if for all $i > 0$ at least one variable appearing in $l_i$ also appears in $q(\mathbf{X}) : -l_1, ..., l_{i-1}$.

**Example 8.5** *Queries $Q1$ and $Q2$ above are linked, but $Q3$ is not, as the last literal does not contain variables, and neither is the following query, where all variables in* $\texttt{alike}(\texttt{Y}, \texttt{Z})$ *are new:*

$$\texttt{q}(\texttt{X}) \ :- \ \texttt{alike}(\texttt{Y}, \texttt{Z}).$$

*Correlated Pattern Mining* [Morishita and Sese, 2000] uses both *positive* and *negative* examples, specified as two designated relations $id^+$ and $id^-$ of the same arity, to find the top $k$ patterns, that is, the $k$ patterns scoring best w.r.t. a function $\psi$. The function $\psi$ employed is convex, e.g. measuring a statistical significance criterion such as $\chi^2$, cf. [Morishita and Sese, 2000], and measures the degree to which the pattern is statistically significant or unexpected. As an instance of Task 8.3, correlated pattern mining thus uses the selection predicate

$$\phi^{\mathcal{D}}(q) = q \in \arg_k \max_{q \in \mathcal{L}} \psi^{\mathcal{D}}(q) \tag{8.3}$$

**Example 8.6** *Consider the database of Example 8.1 with $id^+ = \{a, c\}$ and $id^- = \{d, e\}$, ignoring probability labels. A simple correlation function is*

$$\psi^{\mathcal{D}}(q) = \mathsf{COUNT}^{\mathcal{D}^+}(\texttt{q}(*)) - \mathsf{COUNT}^{\mathcal{D}^-}(\texttt{q}(*)), \tag{8.4}$$

*where $\mathsf{COUNT}^{\mathcal{D}^+}(\texttt{q}(*))$ and $\mathsf{COUNT}^{\mathcal{D}^-}(\texttt{q}(*))$ are the numbers of different provable ground instances of $q$ among the tuples in $id^+$ and $id^-$, respectively. We obtain $\psi^{\mathcal{D}}(Q4) = 2 - 0 = 2$ and $\psi^{\mathcal{D}}(Q5) = 1 - 1 = 0$ for queries*

$$(Q4) \quad \texttt{q}(\texttt{X}) \ :- \ \texttt{alike}(\texttt{X}, \texttt{Y}), \texttt{alike}(\texttt{Y}, \texttt{Z}).$$

$$(Q5) \quad \texttt{q}(\texttt{X}) \ :- \ \texttt{alike}(\texttt{X}, \texttt{d}).$$

*Multi-relational query miners* such as the ones of [Dehaspe et al., 1998; De Raedt and Ramon, 2004] often follow a level-wise approach for frequent query mining [Mannila and Toivonen, 1997], where at each level new candidate queries are generated from the frequent queries found on the previous level. In contrast to Apriori, instead of a "joining" operation, they employ a refinement operator $\rho$ to compute more specific queries, and also manage a set of infrequent queries to take into account the specific language requirements imposed by $\mathcal{L}$. For instance, if $\mathcal{L}$ requires clauses to be linked, the language is not necessarily anti-monotonic.

**Algorithm 8.1** Query mining

1: **function** SELECTEDQUERIES(refinement operator $\rho$, database $\mathcal{D}$, anti-monotonic selection predicate $\phi$)
2: $C := \{q(\mathbf{X})\}$
3: $i := 0$
4: **while** $C \neq \emptyset$ **do**
5: $S_i := \{h \in C \mid \phi^{\mathcal{D}}(h) = true\}$
6: $V_i := C - S_i$
7: $C := \{h \in \rho(h') \mid h' \in S_i \text{ and } \neg\exists s \in \bigcup_j V_j : s \preceq h\}$
8: $i := i + 1$
9: return $\cup_i S_i$

**Example 8.7** *Consider the clause*

$$(Q6) \quad \mathtt{q(X)} \ :- \ \mathtt{alike(X,Y)}, \mathtt{alike(Y,Z)}.$$

*The minimal generalizations of Q6 are*

$$(Q7) \quad \mathtt{q(X)} \ :- \ \mathtt{alike(Y,Z)}.$$

$$(Q8) \quad \mathtt{q(X)} \ :- \ \mathtt{alike(X,Y)}.$$

*Before considering Q6, the Apriori algorithm would require that both Q7 and Q8 have been investigated at the previous level. However, as Q7 is not linked, it would not be generated by typical multi-relational data mining algorithms.*

Apart from this, the algorithm as outlined in Algorithm 8.1 proceeds in the usual level-wise way, starting at level $i = 0$ with the most general query $q(\mathbf{X})$ in its list of candidates $C$. At each level, all candidates of the current level are processed: if a query satisfies the anti-monotonic selection predicate $\phi$, it is added to the list $S_i$ of solutions, otherwise, it is added to the list $V_i$ of queries violating the selection predicate. The next set of candidates is the set of all queries that are immediate specializations obtained by applying the refinement operator $\rho$ on queries in $S_i$, but filtering out the ones that specialize a query in some $V_j$.

A key component of the algorithm is the refinement operator used to specialize queries. A *refinement operator* $\rho$ is a function $\rho : \mathcal{L} \to 2^{\mathcal{L}}$ such that

$$\forall q \in \mathcal{L} : \rho(q) \subseteq \{q' \in \mathcal{L} | q \preceq q'\}.$$

Various properties of refinement operators have been considered and are desirable for different types of algorithms [Nienhuys-Cheng and de Wolf, 1997]. For complete algorithms, which search for all solutions, it is essential that the operator is optimal w.r.t. $\mathcal{L}$. A refinement operator $\rho$ is *optimal* for $\mathcal{L}$ if and only if for all queries

$q \in \mathcal{L}$ there is exactly *one* sequence of queries $q_0, ..., q_n$ such that $q_0 = q(\mathbf{X})$ (the most general element in the search space) and $q_n = q$ for which $q_{i+1} \in \rho(q_i)$. Optimal refinement operators thus ensure that there is exactly one path from the most general query $q(\mathbf{X})$ to every query in the search space. This is typically achieved by defining a canonical form for possible querys, often based on some lexicographic order. For instance, in item-set mining, patterns are canonically represented as sorted lists and refined by adding a new item to the end of the list, respecting the order. The graph mining algorithm gSpan [Yan and Han, 2002] employs canonical labels for graphs to structure the search space. Various canonical forms for conjunctive queries have been studied, cf. [Nijssen and Kok, 2003; De Raedt and Ramon, 2004; Garriga et al., 2007]. Here, we shall employ a canonical form based on ordering literals primarily on their arguments. This choice is motivated by Prolog's evaluation mechanism, where additional atoms using the same variables but different predicates can lead to early failure and thus prune the search space. For instance, when evaluating a given subgraph query in the network setting, checking the type of a node early can avoid matching part of the pattern for nodes of the wrong type, which would not be possible for an order primarily based on predicates.

A query $q(\mathbf{X}) : -l_1, \ldots, l_n$ with variables $V_1, \ldots, V_m$ (ordered according to their first occurrence from left to right) is in *canonical form* if and only if $[l_1, \ldots, l_n]$ is smallest among all $<_l$-ordered lists that can be obtained using the same type of variable numbering for permutations of $[l_1, \ldots, l_n]$. Given a user-defined order on predicate names $<_p$, the order $<_l$ on literals is defined as follows:

1. Constants are ordered lexicographically.
2. Variables are ordered by first occurrence: $V_i <_l V_j$ iff $i < j$
3. Constants come before variables: $c <_l V$ for constants $c$ and variables $V$
4. Literals with same predicate (but possibly different arity) are ordered lexicographically using the order on terms specified above:
   - $p(t_1) <_l p(t_1, t_2, \ldots)$
   - $p(t_1, \ldots) <_l p(t'_1, \ldots)$ if $t_1 <_l t'_1$
   - $p(t_1, t_2, \ldots) <_l p(t_1, t'_2, \ldots)$ iff $p(t_2, \ldots) <_l p(t'_2, \ldots)$
5. Literals with identical arguments are ordered by predicate name: $p_1(\mathbf{t}) <_l p_2(\mathbf{t})$ iff $p_1 <_p p_2$

**Example 8.8** *While $Q9$ is in canonical form, the logically equivalent $Q10$ is not, as it can be transformed into $Q9$ by exchanging* `alike(X,D)` *and* `alike(X,E)`*:*

$(Q9)$ `q(X) :- alike(X,A),alike(X,B),alike(A,C).`

$(Q10)$ `q(X) :- alike(X,D),alike(X,E),alike(E,F).`

**Algorithm 8.2** Correlated pattern mining

**function** CORRELATEDQUERIES(refinement operator $\rho$, database $\mathcal{D}$, constant $k$, correlation measure $\psi$, upper bound function $u$)
  $R := \{q(\mathbf{X})\}$
  $i := 0$; $t := -\infty$; $T = \emptyset$
  **while** $R \neq \emptyset$ **do**
    $C := R \cup T$
    **if** $|C| \geq k$ **then**
      Let $q$ be the $k$-th best query in $C$ according to $\psi^{\mathcal{D}}$
      $t := \psi^{\mathcal{D}}(q)$
    $T := \{h \in C \mid \psi^{\mathcal{D}}(h) \geq t\}$
    $V_i := \{h \in C \mid u^{\mathcal{D}}(h) < t\}$
    $S := \{h \in C \mid u^{\mathcal{D}}(h) \geq t\}$
    $R := \{h \in \rho(h') \mid h' \in S \text{ and } \neg\exists s \in \bigcup_j V_j : s \preceq h\}$
    $i := i + 1$
  return $T$

The *optimal refinement operator* $\rho_o$ used in this work ensures that queries are linked and is defined as

$$\begin{aligned}\rho_o(Q) = \{[Q, p(X_1, \ldots, X_t)] \mid\ & p \text{ a relation} \\ & \text{and } \exists i \leq t : X_i \text{ occurs in } Q \text{ and} \\ & [Q, p(X_1, \ldots, X_t)] \text{ is in canonical form}\}\end{aligned} \tag{8.5}$$

**Example 8.9** *There are four optimal refinements of query Q5:*

```
q(X) :- alike(X,d),alike(X,e).
q(X) :- alike(X,d),alike(X,X).
q(X) :- alike(X,d),alike(X,Y).
q(X) :- alike(X,d),alike(Y,X).
```

*All other candidate literals are smaller than* `alike(X,d)` *according to* $<_l$ *and would therefore result in non-canonical patterns.*

Morishita and Sese [2000] adapt Apriori for finding the top $k$ patterns w.r.t. a *boundable* function $\psi$, i.e. for the case where there exists a function $u$ (different from a global maximum) such that $\forall g, s \in \mathcal{L} : g \preceq s \rightarrow \psi(s) \leq u(g)$.

**Example 8.10** *The function* $\psi^{\mathcal{D}}(q) = \mathsf{COUNT}^{\mathcal{D}^+}(\mathsf{q}(*)) - \mathsf{COUNT}^{\mathcal{D}^-}(\mathsf{q}(*))$ *introduced in Example 8.6 is upper-boundable using* $u(q) = \mathsf{COUNT}^{\mathcal{D}^+}(\mathsf{q}(*))$. *For*

| | query | $c^+$ | $c^-$ | $\psi$ |
|---|---|---|---|---|
| 1 | `q(X) :- alike(X,Y)` | **2** | 1 | **1** |
| 2 | `q(X) :- alike(X,a)` | 0 | 0 | 0 |
| 3 | `q(X) :- alike(X,b)` | **1** | 0 | **1** |
| 4 | `q(X) :- alike(X,c)` | **1** | 0 | **1** |
| 5 | `q(X) :- alike(X,d)` | **1** | 1 | 0 |
| 6 | `q(X) :- alike(X,e)` | **1** | 0 | **1** |
| 7 | `q(X) :- alike(Y,X)` | **1** | 2 | - 1 |
| 8 | `q(X) :- alike(a,X)` | **1** | 0 | **1** |
| 9 | `q(X) :- alike(b,X)` | **1** | 0 | **1** |
| 10 | `q(X) :- alike(c,X)` | 0 | 2 | -2 |
| 11 | `q(X) :- alike(d,X)` | 0 | 0 | 0 |
| 12 | `q(X) :- alike(e,X)` | 0 | 1 | -1 |

Table 8.1: Counts on $id^+$ and $id^-$ and $\psi$-values obtained during the first level of mining in Example 8.11. The current minimal score for best queries is 1, i.e. only queries with $\psi \geq 1$ or $c^+ \geq 1$ will be refined on the next level.

