KATHOLIEKE UNIVERSITEIT
LEUVEN

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

Efficient Algorithms for Prolog Based Probabilistic Logic Programming

Theofrastos MANTADELIS

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

November 2012

Efficient Algorithms for Prolog Based Probabilistic Logic Programming

Theofrastos MANTADELIS

Jury:
Prof. dr. Adhemar Bultheel, chair
Prof. dr. ir. Gerda Janssens, promotor
Prof. dr. ir. Maurice Bruynooghe
Prof. dr. Patrick De Causmaecker
Prof. dr. Luc De Raedt
Prof. dr. Bart Demoen
Prof. dr. Ricardo Rocha
(Universidade do Porto, Portugal)

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

November 2012

D/2012/7515/128
ISBN 978-94-6018-594-6

Acknowledgements

First of all I would like to thank anyone reading this thesis. Hopefully, parts of this thesis will be useful to other for using.

I owe sincere thankfulness to my promoter, Gerda Janssens, who supported and guided me through the whole process of my doctoral research. I specially thank her for her patience and help through my dissertation writing. I am sure that this dissertation would not have been possible without her support and understanding.

I would like to thank the members of my jury Maurice Bruynooghe, Patrick De Causmaecker, Luc De Raedt, Bart Demoen and Ricardo Rocha for the precious time they devoted reading my thesis and their constructive comments. I also want to thank Adhemar Bultheel for chairing the jury and especially for his optimistic comments at the beginning of one of the most stressful presentations I ever gave.

Thank you Maurice, without you as a captain of DTAI I would have lost hope and stope believing that I can make it out of the final storm. I want to thank Patrick and Luc for remaining on the ship until the end. A special thanks to Bart, for helping me in the beginning of my doctoral research and coming to visit me at the hospital. I would also like to show my gratitude to Ricardo for his enthusiasm for cooperation and his invitation to the university of Porto but most of all for his kindness.

I want to thank all my colleagues for creating a great group and especially my closer colleagues Angelika Kimmig, Bernd Gutmann and Daan Fierens for sharing the same office with me without complaining too much over my noisy personality. Special thanks to Koosha Paridel for his friendship and working with me. I wanna thank Nima Taghipour, Gitte Vanwinckelen, Jantobias Muehlberg and Nick Nikiforakis for being in the department and sharing long replenishing breaks with me. I owe special gratitude to Dimitar Shterionov for making me proud as my student, becoming a reliable colleague and a great

friend. I also want to show my gratitude to Paulo Moura for helping me, inviting me to Portugal and accepting my invitation to work together.

My pursue for a doctoral research did not started in the university but when I was a little kid. For that I owe my eternal love and gratitude to my family. The first person that gave me the idea for pursuing science was my grandmother Irene Mantadeli, *στη μνήμη σου γιαγιά*. My parents Aristidis Mantadelis and Marina Mantadeli, very well know that I was never a good student. I managed to reach here by my father's patience and insistence for studying and going to university. *Μπαμπά, σε ευχαριστώ για τις επιλογές που μου έδωσες και για την επιμονή σου στο να σπουδάσω*. I know that my mother always wanted me to become a medical doctor, I want to thank her for her unconditional love and being equally proud for me being the other type of doctor. I also want to specially thank her for proofreading parts of my thesis. I also want to thank my sister Irene Gentili for being close to me and supporting me in moments of distress.

Many people helped me one way or another in writing this dissertation. I would like to thank Maria Errie Phocas for helping me to prepare for TOEFL and Stephanie Morrison for proofreading my text and supporting my family for so many years. I owe my gratitude to Helena De Kok for helping me to start writing the thesis and being a good friend for so long. They say that starting is the most difficult part, but a task becomes equally difficult when you are closing to the end of it. The thesis finished by the mere support of my close friend Vasilis Maragkos. He supported me with his presence at a moment that I was ready to give up showing me that the end is close and reachable.

Finally, I had a great deal of moral support from my valuable friends in Greece: Antonis Kariotis, Costas Babos, Nikos Mavromatis, Stefanos Markopoulos and here: Christos Varsakelis, Naouma Siouta, Alexandros and Michael Spyrantis, Alice Dillon, Nuria Vendrell Llopis, André Arroyo Ruiz, Leandro Doctors. I have met and made many friends in my pursue for doctoral research and each one of them helped me continue.

I gratefully acknowledge the financial support received for the work performed during my doctoral research by the GOA/08/008 Probabilistic Logic Learning project.

Abstract

The integration of probabilistic reasoning with logic programming has become one of the challenges in Artificial Intelligence. Lately, a lot of Probabilistic Logic Programming (PLP) formalisms have surfaced. Given that PLP is the combination of logic programming and probabilities which are two very different fields, it is expected that researchers from several fields come up with different approaches to tackle the presented challenges. This has resulted in a new discipline called Probabilistic Logic Learning (PLL) or Statistical Relational Learning (SRL) and a very active research community.

Within this community, ProbLog a probabilistic extension of Prolog, has appeared. ProbLog was motivated by the task of mining links in large probabilistic graphs. The simple but powerful ProbLog formulation was extended in order to support inference on several different models. Soon ProbLog evolved into a general purpose probabilistic programming language that provides infrastructure for many PLL/SRL tasks. The two most critical aspects of a PLP language are its expression power, and its scalability over common PLL problems.

This thesis focuses on the extension and implementation of ProbLog. Tabling is a well known method for avoiding re-computation in logic programming; we present how tabling can be implemented for ProbLog. Tabling has significant benefits for performance and also allows ProbLog to handle cyclic programs. In order for the ProbLog system to perform inference, it manipulates large Boolean formulae. We present the manipulation of Boolean formulae and several novel algorithms that improve the performance of this task. Furthermore, we present patterns, called AND/OR-clusters, that can be used to reduce a Boolean formula to an equivalent one. Besides performance improvements of the ProbLog system, we also present several novel extensions to the ProbLog language, such as: general negation, probabilistic meta calls and ProbLog answers. Finally, we present two applications of ProbLog that use several of the features we have contributed to ProbLog.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Probabilistic Logic Programming

We start by briefly describing the field of **Probabilistic Logic Programming** (PLP). Lately, we have seen rapid growth in this field which is also known under the names statistical relational learning [21], probabilistic logic learning [14] and probabilistic inductive logic programming [15]. Several frameworks have appeared in order to provide tools for assist in performing these tasks. Over time the generalization of these tools has driven several researchers to develop specialized programming languages [16, 17, 23, 57–59, 67, 73].

The birth of the PLP field actually started when, in parallel, researchers from the inductive logic programming community (ILP) incorporated stochastic aspects in their methodologies and when researchers from traditional machine learning (ML) community incorporated relational information in their methodologies. Soon these two communities combined their specialities in order for the field to appear. Thereafter, the field has bloomed and the new methodologies have been applied in several applications [22].

To address the needs of the new field, several research groups developed frameworks or extended general programming languages in order to provide the infrastructure required for the development of the combined methodologies. In the logic society where ILP originated, the research focused on adding stochastic elements to propositional logic and other logics. Several formulations based on logic programming appeared such as PRISM [73], ICL [63, 64] and, ProbLog [16]. Even formulations based on first order logic emerged such as MLNs [67], FOProbLog [8]. Also similar frameworks came forth from the

functional programming point of view, such as IBAL [57], Church [23] and Figaro [59].

Probabilistic logic programming aims at extending logic programming with probabilistic information. Logic programming is based on using logical statements in order to represent the problem the program aims to solve. These programs have both a declarative and a procedural meaning. Usually, logic programming uses a set of clauses in order to deduce new clauses that are asked from the user.

Typical probabilistic extensions will introduce probability in these rules in the following ways, (a) by introducing the probability that an "event" causes with some probability an other "event" which leads to Causal Probabilistic Logic (CP-Logic) [87] or (b) by introducing probabilistic independent choices [63], or (c) by attaching probability as the existence of a rule like in the Distribution Semantics [72]. Finally, the user can query the probabilistic theory in order to deduce more probabilistic rules.

Related Work

ProbLog is closely related to other probabilistic logic systems such as PHA [60, 61], ICL [62], PRISM [73], and PITA [69]. All these systems either take assumptions that reduce their usability in modelling or are forced to tackle computational expensive problems. The motivation behind developing these systems is to use them for machine learning and data mining tasks for which, in both cases, big amounts of data need to be processed. Scalability under these tasks is very important.

PHA, which stands for Probabilistic Horn Abduction, is one of the first Prolog based frameworks that can perform probabilistic inference. The system was inspired by Bayesian Networks and was designed with the aim of modelling Bayesian Networks. To that end PHA defines "disjoints" which are assumed mutually independent. PHA's disjoints can be defined both for facts and rules. Furthermore, it takes several other assumptions of which the two most important are: PHA programs must be acyclic and all rules of an atom must be mutual exclusive.

Programming in Statistical Modelling (PRISM) is very similar to PHA. Its aim is to also model (Hidden) Markov Models. To that end it uses "multi value switches" which are assumed to be independent of each other. PRISM's multi value switches are a separate entity of the language and are used similarly as facts. Furthermore, it takes the same two important assumptions that PHA

also has. A significant difference from PHA is that PRISM uses tabling to significantly boost the system performance.

ICL stands for Independent Choice Logic and is a natural successor of PHA. PHA's assumption about mutual exclusiveness implies it cannot model several relevant problems. ICL addressed exactly that issue by lifting that assumption. By lifting that assumption ICL inference now requires to solve the disjoint sum problem. Unfortunately, the ICL implementation AILog2 that uses conditioning to assess the probability of a sum of products[1] does not scale to larger problems. Finally, ICL retains the assumption that ICL programs must be acyclic.

ProbLog's initial implementation target is to overcome the expression limitations of PRISM and the performance limitations of ICL. Instead of using independent disjunctions ProbLog uses "probabilistic facts" which again are assumed to be independent. The initial ProbLog implementation still assumes that the ProbLog program is acyclic. The most significant difference was the introduction of Reduced Ordered Binary Decision Diagrams (ROBDDs) for solving the disjoint sum problem. This approach, for inference, scales much better than ICL's implementation. The ProbLog system was motivated in order to perform data mining over large probabilistic networks, such as Sevon's Biomine network [77], which contains relationships among genes, proteins, enzymes etc. Mining these type of networks has been considered an important and challenging task [56].

Finally, PITA is not a full PLP framework but a specific tabled inference approach that very much resembles the tabling used by ProbLog. A limitation of PITA is that it does not support cycle handling and thus is again limited to acyclic programs.

Contributions

In this thesis we focus on implementation issues triggered from the probabilistic extension of Prolog: **ProbLog**. ProbLog's semantics are briefly presented in Chapter 2. We discuss efficiency and implementation issues related to ProbLog and, more generally, with extending Prolog with probabilistic information. For what concerns efficiency, the initial ProbLog implementation used ROBDDs in a rather naive way, did not have tabling and, used a single Boolean formula preprocessing method. Throughout the thesis we present how we improved all these. Furthermore, the initial ProbLog implementation had limited modelling expressiveness, it did not support general negation, did not handle acyclic

[1] To assess the probability of a sum of products we need, to solve the disjoint sum problem; see [66] for more details.

programs and did not supported probabilistic inference nesting. Again, through the thesis, we are going to present how we added these features in the system.

We make a clear distinction between three steps that ProbLog takes in order to calculate the probability of a query: the first step is the SLD resolution, the second step is Boolean formulae preprocessing and the third step is Boolean formulae compilation. In ProbLog we compile Boolean formulae in ROBDDs in order to solve the disjoint sum problem. Inference in many probabilistic logic systems is based on representing the proofs of a query as a Boolean formula in disjunctive normal form (DNF). Assessing the probability of such a formula is known as a #P-hard task. In practice for ProbLog, a large DNF is given to a ROBDD software package to construct the corresponding ROBDD. The DNF has to be transformed into the input format of the package. This procedure is called the preprocessing step.