*any* $g \preceq s$, $\psi^{\mathcal{D}}(s) \leq \mathsf{COUNT}^{\mathcal{D}^+}(\mathsf{s}(*)) \leq \mathsf{COUNT}^{\mathcal{D}^+}(\mathsf{g}(*))$, *as* $\mathsf{COUNT}^{\mathcal{D}^-}(\mathsf{s}(*)) \geq 0$ *and* $\mathsf{COUNT}^{\mathcal{D}}$ *is anti-monotonic.*

Again, at each level new candidate queries are obtained from those queries generated at the previous level that qualify for refinement. More specifically, Algorithm 8.2 maintains a set $T$ containing the current top $k$ queries and a threshold $t$ corresponding to the lowest score of a query in $T$. At level $i$, the set of candidate queries $C$ contains the queries in $T$ as well as the ones obtained from the refinement operator in the previous level. After the set of best queries and the threshold have been updated, candidate queries with upper bound below the new threshold are stored in the set $V_i$ of queries known to violate the selection predicate. All other candidate queries are refined and the result is filtered against the discarded queries before being passed on to the next level.

**Example 8.11** *Assume we mine for the 3 best correlated queries in Example 8.6. Table 8.1 shows counts on* $id^+$ *and* $id^-$ *and* $\psi$*-values obtained during the first level of mining. The highest score achieved is 1. Queries 1, 3, 4, 6, 8, 9 are the current best queries and will thus be refined on the next level. Queries 5 and 7 have lower scores, but upper bound* $c^+ = 1$*, implying that their refinements may still belong to the best queries and have to be considered on the next level as well. The remaining queries are pruned, as they all have an upper bound* $c^+ = 0 < 1$*, i.e. all their refinements are already known to score lower than the current best queries.*

### 8.2.3 Deductive and Inductive EBL

Both explanation based learning and query mining are instances of explanation learning as specified in Task 8.1, albeit in a setting without probability labels. Their key difference is the type of reasoning they employ, which is motivated by the type of information available in the respective settings. In EBL, background knowledge encoding structural information about possible explanations is used to reason *deductively*, that is, to construct explanations based on the proof structure of a given example query. Such background knowledge is not available in query mining, which therefore follows an *inductive* approach of systematically generating explanations using a refinement operator. The learning process is guided by a kind of implicit background knowledge in the form of multiple examples that should or should not be considered analogous instead of by explicit structural information. The distinction between inductive and deductive generalization goes back to [Mitchell et al., 1986], where it has been argued that the power of deductive generalization lies in the ability to justify generalizations in terms of the background knowledge. However, inductive generalization is more flexible as it does not rely on the availability of explicit knowledge about the domain and concept of interest, but rather combines whatever information can be found in the database. While Mitchell et al. [1986] have restricted the use of the term *explanation-based* to the deductive setting, referring to the inductive setting as *similarity-based*, we use *explanation* in both settings to emphasize the common core of our approaches, which both learn explanations, though on somewhat different levels. The background knowledge in EBL offers a powerful tool to narrow the search space and to easily define complex explanation languages, such as for example a path connecting a set of input nodes. While EBL is thus often more efficient to use, this typically comes at the price of a more restricted explanation language. Indeed, in the network context, query mining might identify a path connecting a set of input nodes, but also more complex subgraph structures. From a user perspective, writing definite clauses as background knowledge for EBL is typically much easier than declaring types and modes to specify the language bias for query mining.

## 8.3 Incorporating Probabilistic Information

We now extend explanation based learning and query mining to take into account probabilistic information, which provides two alternative instances of explanation learning as defined in Task 8.1.

As we will see, besides the differences inherited from their logical counterparts, probabilistic explanation based learning (PEBL) and probabilistic query mining also differ in the way probabilities are integrated. While the explanation probability naturally lends itself as basis for probabilistic explanation based learning, the

choice of probability for query mining is open, and in principle, any of the exact or approximate inference methods discussed in Chapter 4 could be used. Furthermore, in PEBL, probabilities drive the search for the most likely explanation, that is, probabilistic and logical inference are interleaved, while probabilistic query mining retains the basic algorithms of standard query mining and merely uses probabilistic inference to score patterns obtained from the refinement operator.

### 8.3.1 Probabilistic Explanation Based Learning

Probabilistic explanation based learning (PEBL) differs from EBL as defined in Task 8.2 in two ways: it uses a probabilistic Prolog program instead of a Prolog program, and the explanation to be returned is based on the most likely proof of the example. Using PEBL for explanation learning as defined in Task 8.1 thus restricts the explanation language to conjunctions of literals with operational predicates, and employs the explanation probability of the underlying proof as scoring function.

Computing the most likely proof is one of the basic inference tasks in ProbLog, solved in Algorithm 4.4 by integrating SLD-resolution with iterative deepening search based on the probabilities of derivations. To use ProbLog for PEBL, we extend this algorithm to simultaneously generate the explanation as described for EBL in Section 8.2.1, maintaining the tight integration of probabilistic and logical inference. We next discuss two important aspects of combining ProbLog and EBL, namely the interaction between uncertainty and operationality, and the relationship between learned explanations and the original background theory.

In EBL, literals with non-operational predicates in the abstracted explanation are resolved against the same clause as the corresponding literal in the concrete proof of the example. For predicates defined in terms of ground probabilistic facts, this would introduce example-specific groundings into the explanation, which is undesirable in general. Simply dropping such a literal would not only cause the probability of the training example being covered by the resulting explanation to be higher than its original explanation probability, but could also lose unification information. To avoid these complications, we require probabilistic facts to either be operational or else to only occur below operational predicates in the proof tree, that is, in the definition of operational predicates in the background knowledge.

Explanations constructed in EBL follow deductively from the background theory. In ProbLog, the logical consequences of the background knowledge depend on the interpretation of the probabilistic facts. The probability of the most likely proof as used in PEBL corresponds to the probability that this concrete instance of the resulting explanation is entailed by the ProbLog program. However, it only provides an upper bound on the probability of the abstracted explanation being entailed, which depends on all other possible groundings as well. The probability

is thus entirely example-specific[2] and merely serves to guide the search for the best explanation given the training example.

Probabilistic explanation based learning as incorporated in ProbLog offers natural solutions to two issues traditionally discussed in the context of explanation based learning [Mitchell et al., 1986; Langley, 1989]. The first one is the *multiple explanation* problem, which is concerned with choosing the proof to be abstracted for examples having multiple proofs. This problem arises in many applications where there are various possible explanations as to why a particular query succeeds. For instance, in our biological link mining domain, a gene can be linked to a particular disease in several ways. The use of a sound probabilistic framework naturally deals with this issue by selecting the *most likely* explanation. The second problem is that of *generalizing from multiple examples*, another issue that received quite some attention in traditional explanation based learning. To realize this in our setting, we modify the best-first search algorithm so that it searches for the most likely abstracted explanation shared by the $n$ examples $e_1, ..., e_n$. Starting from the variabelized atom $e$, we compute $n + 1$ SLD-resolution derivations in parallel. A resolution step resolving an atom for a non-operational predicate in the abstracted proof for $e$ is allowed only if the same resolution step can also be applied to each of the $n$ parallel derivations. Atoms corresponding to operational predicates are – as sketched above – not resolved in the abstracted proof, but it is nevertheless required that for each occurrence of these atoms in the $n$ parallel derivations, there exists a resolution derivation.

**Example 8.12** *Consider again our running example, and assume that we now want to construct a common explanation for* `related(b,e)` *and* `related(a,e)`*. We thus have to simultaneously prove both examples and the variabelized goal* `related(X,Y)`*. This is illustrated in Figure 8.1(c). After resolving all three goals with the first clause for* `related/2`*, we reach the first instance of the operational predicate* `alike/2` *and thus have to prove both* `alike(b,e)` *and* `alike(a,e)`*. As proving* `alike(b,e)` *fails, the last resolution step is rejected and the second clause for* `related/2` *used instead. As both* `alike(b,A)` *and* `alike(a,B)` *can be proven,* `alike(X,Z)` *is added to the explanation, and the procedure continues with the goals* `related(c,e)`*,* `related(c,e)` *and* `related(Z,Y)`*. This succeeds using the base case and adds* `alike(Z,Y)` *to the explanation, resulting in the explanation*

```
exp_related(X,Y) :- alike(X,Z),alike(Z,Y).
```

To reason by analogy, it then suffices to add an explanation generated by PEBL to the input program. Posing queries to the resulting predicate returns tuples

[2]The part related to the general explanation mentioned in [Kimmig et al., 2007] is due to the early formulation of ProbLog, where all groundings of a probabilistic clause were interpreted as a single random event, cf. Footnote 1 in Section 3.1, which made it possible to resolve against such clauses in the abstracted explanation without the complications mentioned above.

analogous to the training examples, which can be ranked according to probability, and hence, analogy.

**Example 8.13** *Using the explanation of Example 8.12 to query for covered instances returns the following answers:*

$0.8 \cdot 0.9 = 0.72$ `exp_related(a,d)`
$0.8 \cdot 0.8 = 0.64$ `exp_related(a,e)`
$0.6 \cdot 0.9 = 0.54$ `exp_related(b,d)`
$0.6 \cdot 0.8 = 0.48$ `exp_related(b,e)`
$0.7 \cdot 0.6 = 0.42$ `exp_related(a,c)`
$0.8 \cdot 0.5 = 0.40$ `exp_related(c,d)`

### 8.3.2 Probabilistic Local Query Mining

Using local query mining for explanation learning as defined in Task 8.1 restricts the explanation language to the set of conjunctive queries specified by the language bias. To obtain the required probabilistic scoring function, we change the selection predicate $\phi$ or correlation measure $\psi$ to work with probabilistic databases, that is, to employ a probabilistic membership function. In non-probabilistic frequent query mining, every tuple in the relation *id* either satisfies the query or not. For instance, for a conjunctive query $q$ and a 0/1 membership function $M^{\mathcal{D}}(q(t))$, we can explicitly write the counting function underlying frequency as a sum:

$$\mathsf{COUNT}^{\mathcal{D}}(q) = \sum_{t \in id} M^{\mathcal{D}}(q(t)) \tag{8.6}$$

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

To apply the algorithms sketched in Section 8.2.2 with a probabilistic database $\mathcal{D}$, it suffices to replace the deterministic membership function $M^{\mathcal{D}}(q(t))$ with a probabilistic variant. Possible choices for such a probabilistic membership function $P^{\mathcal{D}}(q(t))$ include the success probability $P_s^{\mathcal{D}}(q(t))$ or the explanation probability $P_x^{\mathcal{D}}(q(t))$ as introduced for ProbLog in Equations (3.16) and (3.15) on page 35. Note that using such query probabilities as probabilistic membership function is anti-monotonic, that is, if $q_1 \preceq q_2$ then $P^{\mathcal{D}}(q_1(t)) \geq P^{\mathcal{D}}(q_2(t))$. A natural choice of selection predicate $\phi$ is the combination of a minimum threshold with an aggregated probabilistic membership function

$$agg^{\mathcal{D}}(q) = \mathbf{AGG}_{t \in id}\ P^{\mathcal{D}}(q(t)). \tag{8.7}$$

Here, **AGG** denotes an aggregate function such as $\sum$, min, max or $\prod$, which is to be taken over all tuples $t$ in the relation *id*. Choosing $\sum$ with a deterministic

membership relation corresponds to the traditional frequency function, whereas $\prod$ computes a kind of *likelihood* of the data. Note that whenever the membership function $P$ is anti-monotone, selection predicates of the form $agg^{\mathcal{D}}(q) > c$ are anti-monotonic with regard to $OI$-subsumption, which is crucial to enable pruning.

When working with both positive and negative examples, the main focus lies on finding queries with a high aggregated score on the positives and a low aggregated score on the negatives. Note that using unclassified instances $id$ corresponds to the special case where $id^+ = id$ and $id^- = \emptyset$. In the following, we will therefore consider instances of the selection function (8.7) for the case of classified examples $id^+$ and $id^-$ only. We use the following three functions:

$$pf^{\mathcal{D}}(q) = \sum_{t \in id^+} P^{\mathcal{D}}(q(t)) - \sum_{t \in id^-} P^{\mathcal{D}}(q(t)) \tag{8.8}$$

$$LL^{\mathcal{D}}(q) = \prod_{t \in id^+} P^{\mathcal{D}}(q(t)) \cdot \prod_{t \in id^-} \left(1 - P^{\mathcal{D}}(q(t))\right) \tag{8.9}$$

$$LL^{\mathcal{D}}_n(q) = \prod_{t \in id^+_n} P^{\mathcal{D}}(q(t)) \cdot \prod_{t \in id^-} \left(1 - P^{\mathcal{D}}(q(t))\right) \tag{8.10}$$

Here, $id^+_n$ contains the $n$ highest scoring tuples in $id^+$. We obtain an upper bound on each of these functions by ignoring the scores of negative examples, i.e. the aggregation over $id^-$. We will omit $\mathcal{D}$ if it is clear from the context.

Choosing sum as aggregation function results in a *probabilistic frequency* $pf$ (8.8) also employed by [Chui et al., 2007] in the context of item-set mining.

**Example 8.14** *Reconsider Example 8.11, where we mine for top $k = 3$ explanations using the probabilistic frequency $pf$ with $P^{\mathcal{D}}(q(t)) = P^{\mathcal{D}}_x(q(t))$. Table 8.2 shows the scores obtained during the first level. As an example, for query 1 we obtain $P(a)+P(c)-(P(d)+P(e)) = 0.8+0.9-(0+0.5)$. The threshold for the three best queries at this point is* 0.8*, therefore, queries 1, 4, 5, 6, 7 and 8 will be refined, whereas the remaining ones are pruned.*

Using product, on the other hand, defines a kind of *likelihood* $LL$ (8.9), as also used in ProbLog theory compression, cf. Equation (6.2) (p. 117).

**Example 8.15** *Consider frequent pattern mining with $LL^{\mathcal{D}}(q) > 0$ for our example database with $id = id^+ = \{a, c\}$. The only query that will be refined after the first level is* `q(X) :- alike(X,Y)`*, as all other queries have probability zero on at least one of the elements of $id$.*

As using the product in combination with a non-zero threshold implies that *all* positive examples must be covered with non-zero probability, which is often overly

| | query | $\sum_{id^+}$ | $\sum_{id^-}$ | $pf$ |
|---|---|---|---|---|
| 1 | `q(X) :- alike(X,Y)` | **1.7** | 0.5 | **1.2** |
| 2 | `q(X) :- alike(X,a)` | 0 | 0 | 0 |
| 3 | `q(X) :- alike(X,b)` | 0.7 | 0 | 0.7 |
| 4 | `q(X) :- alike(X,c)` | **0.8** | 0 | **0.8** |
| 5 | `q(X) :- alike(X,d)` | **0.9** | 0.5 | 0.4 |
| 6 | `q(X) :- alike(X,e)` | **0.8** | 0 | **0.8** |
| 7 | `q(X) :- alike(Y,X)` | **0.8** | 1.7 | -0.9 |
| 8 | `q(X) :- alike(a,X)` | **0.8** | 0 | **0.8** |
| 9 | `q(X) :- alike(b,X)` | 0.6 | 0 | 0.6 |
| 10 | `q(X) :- alike(c,X)` | 0 | 1.7 | -1.7 |
| 11 | `q(X) :- alike(d,X)` | 0 | 0 | 0 |
| 12 | `q(X) :- alike(e,X)` | 0 | 0.5 | -0.5 |

Table 8.2: Aggregates on $id^+$ and $id^-$ and probabilistic frequency obtained during the first level of correlated query mining with $k = 3$ in Example 8.14. The current minimal score for best explanations is 0.8, i.e. only queries with $pf \geq 0.8$ or $\sum_{id^+} \geq 0.8$ will be refined on the next level.

| query | $LL_1^{\mathcal{D}}(q)$ |
|---|---|
| `q(X) :- alike(X,Y)` | 0.9 |
| `q(X) :- alike(X,b)` | 0.7 |
| `q(X) :- alike(X,c)` | 0.8 |
| `q(X) :- alike(X,d)` | 0.9 |
| `q(X) :- alike(X,e)` | 0.8 |
| `q(X) :- alike(Y,X)` | 0.8 |
| `q(X) :- alike(a,X)` | 0.8 |
| `q(X) :- alike(b,X)` | 0.6 |

Table 8.3: Queries refined after the first level of query mining using the modified likelihood score $LL_1$ in Example 8.16.

harsh, we also introduce a softened version $LL_n$ (8.10) of the likelihood, where $n < |id^+|$ examples have to be covered with non-zero probability. This is achieved by restricting the set of tuples in the product to the $n$ highest scoring tuples in $id^+$, thus integrating a deterministic (anti-monotonic) selection predicate into the probabilistic one.

**Example 8.16** *Using $LL_1^{\mathcal{D}}(q) > 0$ in Example 8.15, the eight queries shown in Table 8.3 cover at least one example and will thus be refined after the first level.*

## 8.4 Implementation

The implementation of PEBL in Yap-5.1.2 used in the experiments of Section 8.5.1 combines a classical EBL meta-interpreter with an iterative-deepening meta-interpreter for ProbLog as introduced in [De Raedt et al., 2007b].

Our implementation of correlated query mining is built upon and extends the public version of ProbLog as introduced in Chapter 4 as well as the public domain implementation of the (non-probabilistic) frequent query mining system *c-armr* of De Raedt and Ramon [2004].

As in *c-armr*, the language bias can be defined using type and mode restrictions as well as background knowledge. This reduces the number of queries generated by taking advantage of general knowledge about the domain of interest. As each query is evaluated on all training examples in turn, we prune query evaluation as soon as the current upper bound of its aggregated score falls below the threshold, where we include maximum estimates for positive examples not processed yet. The user can define an application specific query reordering function aimed at more efficient logical inference.

For correlated query mining we further modified *c-armr*. First, to deal with positive and negative examples, we keep track of queries that are infrequent, but cannot be pruned as their upper bound is still promising. Second, we modified the search strategy to dynamically adapt the threshold to the score of the $k$th best query whenever the set of $k$ best queries found so far changes, thereby allowing for more pruning.

## 8.5 Experiments

We experimentally evaluate our two approaches to explanation learning in the context of the weighted biological database of Sevon et al. [2006], cf. also Section 2.5, where we use a probabilistic network encoding with one probabilistic relation `edge/3` describing edges in terms of two nodes and a label (e.g. `contains`), and a deterministic relation `node/2` assigning a label (e.g. `'Gene'`) to each node, see Figure 8.3 for some illustration in the context of explanations.

While both methods ultimately aim at the same task of finding analogous examples, the experimental evaluation is adapted to the specific setting the respective method tackles. For PEBL, we therefore focus on examples consisting of multiple nodes in the network, define connections between those in the background knowledge, and evaluate resulting explanations both inside their domain of origin and across domains. For probabilistic query mining, on the other hand, we choose single nodes as examples, connected labeled graphs starting from those nodes as explanation

```
e_path(A,B)  :-  node(A,gene), edge(A,C,belongs_to),
    node(C,homologgroup), edge(B,C,refers_to), node(B,phenotype),
    nodes_distinct([B,C,A]).
e_path(A,B)  :-  node(A,gene), edge(A,C,codes_for), node(C,protein),
    edge(D,C,subsumes), node(D,protein), edge(D,E,interacts_with),
    node(E,protein), edge(B,E,refers_to), node(B,phenotype),
    nodes_distinct([B,E,D,C,A]).
e_path(A,B)  :-  node(A,gene), edge(A,C,participates_in),
    node(C,pathway), edge(D,C,participates_in), node(D,gene),
    edge(D,E,codes_for), node(E,protein), edge(B,E,refers_to),
    node(B,phenotype), nodes_distinct([B,E,D,C,A]).
e_path(A,B)  :-  node(A,gene), edge(A,C,is_found_in),
    node(C,cellularcomponent), edge(D,C,is_found_in),
    node(D,protein), edge(B,D,refers_to),
    node(B,phenotype), nodes_distinct([B,D,C,A]).
```

Figure 8.3: Some explanations for `path(A,B)`, connecting gene `A` to phenotype `B`.

language, and focus evaluation on one domain. Furthermore, as probabilistic query mining offers several choices concerning correlation measures and is computationally demanding, we also study performance in terms of runtimes and scalability for this method.

### 8.5.1 Probabilistic EBL

As an example problem to be studied using PEBL, we looked at connections between disease genes and the corresponding phenotype for Alzheimer disease (resp. asthma). Since the key motivation for employing probabilistic explanation based learning is to be able to reason by analogy or to find analogous examples, we set up experiments to answer the following questions:

**Q1** Does PEBL produce meaningful examples when reasoning by analogy?

**Q2** Can we find common explanations?

**Q3** Can PEBL's explanations induced on one domain (say Alzheimer disease) be transferred to another one (say asthma)?

To answer those questions, we use the ALZHEIMER and ASTHMA subgraphs as described in Appendix A. We modify the predicate `path/2` as given by definition (4.14), page 81, to use `edge/3` and to additionally query `node/2` for each newly visited node. We define predicates related to node and edge types as

operational. While the definition of path ensures acyclicity during construction of explanations, this requirement is not captured by the resulting sequence of edge and node atoms. We therefore terminate each proof with an additional operational predicate `nodes_distinct/1` whose argument is the list of visited nodes to ensure acyclic paths when reasoning by analogy. We start by studying example explanations for `path(A,B)` obtained from the graphs, where `A` is a gene and `B` a phenotype (Figure 8.3). These explanations are all semantically meaningful. For instance, the first one indicates that gene A is related to phenotype B if A belongs to a group of homologous (i.e., evolutionarily related) genes that relate to B. The three other explanations are based on interaction of proteins: either an explicit one, by participation in the same pathway, or by being found in the same cellular component. This last discovery suggests that a clause to describe different kinds of possible interactions would be a useful feature in the logical theory. It thus seems that PEBL can produce useful explanations and can help the user discover and synthesize new information, which answers **Q1** positively.

To further study the questions more objectively, we consider a slightly artificial setting. We define a target predicate `connect/3` as

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

This predicate succeeds if three nodes are connected to each other. While each of the three paths needs to be acyclic, nodes and edges can arbitrarily be shared amongst them. We use graphs ALZHEIMER1 and ASTHMA1 with examples where `Z` is a phenotype and `X` and `Y` are genes.

Resulting explanations are used to classify ordered triplets `(G1,G2,P)` of nodes, where `G1` and `G2` are different genes and `P` is a phenotype[3]. We call such a triplet positive with respect to the given network if both genes are annotated with the graph's disease and `P` is the corresponding phenotype, and negative otherwise. Thus for ALZHEIMER1, there are $14 \cdot 13 \cdot 1 = 182$ positive and $29 \cdot 28 \cdot 3 - 182 = 2254$ negative triplets. Table 8.4 summarizes these statistics for all graphs.