The work presented in this thesis, aims to improve the performance of ProbLog in several different aspects. The main performance contributions of this thesis are **tabling** which improves the first step of inference, several **Boolean formulae preprocessing** methods that target to improve the second and third step of inference and, **variable compression** an approach for reducing Boolean formulae before compiling them in order to improve the performance of the ROBDD generation. Besides performance related contributions, we also extended the expressive power of the ProbLog language and added new functionality such as: **general negation**, support for **cyclic programs**, and **nested inference**. Finally, we also implemented two ProbLog applications that were only feasible because of the system extensions we have made.

Our contributions in more detail include a script language that we developed in order to describe Boolean formulae for ROBDD generation. This script language was our first contribution to the ProbLog system and it allowed it to handle ROBDDs in a structured and efficient way. We introduced tabling in the system and managed a significant performance improvement over several types of ProbLog programs. Furthermore, by extending our tabling support to handle probabilistic cycles, we achieved in removing the acyclic program assumption. In the thesis we present how we achieved tabling for ProbLog programs and explain how the approach can be extended for probabilistic logic programs in general. Tabling proved to be an indispensable feature for ProbLog, both allowing harder queries to be proven and simplifying the modelling of several problems. Finally, through tabling, a design for implementing general negation for ProbLog was made. In the preprocessing step we investigate and compare different preprocessing methods and add a novel data structure to further improve the preprocessing. Furthermore, we present a generalized preprocessing algorithm that can handle Boolean formulae that have nested parts such as the ones generated from tabling. The nested preprocessing algorithm could be

combined with any of the presented preprocessing algorithms that ProbLog uses. Up to a point, we identify which method performs better depending on the Boolean formula structure and present several different optimisations. Our next important contribution is the identification of patterns formed by Boolean variables that occur in DNF Boolean formulae, namely AND-clusters and OR-clusters. Our method compresses the variables in these clusters and thus reduces the size of ROBDDs without affecting the probability. We give two polynomial time algorithms that detect AND-clusters and OR-clusters respectively in DNF Boolean formulae. While the algorithms presented are only applicable to Boolean formulae in DNF form, we still achieved significant improvements to the generation of ROBDDs. Our approach makes it feasible to deal with ProbLog queries that give rise to larger DNFs. While our methods for Boolean formulae manipulation are motivated and applied in the ProbLog context, our results and concepts could be applied in any discipline that works with DNF Boolean formulae.

Moreover, we contributed a new ProbLog implementation namely ProbLog2011[2] where we address several challenges that were discovered over time. The most important contribution in that implementation is the abstraction of ProbLog's inference in what we call a ProbLog engine. A ProbLog engine is parametrized in such a way that it allows us to implement different inference methods in a similar way. Through the use of ProbLog engine we are able to implement tabling and general negation in a much better way than what in the ProbLog2010 implementation was implemented. Both the new tabling and the new general negation lift several preconditions that the ProbLog2010 implementation required. Furthermore, the ProbLog engine approach allows us to perform nested probabilistic inference. Nesting becomes possible by suspending and resuming instances of ProbLog engines. With our approach we realise several extensions of ProbLog such as meta-calls, negation, and answers of probabilistic goals.

Finally, part of our contributions are two ProbLog applications. The goal of those applications is to demonstrate the usability of ProbLog features and generally the usability of ProbLog in modelling. The first application uses the ProbLog machinery to realise a probabilistic extension of first order logic. The second application is an analyser for a network protocol designed to be used in Mobile Ad hoc networks (MANETs). This work shows how ProbLog can be useful in a field different from data mining or machine learning.

The work related to the script language was presented at a Poster reception [41]. Our contributions for tabling where reported in [42, 43]. The work related

[2]Within the thesis we have two different ProbLog implementations. The original one which we will refer as ProbLog2010 and an alternative which we call MetaProbLog and we refer to it as ProbLog2011.

to the different preprocessing methods was published in [47]. The AND/OR-clusters where stated in [44]. The new implementation and inference nesting was presented in [45]. The applications FOProbLog and Analysing FADIP appeared in [8, 88], [46] respectively.

The rest of the thesis has the following structure. First, we present the basis of ProbLog semantics and implementation in Chapter 2. Chapter 3 gives a thorough description of ProbLog tabling. In Chapter 4 we present several different preprocessing algorithms and a generalized algorithm that manipulates nested Boolean formulae. Chapter 5 presents the AND/OR-clusters and how they can be used to reduce the size of DNF Boolean formulae. Chapter 6 describes the ProbLog2011 implementation and how we achieve nested inference. At Chapter 7 we show two different applications of ProbLog that use several of the contributions presented in this thesis. Finally, we conclude with Chapter 8.

Chapter 2

ProbLog

In this chapter we give a brief introduction and description of ProbLog, a probabilistic extension of Prolog. We explain the distribution semantics which ProbLog uses in Section 2.1, and define the success probability of logic programs in Section 2.2. Then we describe some of the inference algorithms used in ProbLog. We make the distinction between knowledge compilation approaches and the rest of the approaches. We describe in greater detail the knowledge compilation approaches in Section 2.3 making distinct separation of the steps that are involved in their computation. Finally, we briefly mention the other inference approaches in Section 2.4.

2.1 ProbLog and the Distribution Semantics

A ProbLog program T [16, 35, 37] consists of a set of facts annotated with probabilities $p_i :: pf_i$ – called *probabilistic facts* – together with a set of standard definite clauses $h :- b_1, \ldots, b_n.$ that can have positive and negative probabilistic literals in their body. A probabilistic fact pf_i is true with probability p_i. These facts correspond to random variables, which are assumed to be mutually independent. Together, they thus define a distribution over subsets of $L_T = \{pf_1, \ldots, pf_n\}$. The definite clauses add arbitrary *background knowledge* (BK) to those sets of *logical* facts. To keep a natural interpretation of a ProbLog program we assume that probabilistic facts cannot unify with other probabilistic facts or with the background knowledge rule heads.

Definition 1. *ProbLog Program: Formally, a ProbLog program is of the form* $T = \{pf_1, \ldots, pf_n\} \cup BK$.

Probabilistic Facts:

```
0.3::edge(1, 2).
0.7::edge(1, 3).
0.4::edge(2, 3).
```

Background Knowledge:

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

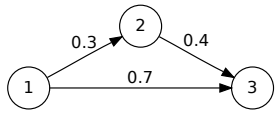


Figure 2.1: A ProbLog program: the probabilistic facts represent the probabilistic graph; the background theory a naive approach to check for reachability.

Given the one-to-one mapping between ground definite clause programs and Herbrand interpretations, a ProbLog program defines a distribution over its Herbrand interpretations.

The distribution semantics are defined by generalising the least Herbrand models of the clauses by including subsets of the probabilistic facts. If fact pf_i is annotated with p_i, pf_i is included in a generalised least Herband model with probability p_i and left out with probability $1 - p_i$. The different facts are assumed to be probabilistically independent, however, negative probabilistic facts in clause bodies allow the user to enforce a choice between two clauses.

As such, a ProbLog program specifies a probability distribution over all its possible non-probabilistic subprograms. The success probability of a query is defined as the probability that the query succeeds in such a random subprogram. ProbLog follows the distribution semantics [72].

For a better explanation of the probability distribution defined by a ProbLog program we use the example program of Figure 2.1.

2.1.1 Possible Worlds

As we previously mentioned ProbLog defines a probability distribution for all the possible non-probabilistic programs it contains. Each single possible set of

Possible Worlds			
edge(1,2)	**edge(1,3)**	**edge(2,3)**	**Probability**
false	false	false	$(1-0.3)\cdot(1-0.7)\cdot(1-0.4)=0.126$
false	false	true	$(1-0.3)\cdot(1-0.7)\cdot 0.4=0.084$
false	true	false	$(1-0.3)\cdot 0.7\cdot(1-0.4)=0.294$
false	true	true	$(1-0.3)\cdot 0.7\cdot 0.4=0.196$
true	false	false	$0.3\cdot(1-0.7)\cdot(1-0.4)=0.054$
true	false	true	$0.3\cdot(1-0.7)\cdot 0.4=0.036$
true	true	false	$0.3\cdot 0.7\cdot(1-0.4)=0.126$
true	true	true	$0.3\cdot 0.7\cdot 0.4=0.084$
			$\sum_{i=1}^{N} p_{world_i}=1.000$

Table 2.1: All possible worlds for the ProbLog program of Figure 2.1. The possible worlds that satisfy the query: `path(1,3)` are highlighted.

probabilistic facts of the ProbLog program is called a **possible world**. The Table 2.1 presents all possible worlds for the ProbLog program of Figure 2.1. One can notice that having only three different probabilistic facts generates eight possible worlds. Actually the possible worlds of a ProbLog program are exponential in the number of probabilistic facts (2^N where N the number of probabilistic facts).

Definition 2. *Probability of Possible World: The probability of a possible world equals to the product of the probability of all probabilistic facts in $L_{true} \subseteq L_T$ that are true in the possible world and 1-probability of all probabilistic fact in $L_{false} \subseteq L_T$ that are false in the possible world.*

$$P_{world} = \prod_{pf_i \in L_{true}} p_i \cdot \prod_{pf_j \in L_{false}} (1-p_j) \tag{2.1}$$

where $L_{true} \cap L_{false} = L_T$ and $L_{true} \cup L_{false} = \emptyset$.

Lemma 1. *Sum of all Possible Worlds: The sum of the probabilities of all possible worlds equals to $\sum_{w_i \in Worlds} P_{w_i} = 1.0$.*

2.2 Success Probability

One of the most important tasks for ProbLog is to calculate the success probability of a query. In ProbLog, inquiring the success probability of a query means asking for the probability that a randomly selected possible world

satisfies that query. Such worlds contain the probabilistic facts needed to satisfy the query, but can also contain many more probabilistic facts.

Definition 3. *Success Probability: The success probability $P_s(q|T)$ of a query q is the summation of the probabilities of all possible worlds for which there exists a substitution θ such that $q\theta$ is entailed by T, i.e.,*

$$P_s(q|T) = \sum_{w_i \in Worlds} P(q|w_i) \cdot P_{w_i} \tag{2.2}$$

where $P(q|w_i) = 1.0$ if there exists a substitution θ such that $w_i \cup BK \models q\theta$ and $P(q|w_i) = 0.0$ otherwise.

Equation (2.2) states that we are able to calculate the success probability of a query by summing the probabilities of all worlds that satisfy the query. As the number of subprograms to be considered is exponential in the number of probabilistic facts, this approach quickly becomes infeasible with increasing problem size.

For example, using the graph of Figure 2.1 the success probability of the query `path(1, 3)` can be calculated by summing all the possible worlds where the query is satisfied. Observing Table 2.1 we find the five highlighted worlds to satisfy the query `path(1, 3)`. Summing the probabilities of each world gives us the success probability of the query `path(1, 3)`, in this case: $0.294 + 0.196 + 0.036 + 0.126 + 0.084 = 0.736$.

2.3 Knowledge Compilation Approaches

In ProbLog one calculates the success probability using several different inference methods. We can separate all inference methods in two big different types. The first type compiles Boolean formulae to a structure that the probability calculation is easy. Inference methods like these are `problog_exact` [35], `problog_kbest` [34], `problog_low` [34], `problog_delta` [16], `problog_threshold` [34]. All these methods follow the following steps to prove a query and calculate the probability of its success.

1. **SLD resolution** is used to prove the query and collect Boolean formulae that represent the possible worlds.
2. **Boolean formulae preprocessing** is used to optimize the collected Boolean formulae.

3. **ROBDD compilation** is used to compile all collected Boolean formulae in Reduced Order Binary Decision Diagram (ROBDD).

The second type of inference methods do not compile any Boolean formulae in a structure. There are several such approaches which we present in Section 2.4. Also there are knowledge compilation approaches that use different structures than ROBDD for compilation, those approaches follow a similar structure for inference like the three step one we present here. Figure 2.2 shows visually the three steps. We later use the same diagram to illustrate where our contributions affect ProbLog.