We use `connect(G1,G2,P)` for positive triplets as training examples. As `connect` is symmetric in the first two arguments, we only consider one ordering per pair of genes, which yields in total $14 \cdot 13/2 = 91$ training examples for ALZHEIMER1 (resp. 21 for ASTHMA1). The aim is to construct explanations for connections between the nodes of positive triplets, and use those to obtain for each test graph a ranked list of triplets covered by the explanation. To do so, we first compute the most likely explanation for each individual training example $e$, and then rank all instances covered by the resulting explanation according to their explanation probability. Table 8.5 summarizes classification accuracies obtained using those rankings and the classification criterion on triplets as described above. Values are averaged over all training examples. On graphs describing the same disease as

[3]The background knowledge ensures that this type information is part of any explanation.

| | depth | nodes | edges | ag | ng | pt | pos | neg |
|---|---|---|---|---|---|---|---|---|
| ALZHEIMER1 | 4 | 122 | 259 | 14 | 15 | 3 | 182 | 2254 |
| ALZHEIMER2 | 5 | 658 | 3544 | 17 | 20 | 4 | 272 | 5056 |
| ALZHEIMER3 | 4 | 351 | 774 | 72 | 33 | 3 | 5112 | 27648 |
| ALZHEIMER4 | 5 | 3364 | 17666 | 130 | 55 | 6 | 16770 | 187470 |
| ASTHMA1 | 4 | 127 | 241 | 7 | 12 | 2 | 42 | 642 |
| ASTHMA2 | 5 | 381 | 787 | 11 | 12 | 2 | 110 | 902 |

Table 8.4: Graph characteristics: search depth used during graph extraction, numbers of nodes and edges, number of genes annotated resp. not annotated with the corresponding disease and number of phenotypes, number of positive and negative examples for connecting two genes and a phenotype.

| | ALZHEIMER1 | | | | | |
|---|---|---|---|---|---|---|
| | pos(1) | pos(3) | pos(5) | pos_n | pos_a | prec |
| ALZHEIMER1 | 0.95 | 2.53 | 3.95 | 6.91 | 16.82 | 0.46 |
| ALZHEIMER2 | 0.84 | 2.24 | 3.60 | 7.37 | 18.65 | 0.42 |
| ALZHEIMER3 | 0.99 | 2.64 | 4.09 | 23.20 | 126.09 | 0.48 |
| ALZHEIMER4 | 0.84 | 2.23 | 3.58 | 7.37 | 18.80 | 0.42 |
| ASTHMA1 | 0.09 | 0.26 | 0.44 | 2.07 | 2.07 | 0.02 |
| ASTHMA2 | 0.08 | 0.23 | 0.38 | 2.00 | 2.00 | 0.01 |

| | ASTHMA1 | | | | | |
|---|---|---|---|---|---|---|
| | pos(1) | pos(3) | pos(5) | pos_n | pos_a | prec |
| ALZHEIMER1 | 1.00 | 3.00 | 4.86 | 6.86 | 10.57 | 0.23 |
| ALZHEIMER2 | 0.86 | 2.86 | 4.71 | 6.86 | 14.56 | 0.22 |
| ALZHEIMER3 | 1.00 | 2.71 | 4.14 | 6.86 | 28.00 | 0.24 |
| ALZHEIMER4 | 0.86 | 2.29 | 3.43 | 5.14 | 28.00 | 0.15 |
| ASTHMA1 | 1.00 | 3.00 | 4.86 | 17.14 | 17.14 | 0.34 |
| ASTHMA2 | 0.86 | 2.57 | 4.29 | 16.57 | 16.57 | 0.20 |

Table 8.5: Averaged results over all examples learned on ALZHEIMER1 (top) resp. ASTHMA1 (bottom) and evaluated on 6 different graphs (lines ALZHEIMER1–4, ASTHMA1–2): number of positives among the first $k$ answers (pos($k$)), number of positives returned before the first negative (pos_n), absolute number of positives among examples with non-zero probability (pos_a), and precision w.r.t. all examples with non-zero probability (prec).

the training graph, the top $k$ instances for $k = 1, 3, 5$ are mostly positive, which again gives a positive answer to **Q1**. We obtained 26 different explanations from ALZHEIMER1 and 3 different explanations from ASTHMA1. Most explanations have been learned on at least two examples, and the biggest set of examples which shared the most likely explanation contains 12 examples. Figure 8.4 shows an example explanation found both on ALZHEIMER1 and on ASTHMA1. This indicates

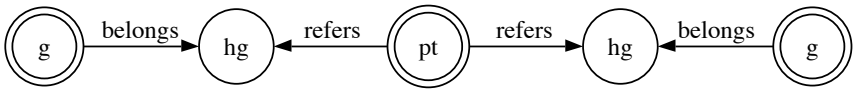



Figure 8.4: One explanation for `connect(G1,G2,P)`, where double circles mark answer variables, and node types used are gene (g), phenotype (pt) and homologgroup (hg).

a positive answer to question **Q2**, discovery of common explanations. The answer to **Q3**, if explanations can be transferred, is less conclusive: while transfer from asthma to Alzheimer disease achieves good results, the other direction is a lot worse. However, one has to take into account that 23 of the 26 explanations learned on Alzheimer disease do not return any example at all on asthma graphs, one only returns negative instances and the remaining 2 carry over very well, returning 30 resp. 16 positive instances before returning negative ones. At the same time, two of the three explanations learned on asthma were also learned on Alzheimer disease, which might explain their good performance there.

### 8.5.2 Probabilistic Query Mining

To evaluate our work on probabilistic query mining, we again report on experiments in the context of the probabilistic biological database. Even though the database is very large, at any point in time, a biologist will typically focus on a particular phenomenon for which only a limited number of nodes is known to be relevant. As a test-case to be studied, we therefore use the 142 genes known to be related to Alzheimer disease contained in our database. We set up experiments to answer the following questions about correlated query mining:

**Q4** How do $P_s$ and $P_x$ differ in performance?

**Q5** Can the top queries discriminate unseen positive and negative examples?

**Q6** Does the correlated query miner prune effectively?

**Q7** Can the correlated query miner use the full network?

To answer these questions, we used three graphs: the full network BIOMINE of roughly 6 million edges as well as ALZHEIMER2 and ALZHEIMER4, see Appendix A for further details. The language bias employed allows adding literals of the form `edge(X,Y,e)`, `edge(X,Y)` (as a shortcut of `edge(X,Y,_)`) and `node(X,n)` where `X` and `Y` are variables of type node name, `X` already appears in the query (thereby ensuring linkage), and `e` and `n` are constants denoting an edge and a node label

| | $LL$ | $LL_n$ | $pf$ |
|---|---|---|---|
| $P_s$ | .72/.45/.33 | .27/.02/.00 | .13/.03/.00 |
| $P_x$ | 1/1/.45 | 1/1/1 | 1/1/1 |

Table 8.6: Fraction of cases where mining for $k = 1/5/20$ best queries successfully terminated within 30 minutes.

appearing in the graph respectively. Note that in contrast to the running example used for illustration, we do not allow node names as constants in the query language here, as this would entail prohibitively many possible refinements for each query. The bias further states that labels are mutually exclusive, that `edge(X,Y,e)` implies `edge(X,Y)`, and how to invert labels to use edges backwards. This ensures that edges in queries map to database entries independently of direction. We use a query reordering function greedily moving literals containing constants to the left.

Positive and negative training examples $p_j^i$ and $n_j^i$ are gene nodes annotated resp. not annotated with AD randomly picked from ALZHEIMER2. We create ten sets each containing ten example of each class

$$S^i = \{p_1^i, n_1^i, \ldots, p_{10}^i, n_{10}^i\} \tag{8.11}$$

For each such $S^i$, we use the first $j$ examples of each class to obtain a total of 100 datasets of varying size

$$D_j^i = \{p_1^i, n_1^i, \ldots, p_j^i, n_j^i\} \tag{8.12}$$

To avoid one trivial refinement step, `q(X):-node(X,'Gene')` is used as most general query.

As probabilistic membership function $P(q(t))$ we employ either the explanation probability $P_x$ or the lower bound of the exact probability $P_s$ obtained by bounded approximation with interval width $\delta = 0.1$ and a time limit of 60 sec for the evaluation of each individual bound. As aggregation functions to obtain probabilistic selection predicates, we use the likelihood $LL$ (8.9), the probabilistic frequency $pf$ (8.8) and the softened likelihood $LL_n$ (8.10) with $n = \lceil m/2 \rceil$ for $m$ examples of each class, indicating the probability function used by superscripts where needed. All experiments are performed on 2.4 GHz 4GB machines running Linux.

To answer questions **Q4** and **Q5**, we mine on ALZHEIMER2 for $k = 1, 5, 20$ with a time limit of 30 minutes per run, using the 60 datasets $D_j^i$ with at least 5 examples of each class, that is, with $5 \leq j \leq 10$. Table 8.6 illustrates the performance in terms of the fraction of successful runs. In the case of $P_x$, the time limit is only reached for $k = 20$ with $LL$, i.e. when a higher number of queries covering all positive examples is desired, whereas $P_s$ crosses the limit frequently in all settings. These results clearly suggest that $P_x$ is more favorable in terms of runtimes. To compare the two choices of probabilistic membership function $P$ in terms of their

| | $LL^s$ | $LL^s_n$ | $pf^s$ | $LL^x$ | $LL^x_n$ | $pf^x$ |
|---|---|---|---|---|---|---|
| precision | 0.76 | 0.95 | 0.92 | 0.77 | 0.93 | 0.93 |
| recall | 0.94 | 0.79 | 0.81 | 1.00 | 0.85 | 0.86 |
| F-measure | 0.84 | 0.85 | 0.86 | 0.87 | 0.88 | 0.88 |

Table 8.7: Using query obtained with $k = 1$ to reason by analogy: Overall precision, recall and F-measure, averaged over cases with mining time at most 30 minutes.

| | $LL^s$ | $LL^s_n$ | $pf^s$ | $LL^x$ | $LL^x_n$ | $pf^x$ |
|---|---|---|---|---|---|---|
| (a) | 1/.99/.95 | 1/1/1 | 1/1/1 | 1/1/1 | 1/1/1 | 1/1/1 |
| (b) | 0.18 | 0.67 | 0.33 | 0.93 | 0.83 | 0.86 |

Table 8.8: Using query obtained with $k = 1$ to reason by analogy: (a) Precision among the first $n = 1/10/20$ ranked examples, (b) fraction of positives ranked before the first negative, averaged over cases with mining time at most 30 minutes.

results, we use the highest scored query (note that scores are independent of $k$) to retrieve covered examples from the larger graph ALZHEIMER4, and rank those using the corresponding $P$. In case of equally likely queries, we choose the most specific one. Note that due to the form of the most general query employed in mining, this will return nodes of type gene only. Training examples are excluded from the rankings. The fraction of annotated genes and thus positive examples among the possible answers is 0.76.

We calculate overall precision (percentage of ranked genes that are positive), recall (percentage of positive genes included in the ranking), and F-measure ($2 \cdot prec \cdot rec/(prec + rec)$) for all rankings. Table 8.7 gives the averages over all successful cases. Using $LL$ results in high recall and low precision (close to the fraction of positives among unseen genes), as queries covering all positive training examples are very general and therefore often cover all negative examples as well. Reducing the number of positives included in the score, i.e. using $LL_n$, and using probabilistic frequency both increase precision at the expense of lower recall. This tendency is confirmed by the F-measure, which balances precision and recall. Again, the explanation probability achieves better results. Combining resource requirements and results, the answer to question **Q4** is thus that using the explanation probability is more favorable.

To study the predictive performance of best queries in more detail, we examine the top regions of the rankings used above. Table 8.8 shows the precision among the top ranked examples. All queries return several positive examples first, only in few cases, negatives occur within the first twenty positions. Furthermore, especially for the case of $P_x$, the queries mined return a large fraction of all positive examples before the first negative one. Together, these results show that the best queries are indeed able to distinguish unseen positive and negative examples, thus answering

| k | 1 | 5 | 10 | 20 | 50 |
|---|---|---|---|---|---|
| (a) | 225 | 626 | 983 | 1814 | 4057 |
| (b) | 1.25 | 6.44 | 32.46 | 98.37 | 1523.26 |

Table 8.9: Mining on ALZHEIMER4: (a) average number of queries tested for datasets of size 10, (b) average runtimes (sec).

**Q5** affirmatively.

Comparing the different aggregation functions in terms of both runtime and results confirms that both $LL_n$ and $pf$ clearly outperform $LL$. As probabilistic frequency has the advantage of not requiring an extra parameter, we restrict ourselves to probabilistic frequency using $P_x$ in the following.

To compare the number of queries examined for various $k$ and thus answer **Q6**, we mine on the larger graph ALZHEIMER4 with 17666 edges, with $pf^x$ as selection predicate. Table 8.9 shows the number of queries tested and runtimes for various values of $k$, averaged over all datasets of size ten. The query language $\mathcal{L}_{\text{ALZHEIMER4}}$ already contains roughly 50K elements of length at most 3. The size of the search space explored nicely scales with $k$, focusing on very small fractions of the entire search space, thereby answering **Q6** affirmatively.

Finally, as both the tasks of query mining and probabilistic reasoning in relational databases are computationally hard and are combined here, we also tested the algorithm with $pf^x$ on the entire network BIOMINE with around 6M edges, using the ten largest datasets and a time limit of one hour. For $k = 1$, runtimes vary from 626 to 1578 seconds, with an average of 865. For $k = 2$, the seven runs finishing within the time limit take between 1701 and 3145 seconds, with an average of 2610. Runtimes for higher $k$ exceed the limit. These results indicate that although probabilistic relational query mining is computationally challenging, it is in principle possible to run the algorithm on large scale networks for small values of $k$. Thus the answer to **Q7** is positive as well, although improving the efficiency of the probabilistic reasoning engine would help to further increase scalability.

## 8.6 Related Work

The work presented in this chapter builds upon the existing work in both probabilistic logics and explanation based learning or query mining, respectively. It is related to work on analogical, similarity and case based reasoning, providing a notion of analogy that is based on logical coverage as well as likelihood.

A first step towards combining EBL and probabilistic frameworks is done in [DeJong, 2006], where logical and empirical evidence are combined in explanation based

learning to get explanations with a certain confidence. PEBL as presented here is a simple extension of the formalisations of explanation based learning within Prolog due to [Hirsh, 1987; Van Harmelen and Bundy, 1988] with probabilistic inference. From a probabilistic logic point of view, it extends the work of Poole [1993b] in that it finds *abstracted* explanations. We have argued that using probabilities to score explanations provides a natural solution to the multiple explanation problem whereas the use of a resolution based proof procedure allows one to naturally deal with multiple examples and identify common explanations.

The probabilistic correlated query mining system introduced here extends existing multi-relational data mining systems such as Warmr [Dehaspe and Toivonen, 1999] and c-armr [De Raedt and Ramon, 2004] to deal with probabilistic data. Research on mining probabilistic data has so far been focused on frequent pattern mining with two different notions of frequency. *Minimum expected support*, which corresponds to the probabilistic frequency of Equation (8.8), has been used for frequent item-set mining [Chui et al., 2007] and for frequent subgraph mining in probabilistic graph databases [Zou et al., 2009]. *Minimum frequentness probability*, which considers a pattern to be frequent if its probability of being frequent in a randomly sampled instance of the database exceeds a given threshold, has been used to find frequent items [Zhang et al., 2008], item-sets [Bernecker et al., 2009] and subgraphs [Zou et al., 2010]. However, these approaches consider neither arbitrary relational data nor correlated pattern mining. As a consequence, their algorithms are tailored to a specific type of queries, whereas relational query mining as considered here has to deal with a much broader class of possible queries that are more complex to evaluate.

## 8.7 Conclusions

This chapter has studied the use of relational explanations to reason by analogy in a probabilistic context. Using such explanations as additional background knowledge clauses, analogous examples can be ranked by probability. Depending on the available domain knowledge, explanations can either be constructed deductively based on both the database and a domain theory, or inductively based on the database only. We have formalized the task of explanation learning and introduced two concrete instantiations addressing the deductive and inductive setting, respectively. The techniques presented have been realized in ProbLog, but could easily be adapted towards other probabilistic relational frameworks.

As a deductive approach, we have introduced probabilistic explanation based learning, which applies principles of explanation based learning to probabilistic logic representations. It learns explanations by abstracting the most likely proof of a given example with respect to the background knowledge.

As an inductive approach suited for situations where no structural domain knowledge is available, we have extended the frequent pattern mining paradigm towards a probabilistic setting. We have introduced a correlated query mining algorithm together with various scoring functions aggregating probabilities.

The resulting general techniques have been evaluated on challenging biological network mining tasks, showing that the resulting explanations can be used to retrieve analogous instances with high accuracy. A detailed experimental comparison of the two approaches in the context of this network as well as on different datasets is an important direction for future work.

# Conclusions Part III

In this part of the thesis, ProbLog has been used as a framework for reasoning by analogy. More specifically, we have formalized analogy in terms of relational explanations that, when added to a ProbLog program, allow one to rank analogous examples by probability. As obtaining such a ranking corresponds to basic probabilistic inference in ProbLog, we have focused on the task of explanation learning. Depending on the available domain knowledge, explanations can either be constructed deductively based on both the database and a domain theory, or inductively based on the database only. We have introduced concrete explanation learning techniques for both cases. Both methods extend well-known relational learning techniques to probabilistic logics and thus also illustrate the use of ProbLog as a framework for lifting traditional ILP tasks to the probabilistic setting. While probabilistic explanation based learning relies on background knowledge to identify the most likely proof of a given example and to construct an abstracted explanation from this proof, probabilistic query mining is targeted towards situations where no domain knowledge is available. We have focused on correlated query mining here, introducing scoring functions that favour high probabilities for positive examples and low probabilities for negative examples. The results of both methods have been used for reasoning by analogy in the context of the Biomine network. A detailed experimental comparison of the two approaches in the context of this network as well as on different datasets is an important direction for future work.

# Finale

# Chapter 9

# Summary and Future Work

We conclude the thesis by providing a summary and a perspective on future work.

## Thesis Summary

Probabilistic logic learning, also known as statistical relational learning [Getoor and Taskar, 2007; De Raedt et al., 2008a], is concerned with the combination of relational languages, machine learning and statistical methods, which all are key ingredients for flexible reasoning and problem solving systems. The field has contributed a variety of approaches, combining different relational and statistical components and addressing various types of inference and learning tasks. One stream of work extends logical languages such as definite clause logic with independent probabilistic alternatives, most often sets of ground facts at most one of which is probabilistically chosen to be true. Examples of such languages include probabilistic logic programs [Dantsin, 1991], the Independent Choice Logic [Poole, 2000] and PRISM [Sato and Kameya, 2001]. The common theoretical basis of these languages is the distribution semantics [Sato, 1995]. While such languages are remarkably expressive despite their independence assumption on basic probabilistic events, available implementations often restrict this expressivity by making additional assumptions to allow for more efficient inference, or do not scale very well. These limitations can hinder or even prevent their application to concrete reasoning and learning tasks.

In this thesis, we have introduced ProbLog, a simple extension of the logic programming language Prolog with probabilistic facts representing independent random variables. While ProbLog shares its theoretical foundations, the distribution semantics, with the probabilistic logic languages mentioned above, its

implementation has been the first to allow for scalable inference in networks of probabilistic links. This is due to the use of binary decision diagrams [Bryant, 1986], which makes it possible to base probabilistic inference on proofs or explanations even if those are not mutually exclusive in terms of the possible worlds they cover. Since their first use in ProbLog, binary decision diagrams are getting increasingly popular in the fields of probabilistic logic learning and probabilistic databases, cf. for instance [Riguzzi, 2007; Ishihata et al., 2008; Olteanu and Huang, 2008; Thon et al., 2008; Riguzzi, 2009].

Given its simplicity as well as the scalable inference algorithms, a second important application domain for ProbLog is the lifting of tasks traditionally studied in relational learning and inductive logic programming [Muggleton and De Raedt, 1994; De Raedt, 2008] to probabilistic logic languages. This has been demonstrated in this thesis for theory compression, which is a restricted form of theory revision [Wrobel et al., 1996], explanation based learning [Mitchell et al., 1986], and query mining [Dehaspe et al., 1998; De Raedt and Ramon, 2004]. We have also applied the latter two techniques to learn explanations which have then be used to reason by analogy, that is, to identify analogous examples that are covered with high probability.

ProbLog's probabilistic facts can also be viewed as a probabilistic database with independent tuples [Suciu, 2008]. In contrast to most other SRL languages, such databases do not define a distribution over derived queries, but over interpretations that either entail a given query or not. We have therefore introduced a novel setting for parameter learning in probabilistic databases that integrates learning from entailment and learning from proofs, and a corresponding gradient method for parameter learning in ProbLog.

Throughout the thesis, we have experimentally validated our approaches in the context of the Biomine network of Sevon et al. [2006], a network of a few million links that probabilistically integrates information from public databases in the domain of biology.

Besides its background in probabilistic logic learning, ProbLog also has strong roots in logic programming. As a probabilistic logic programming language, it is part of the recently emerging discipline of probabilistic programming, which extends programming languages of various paradigms with stochastic elements. Further examples of probabilistic programming languages include IBAL [Pfeffer, 2001], Church [Goodman et al., 2008] and Figaro [Pfeffer, 2009].

Finally, while most of the thesis is concerned with ProbLog, we have also introduced $\omega$ProbLog, a generalization of the modeling and inference principles underlying ProbLog where probabilities are replaced by weights defined in terms of commutative semirings. As a weighted logic programming language, it is closely related to the semiring-weighted dynamic programs of the Dyna system [Eisner et al., 2005]. While there is some overlap with respect to definable weights, each of these

formalisms also covers cases that cannot be represented in the other one. In terms of inference techniques, $\omega$ProbLog is related to the compilation of semiring valuation algebras into Boolean formulae [Pouly et al., 2007; Wachter et al., 2007], though the underlying inference problem is different.

To summarize, the key contributions of this thesis are

- the probabilistic programming language ProbLog and its implementation,[1] which can also serve as a framework for lifting traditional ILP approaches to probabilistic ILP or as a tool for reasoning by analogy,
- a new parameter learning setting for probabilistic databases and a corresponding gradient-based algorithm,
- probabilistic variants of theory compression, explanation based learning and query mining, and
- a generalization of probabilistic inference in ProbLog to semiring weights.

## Future Work

ProbLog as introduced in this thesis is a simple yet powerful probabilistic logic programming language providing a number of different directions for future work.

One of the key application domains of ProbLog is that of transferring relational learning techniques to the probabilistic context, a direction we have only started to explore. The theory compression setting introduced in Chapter 6 could be extended towards a more general version of probabilistic theory revision with a broader choice of revision operators. While an operator that sets fact probabilities to 1 could directly be integrated in the current algorithm, more traditional revision operators manipulating clause structure in the background knowledge would require a new or modified algorithm, as such operators affect the structure of BDDs, which makes their evaluation more involved. Sticking to the task of reducing the size of a ProbLog program, it would also be interesting to investigate algorithms taking the opposite direction of growing the theory instead of shrinking it, which has been found beneficial in the context of probabilistic networks [Hintsanen and Toivonen, 2008]. Another direction worth exploring would be that of grammar induction from proof trees or traces along the lines of [De Raedt et al., 2005] or [Bod, 2009]. In the context of relational retrieval, Lao and Cohen [2010] have recently proposed a combination of so-called path experts, which bears some similarity with the patterns generated by probabilistic explanation based learning or probabilistic

[1] available as part of the stable version of YAP (`http://www.dcc.fc.up.pt/~vsc/Yap/`), see also `http://dtai.cs.kuleuven.be/problog/`

query mining and might thus be another application domain for these techniques. Clearly, these are only a few examples from the broad range of techniques developed in ILP and related fields that could be considered.

A somewhat more abstract direction for future work concerns the analysis of relationships between ProbLog and other approaches both from probabilistic logic learning and from probabilistic programming. The simple probabilistic model used in ProbLog suggests that it might be possible to map many other languages into ProbLog. A first step in this direction has been provided in Section 3.4 for a number of closely related languages, but needs to be complemented with more distant examples. Such mappings – in both directions – would not only provide theoretical insight into the domain, but would also make it possible to transfer techniques developed in different frameworks.