2.3.1 SLD Resolution

As previously mentioned the number of possible worlds is exponential to the number of probabilistic facts. The naive approach of enumerating all possible worlds and then summing the ones that satisfy the query quickly becomes computationally intractable. For that reason ProbLog uses different strategies to calculate the success probability of a query.

Instead of collecting possible worlds, one can collect partial worlds, called proofs, that contain the minimally needed probabilistic facts to prove the success of a query. For example instead of collecting all the possible worlds that have `edge(1,3)=true`, one can collect the proof `edge(1,3)` representing four possible worlds in one proof. Similarly, for the other explanation of query satisfaction one can collect the proof `edge(1,2),edge(2,3)` to represent the two possible worlds.

To collect proofs ProbLog takes advantage of the proving mechanism of Prolog. Through SLD resolution Prolog enumerates all proofs for a query. For our example the generated SLD tree of proving the query `path(1,3)` can be seen at Figure 2.3. Then ProbLog employs a reduction to propositional formula in disjunctive normal form (DNF) for the collected proofs. As stated earlier, probabilistic facts can be seen as random variables, implying that we are able to represent a proof as a conjunction of such facts. The set of all proofs then are represented as a disjunction, producing a DNF formula. The success probability then corresponds to the probability of this formula being true.

Still, the number of proofs a query might have in the worst case scenario is exponential. Efficiently collecting those proofs and being able to represent them and as a consequence represent the DNF formula in a compact data structure is crucial.

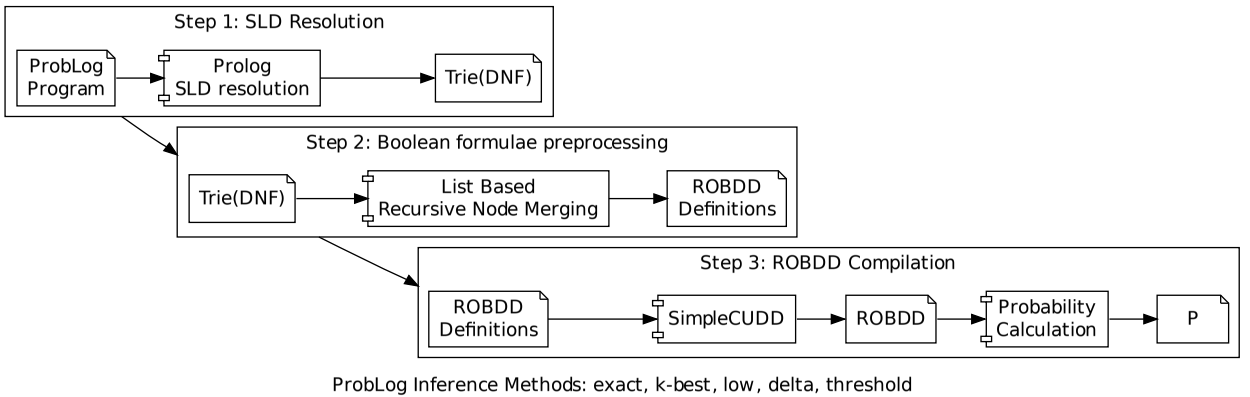


Figure 2.2: The diagram of the three steps of ProbLog knowledge compilation methods.

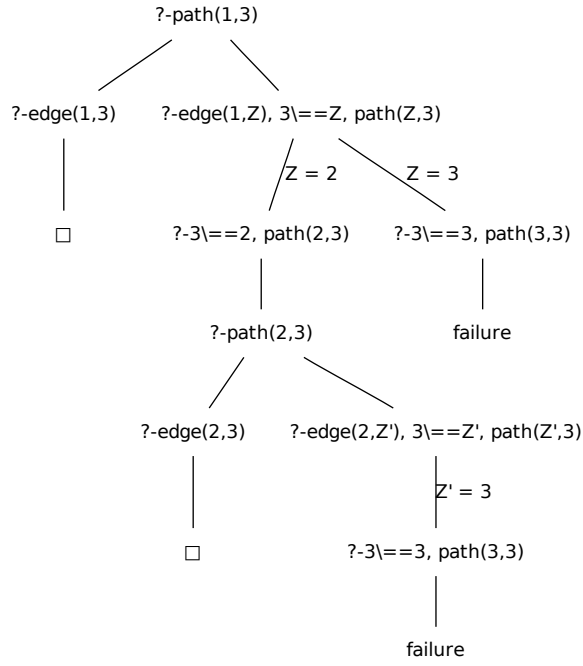


Figure 2.3: The complete Prolog SLD tree when proving the query: `path(1,3)` of the ProbLog program of Figure 2.1. The squares represent success.

2.3.2 Tries

To address the issue of representing the proofs efficiently, ProbLog uses **tries**. Tries are a data structure used for compact dictionary representation and fast word finding. Further, in the logic programming society tries are widely used for indexing terms for tabling. ProbLog uses tries to compactly store terms that represent proofs. Taking advantage of the depth first nature of SLD resolution, the proofs are constructed with having common prefix part. Tries, as a prefix sharing structure, compacts the common parts of proofs saving memory resources.

Back to our example, we present the collected proofs stored in a trie in Figure 2.4.

Collected proofs: $e(1,3) \quad \vee$
$e(1,2) \wedge e(2,3)$

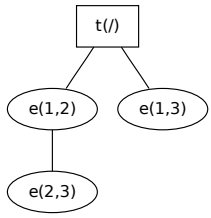


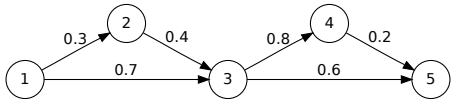
Figure 2.4: Collected proofs and generated trie for the ProbLog program of Figure 2.1. We abbreviate the probabilistic facts $edge/2$ as $e/2$.

To further illustrate the benefits of tries for collecting the proofs we use the slightly bigger graph of Figure 2.5(a). For this graph the query `problog_exact(path(1, 5), P)`, collects four proofs that represent the DNF Boolean formulae shown at Figure 2.5(b), we will abbreviate the query as `path(1,5)`. Using a trie to store these proofs results in the trie shown at Figure 2.5(c). In this trie we can see that some nodes are re-used instead of appearing twice. Nodes `e(1,2)`, `e(2,3)` and `e(1,3)` are a common prefix among the proofs. Instead of storing them twice the trie structure discovers this commonalities and saves space. Later we present how ProbLog takes further advantage of these commonalities to improve the efficiency.

2.3.3 Boolean Formulae Preprocessing

Up to here we have described how ProbLog has collected the set of proofs in a trie structure that represents a DNF Boolean formulae. The next step in ProbLog execution is the preprocessing of the Boolean formulae to an 'optimal' form that will contain less redundant operations. For example the Boolean formulae of Figure 2.5(b) could be written like: $((e(1,2) \wedge e(2,3)) \vee e(1,3)) \wedge ((e(3,4) \wedge e(4,5)) \vee e(3,5))$ to save six operations.

In order to discover possibly redundant operations ProbLog takes advantage of the trie prefix sharing. The algorithm **Recursive Node Merging** which is presented in detail in Chapter 4 takes advantage of some suffix sharing. ProbLog implements three more algorithms namely **Naive**, **Decomposition** and **Depth Breadth Trie**.

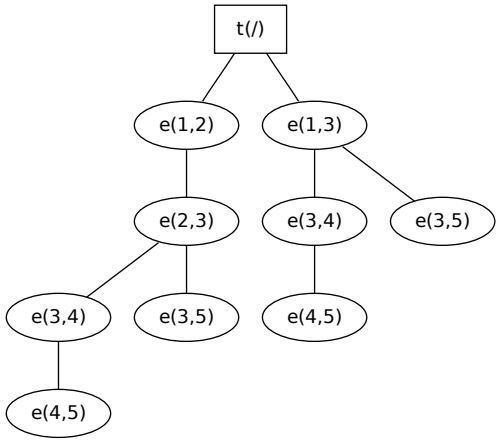



```
0.3::edge(1, 2).
0.7::edge(1, 3).
0.4::edge(2, 3).
0.8::edge(3, 4).
0.6::edge(3, 5).
0.2::edge(4, 5).
```

(a) A probabilistic graph without cycles.

$$
\begin{aligned}
&e(1,2) \wedge e(2,3) \wedge e(3,4) \wedge e(4,5) \quad \vee \\
&e(1,2) \wedge e(2,3) \wedge e(3,5) \quad \vee \\
&e(1,3) \wedge e(3,4) \wedge e(4,5) \quad \vee \\
&e(1,3) \wedge e(3,5)
\end{aligned}
$$

(b) The collected proofs for the graph by the query: `path(1,5)` in their DNF Boolean formula form.



(c) The collected trie for the graph by the query: `path(1,5)`.

Figure 2.5: The graph together with the Boolean formula in DNF form and the collected trie by the query: `path(1,5)`, used as an example.

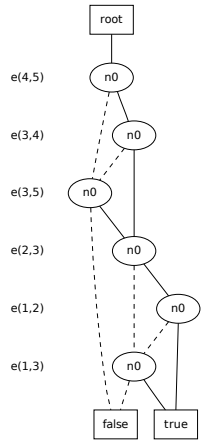


Figure 2.6: The ROBDD representing the Boolean formula of Figure 2.5(b).

2.3.4 Reduced Order Binary Decision Diagrams (ROBDDs)

Reduced Order Binary Decision Diagrams (ROBDDs) are a very widely used data structure in computer science. ROBDDs provide a compact way to represent Boolean formulae. Also ROBDDs maintain some important properties which are required for our usage. A ROBDD is an acyclic rooted graph. A ROBDD might consist of nodes that represent Boolean variables; two different types of outgoing edges from each node represent the two possible assignments of the Boolean variables; and finally, up to two leaf nodes that represent the constants **true** and **false**.

Figure 2.6 presents the ROBDD of the Boolean formula of Figure 2.5(b). Each layer of nodes represent a Boolean variable, a layer can have more than one node annotated $n0$, $n1$, etc.; the Boolean variable the nodes belong to can be seen on the left side. Every node has two outgoing edges: one continues which represents the true assignment of the Boolean variable and one dashed that represents the false assignment of the Boolean variable. Finally, the two leaf nodes are shown in rectangle shape stating the state of the formula for each distinct assignment of variables. More formally:

Definition 4 (ROBBD)**.** *A Reduced Order Binary Decision Diagram (ROBBD) is a rooted, directed acyclic graph with:*

- *One or two terminal nodes of out-degree zero labelled false (0) or true (1).*

- *A set of variable nodes u of out-degree two. The two outgoing edges are given by two functions $low(u)$ and $high(u)$. Each variable node u is associated with a variable $var(u)$.*
- *All paths through the graph respect a given linear variable ordering $x_1 < x_2 < \cdots < x_n$.*
- *No two distinct nodes u and v have the same variable name and low- and high-successor, i.e., $var(u) = var(v) \wedge low(u) = low(v) \wedge high(u) = high(v) \rightarrow u = v$.*
- *No variable node u has identical low- and high-successor, i.e., $low(u) \neq high(u)$.*

For further details on ROBDDs, we refer to [1, 3, 9].

2.3.5 Boolean Formulae Compilation and SimpleCUDD

In order to compile Boolean Formulae in ROBDDs, ProbLog uses the Colorado University Decision Diagrams (CUDD)[1] [80] framework. CUDD was chosen because of its efficiency and completeness as a library for generating successfully ROBDDs. The main difficulty in using CUDD is that both the ROBDD generation and the traversing operations need to be encoded in a C[2] program and thereafter compile it. For that reason one is forced to coding a different program for each separate ROBDD that one wants to generate and traverse. For ProbLog that boils down to a new C program for each single query needed to be calculated.

In order to address the above problem we developed an application in C named SimpleCUDD[3] [40, 41], that interfaces CUDD to the user, in our case ProbLog. SimpleCUDD focuses on providing a simple interface and minimizing the resource cost for communicating the user's instructions to CUDD. SimpleCUDD uses a scripting language to describe Boolean formulae to be compiled in ROBDDs. Also, it has a collection of extra features that are useful for ProbLog and any similar application that needs to add extra information to ROBDD nodes. It is not in the scope of this document to describe SimpleCUDD in depth. Through the rest of the document we present SimpleCUDD ROBDD scripts and for that reason we briefly explain some of the syntax here.