While ProbLog's BDD based inference algorithms are very powerful, this power is not always needed and might even waste resources. Indeed, adapting algorithms to the required level of expressive power has been identified as an important strategy in SRL [Landwehr, 2009]. In the case of ProbLog, special purpose algorithms for cases where the disjoint-sum problem does not occur, such as for instance probabilistic grammars, would be a first step, but would ideally be combined with the existing algorithm into a hybrid method that restricts BDD usage to those subproblems that require one to disjoin explanations. While a first realistic approach would probably start from some type of explicit marking to distinguish these cases, one might also try to develop strategies to automatically recognize situations where BDDs are not needed, or to even transform (parts of) programs to obtain equivalent but disjoint explanations.

Another direction of future work concerns existing generalizations of ProbLog. For $\omega$ProbLog, this first of all includes the implementation of the inference algorithms presented in Chapter 5 and their application to concrete problems, but also a more systematic overview of covered tasks and the development of algorithms that avoid constructing the full explanation DNF where possible. In [Bruynooghe et al., 2010], we have introduced FOProbLog, an extension of first order logic with soft constraints. Similarly to ProbLog, probabilities are attached to a set of basic facts, which then guard arbitrary first order formulae. We defined a semantics for FOProbLog, developed a translation into ProbLog, and reported on initial experience with inference. However, the current inference algorithms need to be improved in order to be able to apply the formalism to larger problems.

In [Kimmig and Costa, 2010], we have started to explore the use of ProbLog for link and node prediction in metabolic networks. Our ProbLog model combines information about a given pathway extracted from the Kyoto Encyclopedia of Genes and Genomes (KEGG) with a probabilistic relation modeling noise on gene-enzyme assignments. Link or node prediction for an unseen test organism is based on the information available for related organisms, where the parameter estimation

method discussed in Chapter 7 is used to associate probabilities to different types of queries. As for most SRL approaches, further exploring this and other applications of ProbLog and its extensions on real-world data, for instance in biology, natural language processing or information retrieval, is an important line of future work both for the further validation of existing methods and for the identification of new settings, tasks and challenges.

# Appendix

# Appendix A

# Biomine Datasets*

This appendix briefly describes the Biomine datasets used in our experiments. Real biological graphs were extracted from NCBI Entrez and some other databases as described in [Sevon et al., 2006]. Since NCBI Entrez alone has tens of millions of objects, a rough but efficient method is used to extract a graph capturing a certain region of the full data collection, typically the subgraph surrounding a number of selected genes. Table A.1 gives an overview of all networks used.

The full BIOMINE network considered here is a snapshot of the Biomine database. The other networks fall into two subgroups. SMALL and MEDIUM use a single `edge/2` relation, whereas the remaining ones distinguish different types of relationships. In the following, we describe the generation of these graphs in some more detail. In all cases, weights are produced as described in [Sevon et al., 2006].

To obtain SMALL and MEDIUM, four random genes related to Alzheimer disease (HGNC ids 620, 582, 983, and 8744) have been selected. MEDIUM was obtained by taking the union of subgraphs of radius 3 from the four genes. For SMALL, radius 2 was used, but only best weighted paths were included in the result. In this case, gene 8744 was not connected to the other genes, and the corresponding component was left out.

For the settings studied in Part III, where we are interested in reasoning by analogy, we need data containing information on node and edge types. The simple graph encoding used for SMALL and MEDIUM is therefore extended to use a predicate `edge/3` whose additional argument holds a type label (such as `belongs_to`, `participates_in`). We further introduce a new predicate mapping node identifiers to node types (such as `gene` or `protein`). We extracted graphs around both Alzheimer disease and asthma from a collection of databases. For each

*We are grateful to Hannu Toivonen and the Biomine team for providing these datasets.

| | Extraction | | Network | | |
|---|---|---|---|---|---|
| | Depth | Genes | Nodes | Edges | Labels |
| BIOMINE | - | - | 936,641 | 5,967,842 | yes |
| SMALL | 2* | 4 | 79 | 144 | no |
| MEDIUM | 3 | 4 | 5,220 | 11,530 | no |
| ALZHEIMER1 | 4 | 17 | 122 | 259 | yes |
| ALZHEIMER2 | 5 | 17 | 658 | 3,544 | yes |
| ALZHEIMER3 | 4 | 142 | 351 | 774 | yes |
| ALZHEIMER4 | 5 | 142 | 3,364 | 17,666 | yes |
| ASTHMA1 | 4 | 17 | 127 | 241 | yes |
| ASTHMA2 | 5 | 17 | 381 | 787 | yes |

Table A.1: Biomine datasets: Search depth and number of seed gene nodes used during network extraction (* SMALL only includes best paths), number of nodes and edges in resulting network, and encoding used (with or without type labels).

disease, we obtained a set of related genes by searching Entrez for human genes with the relevant annotation (AD or asthma); corresponding phenotypes for the diseases are from OMIM. Most of the other information comes from EntrezGene, String, UniProt, HomoloGene, Gene Ontology, and OMIM databases. We did not include articles since they would dominate the graphs otherwise. We used a fixed number of randomly chosen (Alzheimer disease or asthma) genes for graph extraction. Subgraphs were extracted by taking all acyclic paths of no more than length 4 or 5, with a probability of at least 0.01, between any given gene and the corresponding phenotype. Some of the genes did not have any such paths to the phenotype and are thus disconnected from the rest of the graph. Three of the 17 Alzheimer genes used for extraction were not connected to the Alzheimer disease phenotype with a path of length at most 4 and are therefore not present in ALZHEIMER1.

# Bibliography


Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A. I. (1996). Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press.

Bachmair, L., Chen, T., and Ramakrishnan, I. V. (1993). Associative Commutative Discrimination Nets. In Gaudel, M.-C. and Jouannaud, J.-P., editors, *Proceedings of the 4th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT-93)*, volume 668 of *Lecture Notes in Computer Science*, pages 61–74. Springer.

Baral, C., Gelfond, M., and Rushton, N. (2009). Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming*, 9(1):57–144.

Bernecker, T., Kriegel, H.-P., Renz, M., Verhein, F., and Züfle, A. (2009). Probabilistic frequent itemset mining in uncertain databases. In Elder, J. F., Fogelman-Soulié, F., Flach, P. A., and Zaki, M. J., editors, *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-09)*, pages 119–128. ACM.

Bod, R. (2009). From exemplar to grammar: A probabilistic analogy-based model of language learning. *Cognitive Science*, 33:752–793.

Bruynooghe, M., Mantadelis, T., Kimmig, A., Gutmann, B., Vennekens, J., Janssens, G., and De Raedt, L. (2010). ProbLog technology for inference in a probabilistic first order logic. In Coelho, H., Studer, R., and Wooldridge, M., editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI-10)*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 719–724. IOS Press.

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

Charniak, E. (1996). Tree-bank grammars. In *Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference (AAAI-96/IAAI-96), Vol. 2*, pages 1031–1036.

Chavira, M. and Darwiche, A. (2007). Compiling Bayesian networks using variable elimination. In Veloso [2007], pages 2443–2449.

Chen, J., Muggleton, S., and Santos, J. (2008). Learning probabilistic logic models from probabilistic examples. *Machine Learning*, 73(1):55–85.

Chui, C. K., Kao, B., and Hung, E. (2007). Mining frequent itemsets from uncertain data. In Zhou, Z.-H., Li, H., and Yang, Q., editors, *Proceedings of the 11th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD-07)*, volume 4426 of *Lecture Notes in Computer Science*, pages 47–58. Springer.

Cohen, W. W. (1992). Abductive explanation-based learning: A solution to the multiple inconsistent explanation problem. *Machine Learning*, 8:167–219.

Cussens, J. (2000). Stochastic logic programs: Sampling, inference and applications. In Boutilier, C. and Goldszmidt, M., editors, *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI-00)*, pages 115–122. Morgan Kaufmann.

Cussens, J. (2001). Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271.

Dalvi, N. N. and Suciu, D. (2004). Efficient query evaluation on probabilistic databases. In Nascimento, M. A., Özsu, M. T., Kossmann, D., Miller, R. J., Blakeley, J. A., and Schiefer, K. B., editors, *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB-04)*, pages 864–875. Morgan Kaufmann.

Dantsin, E. (1991). Probabilistic logic programs and their semantics. In Voronkov, A., editor, *Proceedings of the First Russian Conference on Logic Programming (RCLP-91)*, volume 592 of *Lecture Notes in Computer Science*, pages 152–164. Springer.

De Raedt, L. (2008). *Logical and Relational Learning*. Cognitive Technologies. Springer.

De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S., editors (2008a). *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*. Springer.

De Raedt, L. and Kersting, K. (2003). Probabilistic Logic Learning. *ACM-SIGKDD Explorations: Special issue on Multi-Relational Data Mining*, 5(1):31–48.

De Raedt, L. and Kersting, K. (2004). Probabilistic inductive logic programming. In Ben-David, S., Case, J., and Maruoka, A., editors, *Proceedings of the 15th International Conference on Algorithmic Learning Theory (ALT-04)*, volume 3244 of *Lecture Notes in Computer Science*, pages 19–36. Springer.

De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., and Toivonen, H. (2006). Revising probabilistic Prolog programs. In *Short Papers of the 16th International Conference on Inductive Logic Programming (ILP-06)*, pages 52–54.

De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., and Toivonen, H. (2007a). Revising probabilistic Prolog programs. In Muggleton, S., Otero, R. P., and Tamaddoni-Nezhad, A., editors, *Revised Selected Papers of the 16th International Conference on Inductive Logic Programming (ILP-06)*, volume 4455 of *Lecture Notes in Computer Science*, pages 30–33. Springer.

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

De Raedt, L., Kersting, K., and Torge, S. (2005). Towards learning stochastic logic programs from proof-banks. In Veloso, M. M. and Kambhampati, S., editors, *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 752–757. AAAI Press / The MIT Press.

De Raedt, L., Kimmig, A., and Toivonen, H. (2007b). ProbLog: A probabilistic Prolog and its application in link discovery. In Veloso [2007], pages 2462–2467.

De Raedt, L. and Ramon, J. (2004). Condensed representations for inductive logic programming. In Dubois, D., Welty, C. A., and Williams, M.-A., editors, *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR-04)*, pages 438–446. AAAI Press.

Dehaspe, L. and Toivonen, H. (1999). Discovery of frequent Datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36.

Dehaspe, L., Toivonen, H., and King, R. D. (1998). Finding frequent substructures in chemical compounds. In Agrawal, R., Stolorz, P. E., and Piatetsky-Shapiro, G., editors, *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 30–36. AAAI Press.

DeJong, G. (2004). Explanation-based learning. In Tucker, A., editor, *Computer Science Handbook, Second Edition*. Chapman & Hall/CRC.

DeJong, G. (2006). Toward robust real-world inference: A new perspective on explanation-based learning. In Fürnkranz, J., Scheffer, T., and Spiliopoulou, M., editors, *Proceedings of the 17th European Conference on Machine Learning (ECML-06)*, volume 4212 of *Lecture Notes in Computer Science*, pages 102–113. Springer.

Denecker, M. and Vennekens, J. (2007). Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In Baral, C., Brewka, G., and Schlipf, J., editors, *Logic Programming and Nonmonotonic Reasoning*, volume 4483 of *Lecture Notes in Computer Science*, pages 84–96. Springer.

Džeroski, S. and Lavrač, N., editors (2001). *Relational Data Mining*. Springer.

Eisner, J. and Blatz, J. (2007). Program transformations for optimization of parsing algorithms and other weighted logic programs. In Wintner, S., editor, *Proceedings of the 11th Conference on Formal Grammar (FG-06)*, pages 45–85. CSLI Publications.

Eisner, J., Goldlust, E., and Smith, N. (2005). Compiling Comp Ling: Weighted dynamic programming and the Dyna language. In *Proceedings of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP-05)*. The Association for Computational Linguistics.

Esposito, F., Fanizzi, N., Ferilli, S., and Semeraro, G. (2000). Ideal theory refinement under object identity. In Langley, P., editor, *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 263–270. Morgan Kaufmann.