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

[2] CUDD also includes a C++ environment which is simpler to use but still requires to code everything in C++.

[3] `https://lirias.kuleuven.be/handle/123456789/253405`

2.3.6 Syntax of the Script Language

The SimpleCUDD package contains a parser that uses the following grammar to describe a Boolean formula that is to be compiled to a ROBDD. The grammar has: operators, variables, intermediate steps, constants, return result and comments.

Operators

SimpleCUDD script language defines the operators: (=, ~, *, +, #).

The operator (=) is used to define assignment. It can appear only once in each instruction line.

The operator (~) represents the negation. It is used as the negation for variables, intermediate steps, constants and even the negation of operators.

The operator (*) represents the logical AND. If it appears combined with the negation operator (~*) the combined operator represents the logical $NAND$.

The operator (+) represents the logical OR. If it appears combined with the negation operator (~+) the combined operator represents the logical NOR.

The operator (#) represents the logical XOR. If it appears combined with the negation operator (~#) the combined operator represents the logical $XNOR$.

An important limitation of the script language is that there are no proper operator priorities between AND, OR and XOR. This results in resolving the operation in the order of appearance. Similarly the syntax does not allow the use of brackets to prioritize the operation. Finally, it is important to mention that the parsing works from right to left instead of left to right.

The described limitations might limit the ways the scripting language can be used, but do not limit its expressive power. The scripting language using the intermediate steps can enforce priority.

ROBDD Variables

A ROBDD variable can consist almost of any symbol. The syntax of ROBDD variables has the following naming limitations:

1. A ROBDD variable name can not start with the capital letter L.
2. A ROBDD variable name can not have any of the operator symbols.
3. It is suggested that ROBDD variables should not use the symbols '<>'.

Examples of valid ROBDD variable names are: *X1, x1, 123, foo, _abc.* Assignments to ROBDD variables are not allowed. The ROBDD variables are the Boolean literals of the formula that is under transformation to become a ROBDD. The "variables" that represent parts of the ROBDD, are called intermediate steps.

Intermediate Steps

Intermediate steps are required to refer to stored parts of the ROBDD. Then by combining these parts the complete ROBDD is constructed. Intermediate steps need to be distinguished from variables. The correct syntax for intermediate steps is as followed:

1. An intermediate step should start with a capital L.
2. An intermediate step should have after the capital L an integer number.
3. The number must be from 1 up to the defined intermediate steps variable set at the file header.

Examples of valid intermediate names are: L1, L2. Intermediate steps are used for ROBDD assignment. After assigning a ROBDD at an intermediate step, that step can be used as a ROBDD variable. Reassigning a new ROBDD to the intermediate step is not allowed.

Constants

The are two constants:

1. TRUE, this is the logical true and should be written capitalized.
2. FALSE, this is the logical false and should be written capitalized.

The constants are used in the same way as the variables.

	Generated ROBDD definitions of Figure 2.4		Figure 2.5(b)
1	`L1=e(1,3)`	1	`L1=e(1,2)*e(2,3)*e(3,4)*e(4,5)`
2	`L2=e(1,2)*e(2,3)`	2	`L2=e(1,2)*e(2,3)*e(3,5)`
3	`L3=L1+L2`	3	`L3=e(1,3)*e(3,4)*e(4,5)`
		4	`L4=e(1,3)*e(3,5)`
		5	`L5=L1+L2+L3+L4`

Table 2.2: ROBDD definitions generated by the naive approach for the Boolean formulae of Figures 2.4 and 2.5(b) respectively.

The Return Result Syntax

At the end of a script a return result needs to be declared. The return result can be any one or more[4] of the previously assigned intermediate steps, ROBDD variables or constants. For example: L3. For the purpose of this text we can skip the return result from the ROBDD definitions by assuming always that only the last definition is returned.

Definition 5 (ROBDD Definition)**.** *A ROBDD definition is a complete line of a ROBDD script.*

We present two example ROBDD scripts in Table 2.2 that correspond to the Boolean formulae of Figures 2.4 and 2.5(b) respectively.

Using Boolean formulae preprocessing, as mentioned before, the ROBDD definitions of the Boolean formula of Figure 2.5(b) would be optimized to match the ones presented at Table 2.3. In this specific example preprocessing reduces the amount of operations from 11 to 6.

2.3.7 ROBDD Generation and Probability Calculation

In addition than easily generating ROBDDs, SimpleCUDD also provides functionality to efficiently traverse ROBDDs and perform calculations over their nodes. At Algorithm 2.1 we present a dynamic programming algorithm used by ProbLog to calculate the probability of an annotated ROBDD. This algorithm was first presented in [16]. SimpleCUDD provides the functionality for annotating variables, to traverse and remember traversed nodes. For these purposes it uses indexed tables, and a hash table when necessary.

[4]Returning a forest of ROBDDs in practice. This is a convenient feature for many ProbLog applications which are not in our scope.

Generated ROBDD definitions	
1	`L1=e(3,4)*e(4,5)`
2	`L2=L1+e(3,5)`
3	`L3=L2*e(2,3)`
4	`L4=L3*e(1,2)`
5	`L5=L2*e(1,3)`
6	`L6=L4+L5`

Table 2.3: ROBDD definitions of the Boolean formula of Figure 2.5(b) generated by performing Boolean formulae pre-preprocessing.

Algorithm 2.1 A recursive dynamic programming depth first algorithm to traverse and calculate in linear time the probability of an annotated ROBDD.

input The assumed root node $Node$ of a ROBDD, a data structure $Vars$ that contains the annotated information for the ROBDD variables, and a history data structure His which initially is empty and is used to store the traversed nodes and calculated probabilities.

output The calculated probability $Prob$ of the ROBDD and a $History$ with the calculation of the probability for each traversed node.

function CalcProbability($Node, Vars, His$)
  if $Node = TRUE$ **then**
    return $(1.0, His)$
  if $Node = FALSE$ **then**
    return $(0.0, His)$
  if $Node \in His$ **then**
    return (HistoryValue($His, Node$), His)
  $LowNode$:= LowNodeOf($Node$)
  ($LowNodeProb, His$) := CalcProbability($LowNode, Vars, His$)
  $HighNode$:= HighNodeOf($CurrentNode$)
  ($HighNodeProb, His$) := CalcProbability($HighNode, Vars, His$)
  $VarProb$:= VariableAnnotatedValue($Vars$, Variable($Node$))
  $Prob := VarProb \cdot LowNodeProb + (1 - VarProb) \cdot HighNodeProb$
  AddToHistory($His, Node, Prob$)
  return $(Prob, His)$

SimpleCUDD provides us with and indexed table that is used for easy access to the values of the annotated ROBDD variables; to easily traverse the ROBDD it provides two functions LowNodeOf and HighNodeOf; and finally, for remembering traversed nodes it provides an efficient hash table His that uses a variable index to remember nodes traversed and calculated values.

The Algorithm 2.1 as can easily be shown has a worst case complexity $O(N+M)$ where N is the number of nodes and M the number of edges. Indeed the memoization of each node results in calculating every node only once.

2.4 Other Approaches

Further than the compilation approaches, ProbLog uses several other methods for calculating the probability of a ProbLog program. Namely, a Monte Carlo Markov Chain approach, a Monte Carlo approach that samples programs `problog_program_sampling` [34], a Monte Carlo approach that samples explanations of DNF formulae `problog_DNF_sampling` [78], and a compilation approach using weighted CNF's instead of annotated ROBDDs [18]. In this thesis we later focus on `problog_program_sampling` and `problog_DNF_sampling`.

There are other extensions of ProbLog that use inference or extend inference like learning [26,27], decision theory ProbLog [86], algebraic ProbLog [38], continues distributions [25]. These issues are out of the scope of this thesis.

2.4.1 Approximate Inference: Program Sampling

An alternative approach to inference is the use of Monte Carlo methods, that is, to use the ProbLog program to generate large numbers of random subprograms and to use those to estimate the probability. More specifically, such a method proceeds by repeating the following steps:

1. sample a logic (sub)program L from the ProbLog program
2. search for a proof of the initially stated query q in the sample $L \cup BK$
3. estimate the success probability as the fraction P of samples which hold a proof of the query

The implementation of this approach for ProbLog, as described in [35], takes advantage of the independence of probabilistic facts to generate samples lazily while proving the query, that is, sampling and searching for proofs which are

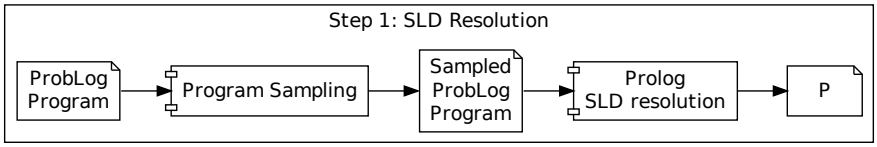


Figure 2.7: The diagram for the program sampling inference method.

interleaved. To assess the precision of the current estimate P, at each m samples the width δ of the 95% confidence interval is approximated as

$$\delta = 2 \cdot \sqrt{\frac{P \cdot (1 - P)}{N}} \tag{2.3}$$

If the number of samples N is large enough the interval of confidence becomes smaller, and the probability that the estimate is close to the true probability of the query increases.

Figure 2.7, shows the diagram for the program sampling inference method.

2.4.2 Approximate Inference: DNF Sampling

Program Sampling as presented in [35] generates samples by exploring the SLD tree, which can be expensive if there are many failing derivations. DNF Sampling for ProbLog as presented in [78] focuses on sampling possible worlds containing a proof of the query of interest. This method first constructs the DNF for the query as in exact inference, and then applies the Monte-Carlo algorithm of Karp and Luby [32] to estimate the probability of the DNF.

DNF Sampling associates each possible world w to the first conjunction that is true in w. Samples are generated by first sampling a conjunction c_i with probability $P(c_i)/\sum_{j=1}^{n} P(c_j)$, and then generating a possible world by setting the truth values of the variables in c_i such that c_i is true, and sampling truth values for remaining variables. As for each pair (c_i, w), the probability of sampling that pair is $P(w)/\sum_{j=1}^{n} P(c_j)$ and thus proportional to $P(w)$, given a sufficient number N of samples, the fraction $N_{accepted}/N$ of positive samples approaches $P(F)/\sum_{j=1}^{n} P(c_j)$, and we can thus estimate $P(F)$ as $\sum_{j=1}^{n} P(c_j) \cdot N_{accepted}/N$.

Algorithm

Let $F = C_1 \vee \ldots \vee C_m$ be a propositional DNF for query q in the ProbLog program T, where the C_i contain neither contradictions nor multiple occurrences of the same variable. We denote possible worlds – truth value assignments to all random variables – by w. The space from which samples are drawn is defined as $U = \{(w, i)|w \models C_i\}$, and we associate each possible world w with the first conjunction that is true in w, that is, the samples that will be accepted are those from $A = \{(w, i)|w \models C_i \wedge \forall j < i : w \not\models C_j\}$. For each possible world w with $w \models F$, U thus contains a pair (w, i) for each C_i that is true in w, whereas A only contains the pair with minimal i. We define the sum of probabilities for DNF F as

$$S_T(F) = \sum_{i=0}^{m} \prod_{f_j \in C_i} p_j \tag{2.4}$$

Note that if conjunctions are mutually exclusive, $S_T(F)$ is equal to the probability of F being true, but it can be much higher in general.

DNF Sampling generates N samples in the following way

1. randomly choose C_i according to $P(C_i|T)/S_T(F)$
2. randomly choose a possible world w where C_i is true
3. increment $N_{accepted}$ if $(w, i) \in A$

The probability of formula F is then estimated as

$$P_{DNF}(q|T) = S_T(F) \cdot \frac{N_{accepted}}{N} \tag{2.5}$$

Note that depending on the structure of the problem and the value of $S_T(F)$, estimates based on small numbers of samples may not be probabilities yet, that is, they can 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 $S_T(F)$.

2.4.3 Convergence

DNF Sampling is an instance of the fully polynomial approximation scheme of Karb and Luby [32], that is, the number of samples required for a given

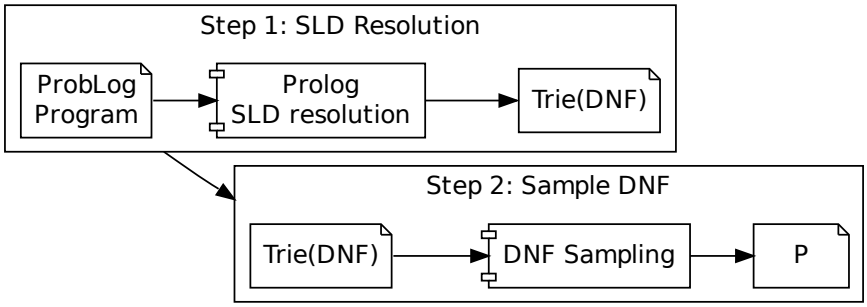


Figure 2.8: The diagram for the DNF sampling inference method.

level of certainty is polynomial in the input length (the DNF in our case). For formal detail, we refer to [32], and instead give a rough illustration here. The algorithm uses the normalization factor $S_T(F)$ from equation (2.4), therefore, in each sampling step, the probability that the i^{th} conjunction is sampled is $P(C_i)/S_T(F)$. It is then completed into a possible world according to the fact probabilities. This possible world will be accepted exactly if all conjunctions with smaller index are false in it. So the probability of sampling a specific C_i and a world which will be accepted for this conjunction is $P(C_i \wedge \neg DNF_{i-1})/S_T(F)$, where $DNF_i = \bigvee_{j=1\ldots i} C_j$. As each world can only be accepted by exactly one conjunction, the probability of sampling an arbitrary world that will be accepted is the sum of this probability over all conjunctions, and for N samples, the estimated number of accepted possible worlds and the corresponding probability estimate thus are:

$$E[N_{accepted}] = N \cdot \frac{P(C_1) + P(C_2 \wedge \neg DNF_1) + \ldots + P(C_n \wedge \neg DNF_{n-1})}{S_T(F)}$$

$$\begin{aligned} P_{estimated} &= \frac{E[N_{accepted}]}{N} \cdot S_T(F) \\ &= P(C_1) + P(C_2 \wedge \neg DNF_1) + \ldots + P(C_n \wedge \neg DNF_{n-1}) \end{aligned}$$

The last line corresponds to one way of solving the disjoint-sum-problem for the original DNF, which is exactly what the purpose of the algorithm is.

Figure 2.8, shows the diagram for the DNF sampling inference method.

2.4.4 Weighted CNF ProbLog

Another approach worth mentioning is the weighted CNF ProbLog [18]. In this approach ProbLog semantics and syntax is used but the three steps (SLD-resolution, Boolean formulae preprocessing and ROBDD compilation) of ProbLog have been significantly modified.

The first step, SLD-resolution has been renamed to **Grounding**. The step is modified to use Prolog's SLD resolution in order to collect the relevant ground logic program for a query with evidence instead of the derivations of a query. In order to achieve that the system uses the concept of a dependency set of a ground atom with respect to a ProbLog program as presented in [27].

The second step named **Conversion to CNF** is the equivalent of the Boolean formulae preprocessing step. The new system requires to convert the collected ground logic program into a CNF instead of keeping the Boolean formulae into its original form. In order to achieve that one could use Clark's completion if the collected logic program is acyclic. For logic programs that contain cyclic rules, the system uses two different approaches. The first approach named **Rule-based conversion to CNF** uses the approach presented in [30] and it is based on converting the rules that introduce cycles to rules with auxiliary atoms and then converting in a CNF formulae. The second approach used is called **Proof-based conversion to CNF** and uses nested tries, a recursive structure which is described in depth in Section 3.2.3 and a cycle breaking algorithm that is described in detail in Section 3.4. The resulting Boolean formulae is then converted in a CNF by using conversion rules from mathematical logic.

Finally, the third step named **sd-DNNF compilation** uses sd-DNNFs (Smooth Deterministic Decomposable Negation Normal Form) instead of ROBDDs as in the original ProbLog. For this step the system uses the CNF to sd-DNNF implementation presented in [49]. After the generation of the sd-DNNF a polynomial bottom-up algorithm is used to calculate the probability. These three steps are shown visually in Figure 2.9.

Let PS be a denumerable set of propositional variables. NNF is a directed acyclic graph where each leaf node is labelled with $true$, $false$, X or $\neg X$, $X \in PS$; and each internal node is labelled with $\wedge$ or $\vee$ and can have arbitrarily many children. An sd-DNNF is an NNF structure that satisfy three extra properties: **decomposability**, **determinism** and **smoothness**. The notion of determinism appears to be necessary (but not sufficient) for algorithms to perform model counting in polynomial time. Assessing the probability of a weighted Boolean formula is a variant of weighted model counting. As presented in [12] both ROBDDs and sd-DNNFs have the determinism property. Other similar structures that have determinism are: d-DNNF, FBDD and MODS. For

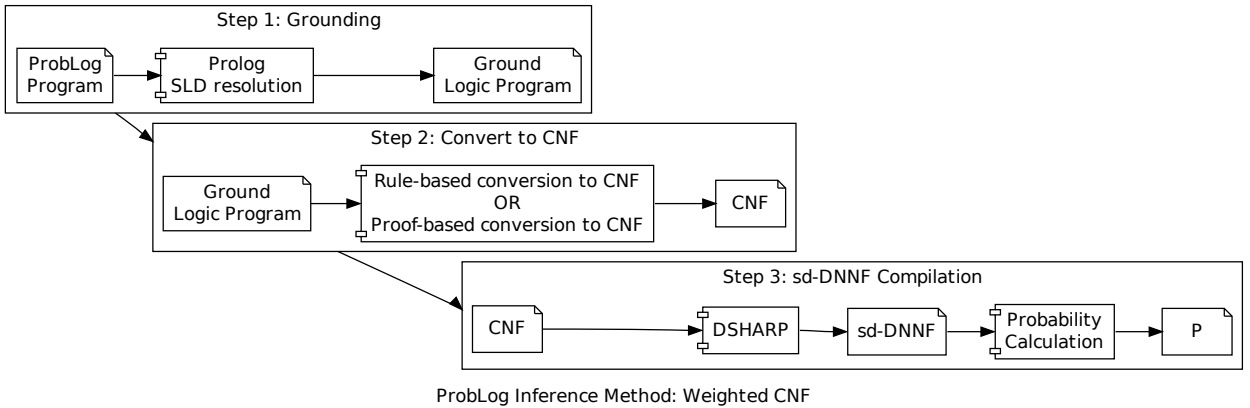


Figure 2.9: The diagram of the three steps for the weighted CNF ProbLog.

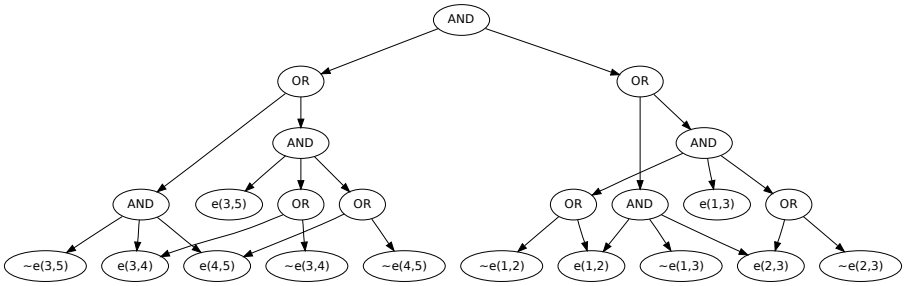


Figure 2.10: The equivalent sd-DNNF for the ROBDD of Figure 2.6 representing the Boolean formula of Figure 2.5(b).

more details on sd-DNNF, the reader can consult [12]. Figure 2.10 shows the equivalent sd-DNNF for the ROBDD presented in Figure 2.6.

2.5 Conclusion

We have briefly presented the probabilistic logic programming language ProbLog which uses the distribution semantics. We presented some details about its inference infrastructure, especially around the knowledge compilation approach that uses ROBDDs. We clearly stated three different stages in the inference process, namely: SLD resolution, Boolean formulae preprocessing, and Boolean formulae compilation. We explain in some detail the syntax for the scripting language that ProbLog uses to describe Boolean formulae. Finally, we mentioned further approaches to inference and briefly presented two approximation inference methods.

Chapter 3

Tabling of Probabilistic Logic Programs

The implementation of ProbLog [37] is based on the use of tries [19] and reduced ordered binary decision diagrams (ROBDDs) [1, 9]. The execution of ProbLog programs uses SLD-resolution to collect all the proofs for a query. In order to compute correct probabilities, ProbLog gathers for each successful proof of the query the list of probabilistic facts the proof uses. The lists of probabilistic facts of all different successful proofs are compactly represented by a trie. Such a trie is then considered to be a sum of products (a disjunction of conjunctions of probabilistic facts). ROBDDs are used to solve the disjoint sum problem and to obtain the correct probability of the query.

This chapter discusses the use of tabling for probabilistic logic programs. Tabling [5,11,84] is a very well known approach that reuses previous calculations instead of recomputing them, in order to improve the performance of some logic programs. Besides giving performance benefits, tabling also can be used for writing more declarative programs and to handle some programs that do not terminate because of execution cycles.

All three benefits of tabling for logic programs apply also to probabilistic logic programs. There has been a significant amount of research for tabling lately, which has provided many Prolog systems with tabling mechanisms. Furthermore, several different strategies and many different optimizations [82] have been developed.

The challenge is to find out how tabling can be combined with the ProbLog

execution mechanism. Tabling mechanisms are available in XSB [65, 83], YAP [70, 71] and other Prolog systems. The basic idea is to collect the answers of a tabled subgoal in a table and, when the subgoal is re-encountered, to reuse the tabled answers instead of computing them. As a consequence of this memoization, tabling ensures termination of programs with the bounded term-size property, i.e., programs where the size of subgoals and answers produced during an evaluation is less than some fixed number. In the case of ProbLog, tabling answers is not sufficient, as the proofs are needed too. Also cycles have to be dealt with correctly.

The main contributions of this chapter are the identification and realisation of the necessary tabling mechanism for ProbLog programs while respecting the current ProbLog optimizations, such as the exploitation of tries and the optimized translation of tries into ROBDDs. Through program transformation we adapt the selected tabled predicates when called, to construct their own trie, memorise it, and integrate it in the final trie.

ProbLog's motivating link discovery applications and other typical ProbLog programs have only ground goals. In order to table them, we represent the SLD proof tree as a set of **nested tries**. We implemented a light-weight dedicated tabling for ground goals that supports nested tries and obtained impressive time improvements for some classes of programs. By the virtue of the nested tries, we also realize suffix sharing and thus a substantial memory compaction.

Furthermore, we study in Section 3.4 how the modifications to the trie structure affect the next calculation steps, in particular the construction of the ROBDDs. In the case of non cyclic programs, our nested tries have important performance impact to the generation of ROBDDs. Tabling and nested tries are crucial in the tractability of the whole process. We get promising results and are able to answer a larger set of queries for typical ProbLog programs. In a probabilistic setting it is often the case that the same goals re-appear. For example for determining the weather on day N there are several cases to be considered and in several of them one needs to know the weather on the day before $(N-1)$, or the day before that $(N-2)$.

By adding **cycle detection**, path-finding programs, typical for link discovery, also benefit from the memoization. Unfortunately, the cycles are transferred to the nested tries; this results in propagating the work load to the generation of the ROBDD definitions. In order to tackle this inefficiency, we describe several novel optimizations that aim in improving the performance of generating and constructing ROBDD definitions. We see promising improvements from all these optimizations.