Fayyad, U. M., Piatetsky-Shapiro, G., and Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37–54.

Fierens, D., Blockeel, H., Bruynooghe, M., and Ramon, J. (2005). Logical Bayesian networks and their relation to other probabilistic logical models. In Kramer and Pfahringer [2005], pages 121–135.

Flach, P. (1994). *Simply logical - Intelligent Reasoning by Example*. John Wiley.

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

Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. (1999). Learning probabilistic relational models. In Dean, T., editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1300–1309. Morgan Kaufmann.

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

Garriga, G. C., Khardon, R., and De Raedt, L. (2007). On mining closed sets in multi-relational data. In Veloso [2007], pages 804–809.

Gelbukh, A. F. and Morales, A. F. K., editors (2007). *Proceedings of the 6th Mexican International Conference on Artificial Intelligence (MICAI-07)*, volume 4827 of *Lecture Notes in Computer Science*. Springer.

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K. A., editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming (ICLP/SLP-88)*, pages 1070–1080.

Getoor, L. and Taskar, B., editors (2007). *Statistical Relational Learning*. The MIT press.

Goodman, J. (1999). Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Goodman, N., Mansinghka, V. K., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. (2008). Church: a language for generative models. In McAllester, D. A. and Myllymäki, P., editors, *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence (UAI-08)*, pages 220–229. AUAI Press.

Graf, P. (1996). *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer.

Gupta, R. and Sarawagi, S. (2006). Creating probabilistic databases from information extraction models. In Dayal, U., Whang, K.-Y., Lomet, D. B., Alonso, G., Lohman, G. M., Kersten, M. L., Cha, S. K., and Kim, Y.-K., editors, *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB-06)*, pages 965–976. ACM.

Gutmann, B., Jäger, M., and De Raedt, L. (2010a). Extending ProbLog with continuous distributions. In Frasconi, P. and Lisi, F. A., editors, *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP–10)*.

Gutmann, B., Kimmig, A., Kersting, K., and De Raedt, L. (2008). Parameter learning in probabilistic databases: A least squares approach. In Daelemans, W., Goethals, B., and Morik, K., editors, *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD-08), Part I*, volume 5211 of *Lecture Notes in Computer Science*, pages 473–488. Springer.

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

Gutmann, B., Thon, I., and De Raedt, L. (2010c). Learning the parameters of probabilistic logic programs from interpretations. Technical Report CW 584, Department of Computer Science, Katholieke Universiteit Leuven, Belgium.

Haas, L. M. and Tiwary, A., editors (1998). *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*. ACM Press.

Hintsanen, P. (2007). The most reliable subgraph problem. In Kok, J. N., Koronacki, J., López de Mántaras, R., Matwin, S., Mladenic, D., and Skowron, A., editors, *Proceedings of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD-07)*, volume 4702 of *Lecture Notes in Computer Science*, pages 471–478. Springer.

Hintsanen, P. and Toivonen, H. (2008). Finding reliable subgraphs from large probabilistic graphs. *Data Mining and Knowledge Discovery*, 17(1):3–23.

Hirsh, H. (1987). Explanation-based generalization in a logic-programming environment. In McDermott, J. P., editor, *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 221–227.

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

Jäger, M. (1997). Relational Bayesian networks. In Geiger, D. and Shenoy, P. P., editors, *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 266–273. Morgan Kaufmann.

Karp, R. M. and Luby, M. (1983). Monte-Carlo algorithms for enumeration and reliability problems. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS-83)*, pages 56–64. IEEE Computer Society.

Kasari, M., Toivonen, H., and Hintsanen, P. (2010). Fast discovery of reliable $k$-terminal subgraphs. In Zaki, M. J., Yu, J. X., Ravindran, B., and Pudi, V., editors, *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD-10), Part II*, volume 6119 of *Lecture Notes in Computer Science*, pages 168–177. Springer.

Kersting, K. and De Raedt, L. (2008). Basic principles of learning Bayesian logic programs. In De Raedt et al. [2008a], pages 189–221.

Kimmig, A. and Costa, F. (2010). Link and node prediction in metabolic networks with probabilistic logic. In *Workshop on Analysis of Complex NEtworks at ECML/PKDD*.

Kimmig, A. and De Raedt, L. (2008). Probabilistic local pattern mining. In Železný, F. and Lavrač, N., editors, *Proceedings of Inductive Logic Programming (ILP 2008), Late Breaking Papers*, pages 63–68.

Kimmig, A. and De Raedt, L. (2009). Local query mining in a probabilistic Prolog. In Boutilier, C., editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 1095–1100.

Kimmig, A., De Raedt, L., and Toivonen, H. (2007). Probabilistic explanation based learning. In Kok, J. N., Koronacki, J., López de Mántaras, R., Matwin, S., Mladenic, D., and Skowron, A., editors, *Proceedings of the 18th European Conference on Machine Learning (ECML-07)*, volume 4701 of *Lecture Notes in Computer Science*, pages 176–187. Springer.

Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., and Rocha, R. (2010). On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming (TPLP)*. to appear.

Kimmig, A., Gutmann, B., and Santos Costa, V. (2009). Trading memory for answers: Towards tabling ProbLog. In Domingos, P. and Kersting, K., editors, *International Workshop on Statistical Relational Learning*.

Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., and De Raedt, L. (2008). On the Efficient Execution of ProbLog Programs. In Garcia de la Banda, M. and Pontelli, E., editors, *Proceedings of the 24th International Conference on Logic Programming (ICLP-08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 175–189. Springer.

Koppel, M., Feldman, R., and Segre, A. M. (1994). Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research (JAIR)*, 1:159–208.

Kramer, S. and Pfahringer, B., editors (2005). *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP-05)*, volume 3625 of *Lecture Notes in Computer Science*. Springer.

Kwoh, C.-K. and Gillies, D. (1996). Using hidden nodes in Bayesian networks. *Artificial Intelligence*, 88(1-2):1–38.

Landwehr, N. (2009). *Trading Expressivity for Efficiency in Statistical Relational Learning*. PhD thesis, K.U.Leuven.

Langley, P. (1989). Unifying themes in empirical and explanation-based learning. In Segre, A. M., editor, *Proceedings of the 6th International Workshop on Machine Learning (ML-89)*, pages 2–4. Morgan Kaufmann.

Langohr, L. and Toivonen, H. (2009). Finding representative nodes in probabilistic graphs. In *Workshop on Explorative Analytics of Information Networks at ECML PKDD*, pages 65–76.

Lao, N. and Cohen, W. (2010). Relational retrieval using a combination of path-constrained random walks. *Machine Learning*, 81:53–67.

Lloyd, J. W. (1989). *Foundations of Logic Programming*. Springer, 2. edition.

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

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

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

Mantadelis, T. and Janssens, G. (2009). Tabling relevant parts of SLD proofs for ground goals in a probabilistic setting. In Tarau, P., Moura, P., and Zhou, N.-F., editors, *Proceedings of the 9th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS-09)*, pages 36–50.

Mantadelis, T. and Janssens, G. (2010). Dedicated tabling for a probabilistic setting. In Hermenegildo, M. V. and Schaub, T., editors, *Technical Communications of the 26th International Conference on Logic Programming (ICLP-10)*, volume 7 of *LIPIcs*, pages 124–133. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Mantadelis, T., Rocha, R., Kimmig, A., and Janssens, G. (2010). Preprocessing Boolean formulae for BDDs in a probabilistic context. In Janhunen, T. and Niemelä, I., editors, *Proceedings of the 12th European Conference on Logics in Artificial Intelligence (JELIA-2010)*, volume 6341 of *Lecture Notes in Computer Science*, pages 260–272. Springer.

Minato, S., Satoh, K., and Sato, T. (2007). Compiling Bayesian networks by symbolic probability calculation based on zero-suppressed BDDs. In Veloso [2007], pages 2550–2555.

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

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Morishita, S. and Sese, J. (2000). Traversing itemset lattice with statistical metric pruning. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS-00)*, pages 226–236. ACM.

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

Muggleton, S. and De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679.

Ng, R. and Subrahmanian, V. S. (1992). Probabilistic logic programming. *Information and Computation*, 101(2):150–201.

Ng, R. T., Lakshmanan, L. V. S., Han, J., and Pang, A. (1998). Exploratory mining and pruning optimizations of constrained association rules. In Haas and Tiwary [1998], pages 13–24.

Nienhuys-Cheng, S. and de Wolf, R., editors (1997). *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science*. Springer.

Nijssen, S. and Kok, J. N. (2003). Efficient frequent query discovery in farmer. In Lavrac, N., Gamberger, D., Blockeel, H., and Todorovski, L., editors, *Proceedings of the 7th European Conference on Principles and Practice of Knowledge*

*Discovery in Databases (PKDD-03)*, volume 2838 of *Lecture Notes in Computer Science*, pages 350–362. Springer.

Nottelmann, H. and Fuhr, N. (2001). Learning probabilistic Datalog rules for information classification and transformation. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management (CIKM-01)*, pages 387–394. ACM.

Olteanu, D. and Huang, J. (2008). Using OBDDs for efficient query evaluation on probabilistic databases. In Greco, S. and Lukasiewicz, T., editors, *Proceedings of the 2nd international conference on Scalable Uncertainty Management (SUM-08)*, pages 326–340. Springer.

Paes, A., Revoredo, K., Zaverucha, G., and Santos Costa, V. (2005). Probabilistic first-order theory revision from examples. In Kramer and Pfahringer [2005], pages 295–311.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

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

Pfeffer, A. (2001). IBAL: A probabilistic rational programming language. In Nebel, B., editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 733–740. Morgan Kaufmann.

Pfeffer, A. (2009). Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics.

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

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

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

Poole, D. (2008). The independent choice logic and beyond. In De Raedt et al. [2008a], pages 222–243.

Pouly, M., Haenni, R., and Wachter, M. (2007). Compiling solution configurations in semiring valuation systems. In Gelbukh and Morales [2007], pages 248–259.

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

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

Richardson, M. and Domingos, P. (2006). Markov logic networks. *Machine Learning*, 62(1-2):107–136.

Riguzzi, F. (2007). A top down interpreter for LPAD and CP-logic. In Basili, R. and Pazienza, M. T., editors, *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence (AI*IA-07)*, volume 4733 of *Lecture Notes in Computer Science*, pages 109–120. Springer.

Riguzzi, F. (2009). Extended semantics and inference for the Independent Choice Logic. *Logic Journal of the IGPL*, 17(6):589–629.

Russell, S. and Norvig, P. (2004). *Artificial Intelligence: a Modern Approach*. Prentice Hall, 2. edition.

Santos Costa, V. (2007). Prolog performance on larger datasets. In Hanus, M., editor, *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages (PADL-07)*, volume 4354 of *Lecture Notes in Computer Science*, pages 185–199. Springer.

Santos Costa, V., Page, D., Qazi, M., and Cussens, J. (2003). CLP(BN): constraint logic programming for probabilistic knowledge. In Meek, C. and Kjærulff, U., editors, *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI-03)*, pages 517–524. Morgan Kaufmann.

Santos Costa, V., Sagonas, K. F., and Lopes, R. (2007). Demand-driven indexing of Prolog clauses. In Dahl, V. and Niemelä, I., editors, *Proceedings of the 23rd International Conference on Logic Programming (ICLP-07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 395–409. Springer.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In Sterling, L., editor, *Proceedings of the 12th International Conference on Logic Programming (ICLP-95)*, pages 715–729. MIT Press.

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

Sato, T., Zhou, N.-F., Kameya, Y., and Izumi, Y. (2010). *PRISM User's Manual (Version 2.0)*.

Saul, L., Jaakkola, T., and Jordan, M. (1996). Mean field theory for sigmoid belief networks. *Journal of Artificial Intelligence Research (JAIR)*, 4:61–76.

Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., and Toivonen, H. (2006). Link discovery in graphs derived from biological databases. In Leser, U., Naumann, F., and Eckman, B. A., editors, *Proceedings of the Third International Workshop on Data Integration in the Life Sciences (DILS-06)*, volume 4075 of *Lecture Notes in Computer Science*, pages 35–49. Springer.