Experiments with typical statistical relational learning tasks show that our lightweight tabling boosts the performance of ProbLog with respect to both execution time and memory consumption. Alternatively, we have obtained similar results by using the standard YAP tabling mechanisms. In order to benefit directly from standard tabling, we handle probabilistic inference differently from what the ProbLog2010 implementation does using an abstracted ProbLog engine which is further explained in Chapter 6. Figure 3.1 presents graphically the ProbLog parts that are modified by tabling.

Our work is similar to the PRISM [73, 74] tabling mechanism, as both mechanisms are restricted to ground goals and both mechanisms are memorising all the proofs. PRISM assumptions such as exclusiveness and no cycles imply that PRISM computations are simpler, in the sense that they do not have to deal with the disjoint sum problem. PRISM contains a linear tabling system [75]. Only when PRISM is executing its learning algorithms, its tabling is extended to do something special, namely to build support graphs which represent the shared structure of explanations for an observed goal. The support graphs play a central role in efficient EM learning of PRISM programs. The proofs ProbLog needs to compute are somewhat similar to these explanations. One could compare ProbLog's nested tries with PRISM support graphs. Differences are that PRISM assumes the exclusiveness condition for the proofs, while ProbLog does not, and that ProbLog requires the handling of cycles as it is intended for link discovery in graphs.

Another very similar approach is the one from PITA [69], which is an other tabled inference method. In PITA, instead of handling the cycles, they authors table programs that do not introduce cycles for example, the path program with a member check. But differently from PRISM the system continues inference in a similar way as ProbLog (with the use of ROBDDs) by not assuming exclusiveness and solving the disjoint sum problem.

Another example of tabling that needs the memoization of proofs, is in the scope of justification [24]. In logic programming justification is referred to the derivation which justifies the success or failure of a goal. Justifications are mainly used for debugging logic programs. It is important to note that tabling in justification keeps only one proof [55] instead of all the proofs, and that it requires tabling of non-ground goals. Finally, [36] proposes tabling for another ProbLog inference method, namely Monte Carlo sampling.

The chapter is structured as follows. First, we briefly explain how tabling alters SLD to SLG resolution in Section 3.1. We present our running example and the nested tries and identify the necessary tabling support in Section 3.2. Then follows ProbLog tabling in Section 3.3 where we explain the usage of tabling and introduce the cycles in a probabilistic setting. Transforming the nested tries

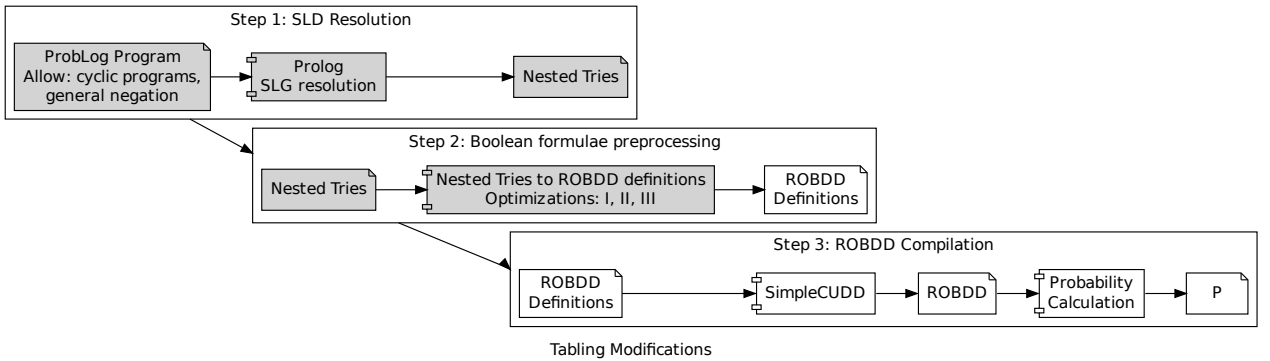


Figure 3.1: The diagram of the three steps of ProbLog with tabling. Shaded blocks mark the modifications from the block figure of Figure 2.2.

to ROBDD definitions and further to a sum of products efficiently is described in Section 3.4. The description of the implementation and the needed program transformation that realise ProbLog tabling are found in Section 3.5. Section 3.6 contains the experimental evaluation. And we finally conclude in Section 3.7.

This chapter is based on the publications [42, 43] and is extended to include the complete program transformation that realizes ProbLog tabling, as well as a second program transformation that uses the built-in tabling mechanism of the Prolog system. The chapter also describes several optimizations that further improve the performance of generating ROBDD definitions and briefly evaluates them experimentally.

3.1 SLG Resolution

A logic program that uses tabling alters the resolution mechanism from SLD to SLG. Tabling strategies either use a suspension/resumption mechanism or a linear tabling mechanism to build a forest of SLG trees [11], while in SLD resolution there is only a single tree.

In logic programming the programmer can choose to table a predicate. When a subgoal of the tabled predicate is encountered for the first time, most Prolog tabling mechanisms suspend the resolution of the parent goal and start proving the subgoal. As soon as a successful derivation for the subgoal is found the answer is memoized in a table and the execution of the parent goal is resumed. Note that if the subgoal fails, its failure is memoized. This subgoal is called the **generator** subgoal.

All other occurrences of the subgoal, after the first, are called **consumer** subgoals. For programs without cycles, the consumer subgoals are in a different derivation branch than that of the generator subgoal. This means that the subgoal has been evaluated either as failure or successful with at least one answer. These repeated occurrences are not evaluated but the result of their generator subgoal evaluation is used. In case of a cycle, the two subgoals share the same derivation branch. It is possible that the generator subgoal has already found and memoized a successful answer. In this case the consumer cyclic subgoal consumes that answer and succeeds. If no answer has been generated yet, then the consumer subgoal is suspended and waits until the generator subgoal either finds an answer or fails. Then the execution of the suspended consumer subgoal is resumed to consume the answer.

In tabling, new answers for a subgoal are generated only from the generator subgoal. A consumer that requires a new answer that has not been generated

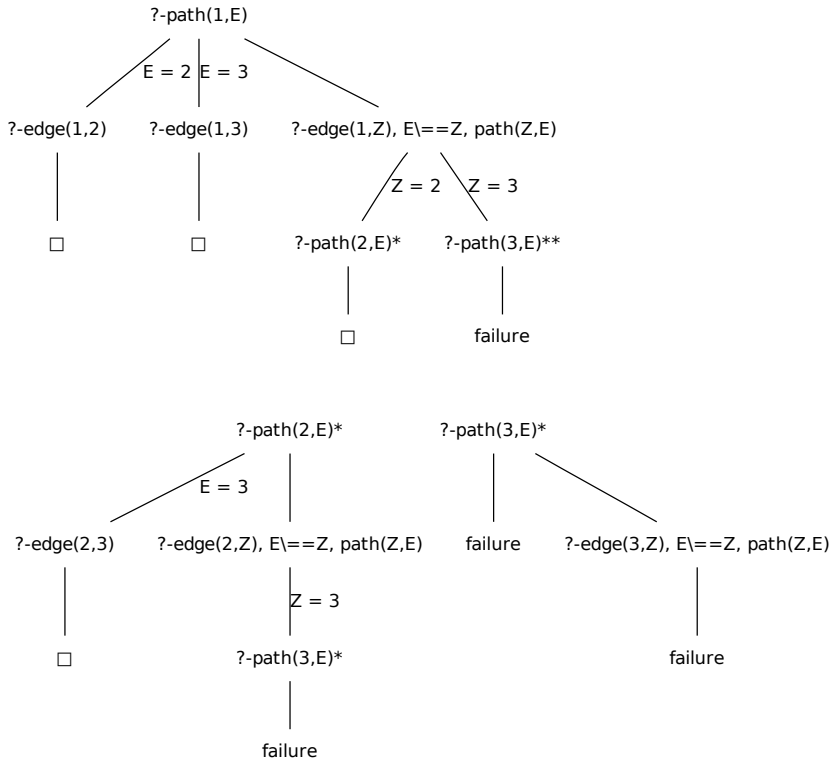


Figure 3.2: The complete Prolog SLG forest when proving the query: `path(1,3)` of the ProbLog program of Figure 2.1 at page 8. The squares represent success. * Represents the initiation of a new SLG tree. ** Represents a reused result.

yet returns the control to the generator subgoal. For this to be feasible a **completion mechanism** for each tabled predicate is kept which marks if a generator subgoal has still unexplored derivations. Figure 3.2 presents the SLG trees that are generated from proving query `path(1,E)` of the ProbLog program shown in Figure 2.1 at page 8.

There are two main strategies for collecting answers with tabling, namely local and batch evaluation [65, 83]. Local and batch evaluation differ in that batch evaluation eagerly returns answers while local evaluation may not return any answers to a parent goal until the subgoal is as far as it can be evaluated.

```
path(X, Y):-
  path(X, Y, [X]).
path(X, Y, _):-
  edge(X, Y).
path(X, Y, V):-
  edge(X, Z),
  Y \== Z,
  \+ member(Z, V),
  path(Z, Y, [Z|V]).
```

(a) Non-tabled path/2 predicate.

```
:- problog_table path/2.

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

(b) Tabled path/2 predicate.

Figure 3.3: Non-tabled & tabled path/2 predicate.

In logic programs, tabling only needs to remember whether the goal succeeded or failed, and, in the case of non ground goals, the answers with which it succeeded. Tabling a probabilistic logic program imposes the extra challenge that each proven subgoal has probabilistic information that needs to be remembered.

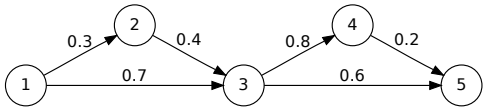
3.2 ProbLog & Tabling Preliminaries

Before explaining the exact tabling mechanism used in ProbLog, we first present the running example for tabling. We then need to mention the ground goal assumption we take and explain what a nested trie is. Finally, we present the technical details about the tabling mechanism.

3.2.1 ProbLog Tabling Example

It is common practice in tabling research to use the path predicate as a running example. Also for ProbLog tabling we use the path predicate of Figure 3.3 but with probabilistic graphs instead of deterministic ones. Figure 3.3(b) presents the tabled version of the path predicate of Figure 3.3(a). The tabled path/2 relies on tabling to handle any cycles introduced by the graph structure, while the non tabled version needs to keep a list V of visited nodes and to do a member check.

To explain all the features of tabling we use two different graphs, the first graph is presented in Figure 3.4 together with the probabilistic edge/2 facts that represent it in ProbLog; and the second graph is presented in Figure 3.5 together with the extra probabilistic edge/2 facts required to represented it in




```
0.3::edge(1, 2).
0.7::edge(1, 3).
0.4::edge(2, 3).
0.8::edge(3, 4).
0.6::edge(3, 5).
0.2::edge(4, 5).
```

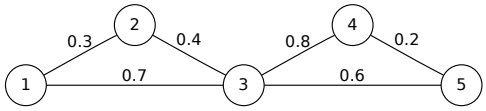
Figure 3.4: A probabilistic graph without cycles.

ProbLog. The first graph does not contain any cycles while the second one does.

Through the chapter we use the following two ProbLog queries: `problog_exact( path(1, 3), P)` and `problog_exact(path(1, 5), P)` to illustrate different aspects of the behaviour with tabling; we abbreviate the queries as `path(1,3)` and `path(1,5)` respectively. The answer to the first query is $P = 0.736$ and the answer to the second query is $P = 0.488704$. The answer to these specific queries is the same for both the directed and undirected graph. The reason is that the undirected graph does not add any new paths for those specific queries. As shown later, the difference between the two graphs is the introduction of cycles.

3.2.2 Ground Goal Assumption

Typical ProbLog goals are ground; indeed, in a probabilistic framework, one is interested in the probability that a goal can be proven, rather than in what the answers are. While later we show how to generalize tabling for non ground goals and return its answers, in this first part we focus on ground goals.