Shterionov, D. S., Kimmig, A., Mantadelis, T., and Janssens, G. (2010). DNF sampling for ProbLog inference. In *Proceedings of the 10th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS-10)*.

Suciu, D. (2008). Probabilistic databases. *SIGACT News*, 39(2):111–124.

Thon, I., Landwehr, N., and De Raedt, L. (2008). A simple model for sequences of relational state descriptions. In Daelemans, W., Goethals, B., and Morik, K., editors, *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD-08), Part II*, volume 5212 of *Lecture Notes in Computer Science*, pages 506–521. Springer.

Toivonen, H., Mahler, S., and Zhou, F. (2010). A framework for path-oriented network simplification. In Cohen, P. R., Adams, N. M., and Berthold, M. R., editors, *Proceedings of the 9th International Symposium on Advances in Intelligent Data Analysis (IDA-10)*, volume 6065 of *Lecture Notes in Computer Science*, pages 220–231. Springer.

Tsur, S., Ullman, J. D., Abiteboul, S., Clifton, C., Motwani, R., Nestorov, S., and Rosenthal, A. (1998). Query flocks: A generalization of association-rule mining. In Haas and Tiwary [1998], pages 1–12.

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

Van den Broeck, G., Thon, I., van Otterlo, M., and De Raedt, L. (2010). DTProbLog: A decision-theoretic probabilistic Prolog. In Fox, M. and Poole, D., editors, *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*. AAAI Press.

Van Gelder, A., Ross, K. A., and Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650.

Van Harmelen, F. and Bundy, A. (1988). Explanation based generalization = partial evaluation. *Artificial Intelligence*, 36:401–412.

Veloso, M. M., editor (2007). *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*.

Vennekens, J. (2007). *Algebraic and logical study of constructive processes in knowledge representation*. PhD thesis, K.U. Leuven.

Vennekens, J., Denecker, M., and Bruynooghe, M. (2009). FO(ID) as an extension of DL with rules. In Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., and Simperl, E., editors, *Proceedings of the 6th European Semantic Web Conference (ESWC-09)*, volume 5554 of *Lecture Notes in Computer Science*, pages 384–398. Springer.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. (2004). Logic programs with annotated disjunctions. In Demoen, B. and Lifschitz, V., editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP-04)*, volume 3132 of *Lecture Notes in Computer Science*, pages 431–445. Springer.

Wachter, M., Haenni, R., and Pouly, M. (2007). Optimizing inference in Bayesian networks and semiring valuation algebras. In Gelbukh and Morales [2007], pages 236–247.

Wrobel, S., Wettschereck, D., Sommer, E., and Emde, W. (1996). Extensibility in data mining systems. In Simoudis, E., Han, J., and Fayyad, U. M., editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 214–219. AAAI Press.

Yan, X. and Han, J. (2002). gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM-02)*, pages 721–724. IEEE Computer Society.

Zelle, J. M. and 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), Volume 1*, pages 748–753.

Zhang, Q., Li, F., and Yi, K. (2008). Finding frequent items in probabilistic data. In Wang, J. T.-L., editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-08)*, pages 819–832. ACM.

Zou, Z., Gao, H., and Li, J. (2010). Discovering frequent subgraphs over uncertain graph databases under probabilistic semantics. In Rao, B., Krishnapuram, B., Tomkins, A., and Yang, Q., editors, *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-10)*, pages 633–642. ACM.

Zou, Z., Li, J., Gao, H., and Zhang, S. (2009). Frequent subgraph pattern mining on uncertain graph data. In Cheung, D. W.-L., Song, I.-Y., Chu, W. W., Hu, X., and Lin, J. J., editors, *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM-09)*, pages 583–592. ACM.

# Publication List

## Journal Articles

- Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. *On the implementation of the probabilistic logic programming language ProbLog.* Theory and Practice of Logic Programming, 2010. Accepted.
- Luc De Raedt, Kristian Kersting, Angelika Kimmig, Kate Revoredo, and Hannu Toivonen. *Compressing probabilistic Prolog programs.* Machine Learning, 70(2-3):151–168. Springer, 2008.

## Conference Papers

- Maurice Bruynooghe, Theofrastos Mantadelis, Angelika Kimmig, Bernd Gutmann, Joost Vennekens, Gerda Janssens, and Luc De Raedt. *ProbLog technology for inference in a probabilistic first order logic.* In H. Coelho, R. Studer, and M. Wooldridge, editors, Proceedings of the 19th European Conference on Artificial Intelligence (ECAI–2010), pages 719–724. IOS Press, 2010.
- Theofrastos Mantadelis, Ricardo Rocha, Angelika Kimmig, and Gerda Janssens. *Preprocessing Boolean formulae for BDDs in a probabilistic context.* In T. Janhunen and I. Niemelä, editors, Proceedings of the 12th European Conference on Logics in Artificial Intelligence (JELIA–2010), pages 260–272. Springer, 2010.
- Angelika Kimmig and Luc De Raedt. *Local query mining in a probabilistic Prolog.* In C. Boutilier, editor, Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI–09), pages 1095–1100. AAAI Press, 2009.

- Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. *Parameter learning in probabilistic databases: A least squares approach.* In W. Daelemans, B. Goethals, and K. Morik, editors, Proceedings of the 19th European Conference on Machine Learning (ECML–2008), pages 473–488. Springer, 2008.

- Angelika Kimmig, Vítor Santos Costa, Ricardo Rocha, Bart Demoen, and Luc De Raedt. *On the efficient execution of ProbLog programs.* In M. Garcia de la Banda and E. Pontelli, editors, Proceedings of the 24th International Conference on Logic Programming (ICLP–2008), pages 175–189. Springer, 2008.

- Luc De Raedt, Kristian Kersting, Angelika Kimmig, Kate Revoredo, and Hannu Toivonen. *Revising probabilistic Prolog programs.* In S. Muggleton, R. Otero, and A. Tamaddoni-Nezhad, editors, 16th International Conference on Inductive Logic Programming (ILP–2006) – Revised Selected Papers, pages 30–33. Springer, 2007.

- Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. *ProbLog: A probabilistic Prolog and its application in link discovery.* In M. Veloso, editor, Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI–2007), pages 2462–2467. 2007.

- Angelika Kimmig, Luc De Raedt, and Hannu Toivonen. *Probabilistic explanation based learning.* In J.N. Kok, J. Koronacki, R. López de Mántaras, S. Matwin, D. Mladenic, and A. Skowron, editors, Proceedings of the 18th European Conference on Machine Learning (ECML–2007), pages 176–187. Springer, 2007. **Winner of the ECML Best Paper Award** (592 submissions).

- Luc De Raedt, Kristian Kersting, Angelika Kimmig, Kate Revoredo, and Hannu Toivonen. *Revising probabilistic Prolog Programs.* In S. Muggleton and R. Otero, editors, 16th International Conference on Inductive Logic Programming (ILP–2006) – Short Papers, pages 52–54. 2006.

## Workshop Papers

- Angelika Kimmig and Fabrizio Costa. *Link and node prediction in metabolic networks with probabilistic logic.* In Analysis of Complex NEtworks, 2010. A shorter version of this paper also appeared at the 7th International Symposium on Networks in Bioinformatics, 2010.

- Dimitar Sht. Shterionov, Angelika Kimmig, Theofrastos Mantadelis, and Gerda Janssens. *DNF sampling for ProbLog inference.* In Proceedings of

the International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), 2010.

- Maurice Bruynooghe, Broes De Cat, Jochen Drijkoningen, Daan Fierens, Jan Goos, Bernd Gutmann, Angelika Kimmig, Wouter Labeeuw, Steven Langenaken, Niels Landwehr, Wannes Meert, Ewoud Nuyts, Robin Pellegrims, Roel Rymenants, Stefan Segers, Ingo Thon, Jelle Van Eyck, Guy Van den Broeck, Tine Vangansewinkel, Lucie Van Hove, Joost Vennekens, Timmy Weytjens, and Luc De Raedt. *An exercise with statistical relational learning systems.* In P. Domingos and K. Kersting, editors, International Workshop on Statistical Relational Learning, 2009.

- Angelika Kimmig, Bernd Gutmann, and Vítor Santos Costa. *Trading memory for answers: Towards tabling ProbLog.* In P. Domingos and K. Kersting, editors, International Workshop on Statistical Relational Learning, 2009.

- Joost Vennekens, Angelika Kimmig, Theofrastos Mantadelis, Bernd Gutmann, Maurice Bruynooghe, and Luc De Raedt. *From ProbLog to first order logic: A first exploration.* In P. Domingos and K. Kersting, editors, International Workshop on Statistical Relational Learning, 2009.

- Luc De Raedt, Bart Demoen, Daan Fierens, Bernd Gutmann, Gerda Janssens, Angelika Kimmig, Niels Landwehr, Theofrastos Mantadelis, Wannes Meert, Ricardo Rocha, Vítor Santos Costa, Ingo Thon, and Joost Vennekens. *Towards digesting the alphabet-soup of statistical relational learning.* In D. Roy, J. Winn, D. McAllester, V. Mansinghka, and J. Tenenbaum, editors, Proceedings of the 1st Workshop on Probabilistic Programming: Universal Languages, Systems and Applications, 2008.

- Bernd Gutmann, Angelika Kimmig, Luc De Raedt, and Kristian Kersting. *Parameter learning in probabilistic databases: A least squares approach.* In 6th International Workshop on Mining and Learning with Graphs (MLG 2008); in F. Železný and N. Lavrač, editors, 18th International Conference on Inductive Logic Programming (ILP–2008) – Late Breaking Papers, pages 38–43, 2008; and in L. Wehenkel, P. Geurts, and R. Marée, editors, Proceedings of Benelearn 08, pages 25–26, 2008.

- Angelika Kimmig and Luc De Raedt. *Probabilistic local pattern mining.* In F. Železný and N. Lavrač, editors, 18th International Conference on Inductive Logic Programming (ILP–2008) – Late Breaking Papers, pages 63–68, 2008.

- Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. *ProbLog: A probabilistic Prolog and its application in link discovery.* In Dutch-Belgian Database Day, 2006.

## Book Chapters

- Luc De Raedt, Angelika Kimmig, Bernd Gutmann, Kristian Kersting, Vítor Santos Costa, and Hannu Toivonen. *Probabilistic inductive querying using ProbLog.* In S. Džeroski, B. Goethals, and P. Panov, editors, Inductive Databases and Constraint-Based Data Mining. Springer, accepted.

## Technical Reports

- Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. *Parameter estimation in ProbLog from annotated queries.* CW Reports CW583, Department of Computer Science, K.U.Leuven, 2010.
- Luc De Raedt, Angelika Kimmig, Bernd Gutmann, Kristian Kersting, Vítor Santos Costa, and Hannu Toivonen. *Probabilistic inductive querying using ProbLog.* CW Reports CW552, Department of Computer Science, K.U.Leuven, 2009.

# Curriculum vitae

Angelika Kimmig was born on March 27, 1979, in Rottweil, Germany. She went to school at the Konrad-Witz-Schule and the Leibniz Gymnasium in Rottweil, from where she graduated with “Abitur” in July 1998. After a voluntary year of social service, she started studying computer science at the Albert-Ludwigs-Universität in Freiburg, Germany, where she obtained a “Vordiplom” in November 2001 and a “Diplom” in computer science in January 2006. In March 2006, she joined the machine learning group at the Albert-Ludwigs-Universität Freiburg and started work on a Ph.D. in the area of probabilistic logic learning, supervised by Prof. Luc De Raedt. In January 2007, she moved with the group of Luc De Raedt to the DTAI (Declaratieve Talen en Artificiële Intelligentie) group at the Katholieke Universiteit Leuven, Belgium. Since October 2007, her research has been funded by a personal grant from the Research Foundation - Flanders (FWO Vlaanderen). In November 2010, she will defend her Ph.D. thesis on ”A probabilistic Prolog and its applications” at the Katholieke Universiteit Leuven.