```
0.3::edge(2, 1).
0.7::edge(3, 1).
0.4::edge(3, 2).
0.8::edge(4, 3).
0.6::edge(5, 3).
0.2::edge(5, 4).
```

Figure 3.5: An undirected probabilistic graph introducing cycles.

3.2.3 Nested Tries

ProbLog collects all the SLD refutations of a query as lists of probabilistic facts in a trie. The probabilistic part of ProbLog programs creates new requirements for the tabling mechanism. Our tabling builds a forest of SLG trees [11], one for the original query and one for each tabled subgoal. For each tabled goal, we need to memoize its contribution to the trie. To achieve this, we break up a single trie into a set of nested tries. The **nested** trie of a goal represents the successful proofs of the goal just as a normal trie does, but the parts of the trie that are contributed by other tabled subgoals are replaced by a reference to the trie of that subgoal.

When ProbLog uses tabling while proving the topquery, it constructs a set of nested tries. This set of nested tries is equivalent with the trie that a non-tabled program would generate: it contains all the information about the complete successful proofs, the SLD refutations, of the topquery.

To better illustrate the differences between a single trie and a forest of nested tries we use the query `path(1,5)` for the non-cyclic graph of Figure 3.4. We present in Figure 3.6 the collected trie for the non tabled evaluation of our query and we present in Figure 3.7 the collected forest of nested tries for the tabled evaluation of the same query. We abbreviate the probabilistic facts $edge/2$ as $e/2$ and the subgoals $path/2$ as $p/2$. With $t/1$ we represent the trie of the relevant subgoal and by $t(/)$ the trie of the topquery.

An attentive reader can notice the trie prefix sharing in Figure 3.6 where the first part of the proofs is being reused. Also, notice that the suffix of the proofs is similar but tries can not detect this similarity and fail to perform reuses there. On the other hand the nested tries allow us to easily detect this suffix similarity which comes from proving the same goal twice. And this suffix is reused as can be seen in Figure 3.7 where the trie for the subgoal `path(3, 5)` is being referred to two places.

The original trie shown in Figure 3.6 is equivalent with the nested tries of Figure 3.7. One could completely reconstruct the original trie by following in a depth first matter the nested trie entries in the nested tries. For example, the leftmost branch of the trie is reconstructed if one follows the leftmost branches in the nested tries. Starting from the topquery trie we need to follow the trie $t(p(1,5))$, following the leftmost branch of trie $t(p(1,5))$ we get the first node $e(1,2)$ and find a reference to the trie $t(p(2,5))$. Continuing this process we construct the proof $e(1,2), e(2,3), e(3,4), e(4,5)$ which is identical with the leftmost entry of the original trie.

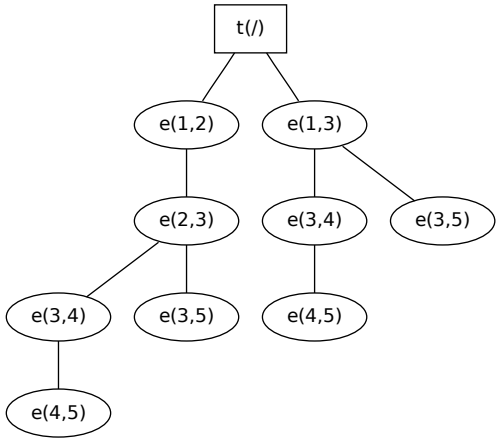


Figure 3.6: The collected trie for the graph of Figure 3.4 using the predicate of Figure 3.3(a) by the query: `path(1,5)`.

3.3 ProbLog Tabling

Definition 6 (Probabilistic Predicate)**.** *We call probabilistic predicate, any predicate that directly or indirectly uses at least one probabilistic fact in any of its definitions.*

As shown earlier in Prolog a predicate must be indicated by the programmer to be tabled with the use of the directive `table/1`[1]. ProbLog tabling follows a similar approach but uses the directive `problog_table/1`. The directive `problog_table/1` must be used to properly table a probabilistic predicate. One could use the ProbLog directive for tabling purely logical predicates or use the original Prolog `table/1` directive. While both would work the latter is more efficient as it does not perform the extra bookkeeping that probabilistic predicates require. To table a probabilistic fact, again one should use ProbLog's tabling directive. Like in Prolog tabling, using tabling on probabilistic facts does not have any significant performance impact.

When ProbLog encounters a generator subgoal, it creates a table entry for the subgoal with an empty nested trie, and starts proving it. As soon as a successful

[1]For example: `:- table path/2`

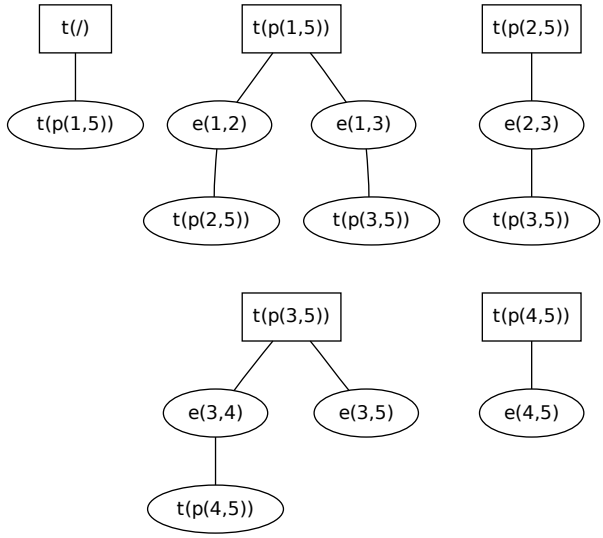


Figure 3.7: The collected nested tries for the graph of Figure 3.4 using the predicate of Figure 3.3(b) by the query: `path(1,5)`.

proof for the subgoal is found, it is added to its nested trie. Note that if the subgoal fails, no proof is added and the nested trie remains empty.

ProbLog eagerly collects the proofs for the generator goals implementing a local tabling strategy. For programs without cycles, tabling deals completely with a generator subgoal before the parent goal is resumed and it is known whether the subgoal failed or succeeded. If the generator subgoal fails, resumption of the parent goal fails its current proof. If the generator subgoal succeeds, on resumption of the parent goal, a reference to the nested trie of the tabled subgoal is added to the current proof of the parent goal. For occurrences of consumer subgoals, tabling avoids re-computation by adding a reference to the appropriate nested trie in the current proof.

3.3.1 Cycles in a Probabilistic Setting

Until now, we presented how tabling avoids repeated subcomputations. Tabling typically also deals with cycles. The probabilistic context determines how to

handle cycles.

An attentive reader might indeed have noticed that the path program of Figure 3.3(a) is a version that encodes cycle detection explicitly by using `\+ member(X, V)`. That version does not benefit from tabling as almost all the calls to `path/3` are different. The `path/2` program in Example 3.3(b) has no code to detect cycles. We use this program and the undirected graph of Figure 3.5 to explain how tabling should support cycle handling in a probabilistic framework.

For the query `path(1,3)` on the graph of Figure 3.5 with the program of Figure 3.3(b), we collect the following two proofs: `edge(1,2), edge(2,3)` and `edge(1,3)`. Trying to prove the query `path(1,3)` with the program of Figure 3.3(b) without tabling, one will enter in an infinite cycle when asking for multiple answers, such as the one between nodes 1 and 2 due to `edge(1,2)` and `edge(2,1)`. ProbLog's calculation of success probability requires the collection of all proofs and to that end it searches all branches of the SLD tree of a query which means that in the presence of a cycle the execution gets caught in a loop. The SLD tree for `path(1,3)` is indeed infinite. Tabling detects this type of cycles on the SLG trees and handles them either as failures when the cycle subgoal has no succeeding derivations or as success if the cycle subgoal has succeeding derivations.

In the presence of cycles, logic programming tabling uses a completion mechanism to ensure that all answers are returned to all consumers. For our ground ProbLog goals, this boils down to returning information about failure or success and in case of success the nested trie. A simple way to achieve this for consumer goals that give rise to a cycle, is to assume that the generator goal will succeed and add a reference to the partially completed nested trie of the generator goal to the parent goal proof. Although a nested trie represents all the proofs for the subgoal, we do not need its final value to reuse it, i.e. to put a reference to it in the other tries.

The logic programming tabling completion mechanism monitors that all choice points of a generator goal have been explored. It also remembers all the locations where a non completed generator goal was used by a consumer. When the generator goal produces a new answer, the execution returns to any consumer to use that new answer [82]. In our case, as the answer is limited to success or failure of the goal, we do not require the execution to return to the consumer.

Now, our nested tries no longer only contain successful proofs. We optimistically assumed success for consumer goals giving rise to a cycle. If none of the goals involved in the cycle has a finite successful proof, they all fail. In this case, the nested tries contain references to failed subgoals. This failure is dealt with

during the ROBDD script generation step.

3.4 Nested Tries to ROBDD Definitions

As described in Section 2.3, SLD resolution is responsible for collecting the explanations that prove the query. These proofs form a Boolean formula, more specifically the collected formula of a non tabled ProbLog program without calls to negated probabilistic predicates[2] is in disjunctive normal form (DNF).

With tabling, SLD resolution is modified to SLG resolution and together with this modification also the DNF Boolean formula is modified to a forest of nested DNF sub-formulae. More precisely the Boolean formula now is a more compact representation of the previous DNF formula. Each subgoal has its own DNF formula that might refer to another DNF formula of another subgoal.

This section discusses how to extract the ROBDD definitions from the set of nested tries generated by ProbLog tabling. The main challenge for generating the ROBDD definitions from the nested tries, is tackling the encoded cycles in the set of nested tries efficiently.

By "unfolding" the nested tries back to a single Boolean formula we risk to actually redo much of the work that tabling saved us. Actually this process can end up being significantly more expensive, than the non tabled SLD evaluation, if treated naively. For that reason it is very important to perform as much reuse we can on the unfolding and avoid as many unfolding computations possibly.

3.4.1 Handling the Simple Case

We first present the case where there are no cycles in the nested tries. Using the generated forest of nested tries of Figure 3.7 at page 39 and traversing it naively depth first we get the ROBDD definitions of Table 3.1. The first column of the table presents the nested trie where currently the algorithm is; the second table column gives either the action taken by the algorithm or the generated ROBDD definition. Notice that lines from $L1$ to $L3$ are repeated in lines from $L6$ to $L8$. This repetition appears because the nested trie $t(p(3,5))$ is being referred twice in the forest of tries. The second appearance of $t(p(3,5))$ is credited in tabling reusage of proven goals. We optimistically would want to take advantage of this multiple appearance by not unfolding multiple times nested tries and by taking advantage of suffix similarity.

[2]The non tabled ProbLog program could still contain negated probabilistic facts or negated non probabilistic subgoals, facts.

Nested Trie	Generated ROBDD definitions
t(p(1,5))	Follow reference: *t(p(2,5))*
t(p(2,5))	Follow reference: *t(p(3,5))*
t(p(3,5))	Follow reference: *t(p(4,5))*
t(p(4,5))	1 `L1 = e(4,5)`
t(p(3,5))	2 `L2 = e(3,4) * L1`
t(p(3,5))	3 `L3 = e(3,5) + L2`
t(p(2,5))	4 `L4 = e(2,3) * L3`
t(p(1,5))	5 `L5 = e(1,2) * L4`
t(p(1,5))	Follow reference: *t(p(3,5))*
t(p(3,5))	Follow reference: *t(p(4,5))*
t(p(4,5))	6 `L6 = e(4,5)`
t(p(3,5))	7 `L7 = e(3,4) * L6`
t(p(3,5))	8 `L8 = e(3,5) + L7`
t(p(1,5))	9 `L9 = e(1,3) * L8`
t(p(1,5))	10 `L10 = L5 + L9`

Table 3.1: The ROBDD definitions for the Boolean formula of the nested tries of Figure 3.7 generated by the naive approach.

To achieve these improvements we use a table that memoizes nested tries unfolding and use that to reuse the generated ROBDD definitions of unfolded tries. Table 3.2 presents the ROBDD definitions of our previous example of the nested trie forest of Figure 3.7 at page 39 and the table used to memoize trie unfolding. Notice that, through this dynamic programming approach, the unfolding of a forest of tries that does not contain any cycles is linear in the nodes of all nested tries.

The third column of Table 3.2 presents the memoized ROBDD intermediate step which represents the generated ROBDD definitions for the nested trie shown in the first column. The algorithm that generates these definitions again works in depth first manner. It first traverses the leftmost branch of the first nested trie. Whenever it finds a reference to another nested trie it must first generate the ROBDD definitions of the new found (child) nested trie before it generates any ROBDD definition for the parent nested trie. In that manner for the nested tries of Figure 3.7 at page 39 the algorithm will first traverse through all child nested tries until it reaches $t(p(4,5))$ which does not have child nested tries. Then it generates the first ROBDD definition which by memoizing the intermediate step (L1) we can reuse the unfolding of this nested trie. Following this strategy we get the ROBDD definitions of Table 3.2. Notice that this time the nested trie $t(p(3,5))$ is reused at line 6 of instead of being generated again as in Table 3.1 lines 6 to 8.

Nested Trie	Generated ROBDD definitions	Memoized ROBDD Step
$t(p(1,5))$	Follow reference: *t(p(2,5))*	
$t(p(2,5))$	Follow reference: *t(p(3,5))*	
$t(p(3,5))$	Follow reference: *t(p(4,5))*	
$t(p(4,5))$	1 `L1 = e(4,5)`	`L1`
$t(p(3,5))$	2 `L2 = e(3,4) * L1`	
$t(p(3,5))$	3 `L3 = e(3,5) + L2`	`L3`
$t(p(2,5))$	4 `L4 = e(2,3) * L3`	`L4`
$t(p(1,5))$	5 `L5 = e(1,2) * L4`	
$t(p(1,5))$	6 `L6 = e(1,3) * L3`[1]	
$t(p(1,5))$	7 `L7 = L5 + L6`	`L7`

[1]Reusage of the generated ROBDD definitions.

Table 3.2: ROBDD definitions for the Boolean formula of the nested tries of Figure 3.7 generated through memoization.

This approach presented up until here is very much related and similar to the dynamic programming approach PRISM [73,74] follows. The extra assumptions imposed by PRISM ensure the absence of cycles in the probabilistic information.

3.4.2 Handling Cycles and the Ancestor List

Introducing cycles in the graph alters things significantly. While the tabling mechanism deals the cycles at the proof collection stage, it propagates the problem to the next stage namely the generation of ROBDD definitions. Three new complications arise because of the cycles. First we need a mechanism to detect cycles, then find what ROBDD definition to generate for a cycle, and finally decide when an unfolded trie can be reused.

To detect the cycles we introduce an ancestor list, for each encountered nested trie, that remembers the previously visited nested tries. This ancestor list is similar to the path kept by a depth first search algorithm for detecting cycles.

Definition 7 (Ancestor List)**.** *The ancestor list $A_{t(g)}$ of a nested trie $t(g)$ is the set of nested tries that were traversed to reach $t(g)$.*

Cycles typically give rise to proofs that do not contribute to the final probability. Figure 3.8 shows the collected nested tries, that contain infinite cycles, for the query `path(1,5)` of the undirected graph in Figure 3.5. Consider the following two successful proofs reconstructed from the nested tries:

```
edge(1,2),edge(2,3),edge(3,5)
```

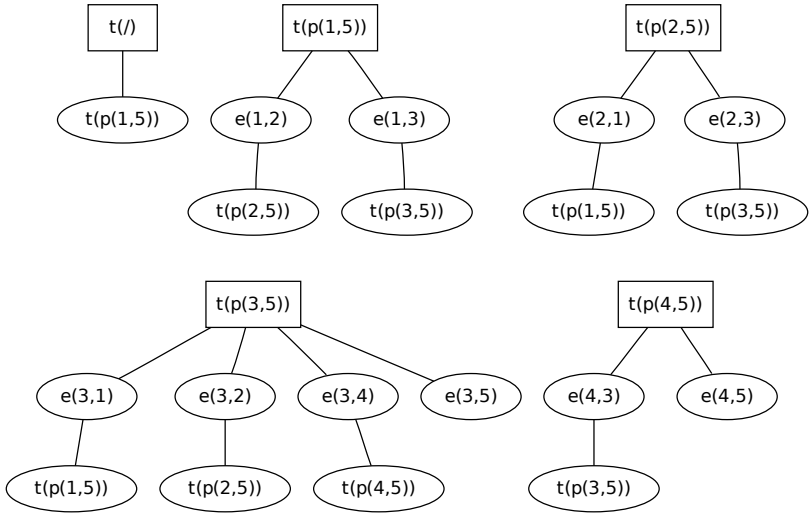


Figure 3.8: The collected nested tries for the undirected graph of Figure 3.5 using the predicate of Figure 3.3(b) by the query: `path(1,5)`.

```
edge(1,2),edge(2,3),edge(3,1),edge(1,3),edge(3,5)
```

What do we notice about their contribution to the final probability? At the level of the Boolean formula, we can simplify the disjunction of the first and the second proof to the first proof: $(edge(1,2) \wedge edge(2,3) \wedge edge(3,5)) \vee (edge(1,2) \wedge edge(2,3) \wedge edge(3,1) \wedge edge(1,3) \wedge edge(3,5)) \equiv edge(1,2) \wedge edge(2,3) \wedge edge(3,5) \wedge (true \vee (edge(3,1) \wedge edge(1,3))) \equiv edge(1,2) \wedge edge(2,3) \wedge edge(3,5)$. We say that the first proof is more general than the second one. Let $set(p)$ denote the set of probabilistic facts of a successful proof p in a trie.

Definition 8 (More general proof)**.** *A proof p is **more general** than a proof p' denoted by $p \prec p'$ iff $set(p) \subset set(p')$.*

Definition 9 (Proof in a trie)**.** *A proof p is in a trie T iff $p \in T$.*

Definition 10 (Non-minimal proof)**.** *A proof p is a **non-minimal** proof in a trie T iff there $\exists p' \in T$ where $p' \preceq p$.*

The following theorem states that for a goal g with a trie $t(g)$, we can safely eliminate the proofs that contain an identical goal with the ancestor list $A_{t(g)}$.

Theorem 1 (Non-minimality of cycles)**.** *Let g_a be the first occurrence (ancestor) and g_d the second occurrence (descendant) of a goal g in the ancestor list of a complete successful proof. If unfolding $t(g)$ for the first occurrence g_a constructs for g a finite successful proof without adding g_d to the ancestor list, then, the unfolding of $t(g)$ that also adds g_d in the ancestor list, constructs non-minimal proofs.*

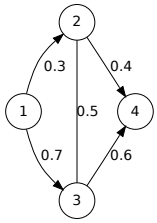
Proof. Suppose that unfolding $t(g)$ for the first occurrence g_a constructs the following complete successful proof pr_l for g_a: $pr_b.pr_t.pr_c$ with $set(pr_b) = \{p_{b1}, ..., p_{bn}\}$, $set(pr_c) = \{p_{c1}, ..., p_{cl}\}$ and pr_t resulting from unfolding $t(g)$ for the second occurrence g_d. We have to show that pr_l is a non-minimal proof in $t(g)$: there must be another proof pr_s in $t(g)$ such that $set(pr_s) \subseteq set(pr_l)$.

From the hypothesis of the theorem we know that g has at least one finite successful proof pr_a with $set(pr_a) = \{p_{a1}, ..., p_{am}\}$ that does not add g_d to the ancestor list. When this is used to unfold the second occurrence of $t(g)$, $set(pr_a) \subseteq set(pr_l)$ and thus pr_l is non-minimal. The same reasoning holds for all other finite successful proofs of g that do not add g_d to the ancestor list. Finally, we could unfold $t(g)$ again by $pr_b.pr_t.pr_c$ resulting in $pr_b\ pr_b.pr_t.pr_c.pr_c$, but in the end, to get a finite successful proof, one of the finite successful proofs of g that do not add g_d to the ancestor list will be used. Thus again pr_l is a non-minimal proof. □

When unfolding detects a cycle, we can prune the current proof: either because it is a failing proof or by Theorem 1 it gives rise to non-minimal proofs. If pruning results in an empty trie, which encodes failure, we can prune the parent branch.

Previously we showed that it is crucial to reuse the unfolding of nested tries as much as possible. The absence of cycles allowed us to use all re-appearances of nested tries and thus ensures a linear algorithm for generating the ROBDD definitions. Unfortunately with the presence of cycles the naive reusage of unfolding is not possible. We use the example graph of Figure 3.9 as a counter example where the reusage of unfolding produces erroneous results. The counter example uses the query: `path(1,4)` to collect the nested tries of Figure 3.10. Using the naive approach of blindly reusing any memoized ROBBD definitions by a nested trie unfolding, for our counter example, the ROBDD definitions in Table 3.3 are generated.

The ROBDD definitions of Table 3.3 calculate a probability of $P = 0.5058$; but the correct probability is actually $P_{correct} = 0.545$. This difference results from the cycle introduced by the undirected edge from node 2 to 3 of the graph in Figure 3.9. As the unfolding occurs depth first like, the algorithm unfolds the




```
0.3::edge(1, 2).
0.7::edge(1, 3).
0.5::edge(2, 3).
0.4::edge(2, 4).
0.5::edge(3, 2).
0.6::edge(3, 4).
```

Figure 3.9: A probabilistic graph with cycles which is used as a counter example.

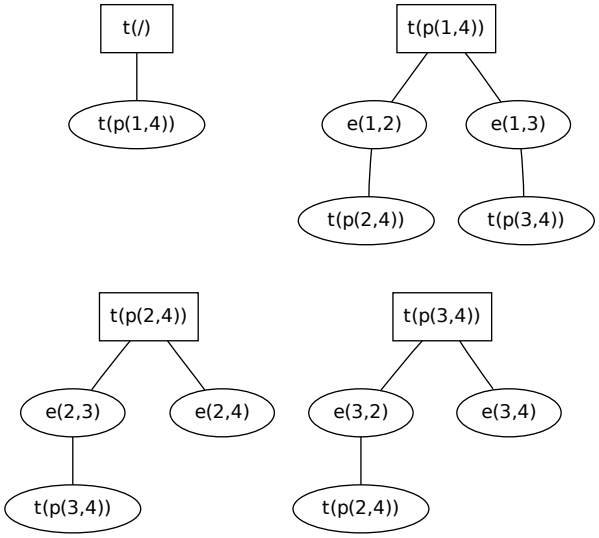


Figure 3.10: The collected nested tries for the counter example graph of Figure 3.9 using the predicate of Figure 3.3(b) by query: `path(1,4)`.

Nested Trie	Generated ROBDD definitions	Memoized ROBDD Step
$t(p(1,4))$	Follow reference: *t(p(2,4))*	
$t(p(2,4))$	Follow reference: *t(p(3,4))*	
$t(p(3,4))$	Found cycle: prune proof	
$t(p(3,4))$	1 `L1 = e(3,4)`	`L1`
$t(p(2,4))$	2 `L2 = e(2,3) * L1`	
$t(p(2,4))$	3 `L3 = e(2,4) + L2`	`L3`
$t(p(1,4))$	4 `L4 = e(1,2) * L3`	
$t(p(1,4))$	5 `L5 = e(1,3) * L1`[1]	
$t(p(1,4))$	6 `L6 = L4 + L5`	`L6`

[1]Reusage of the generated ROBDD definitions.

Table 3.3: ROBDD definitions for the Boolean formula of the counter example nested tries of Figure 3.10 generated through memoization. These definitions calculate an erroneous result and are used as a counter example proving that simple memoization is not sufficient.

ROBDD definitions of the nested trie $t(p(3,4))$ before it unfolds the nested trie $t(p(2,4))$. The first branch of $t(p(3,4))$ resolves in a cycle as it contains a reference to $t(p(2,4))$ which currently is an ancestor nested trie. By Theorem 1 the branch with the cycle is dropped. The ROBDD definition for $t(p(3,4))$ is shown at line 1 of Table 3.3. Later on, the nested trie $t(p(3,4))$ is encountered from the branch of the nested trie $t(p(1,4))$, in this case $t(p(2,4))$ is not an ancestor and thus the first branch of $t(p(3,4))$ is not part of a cycle. Now it has a probability mass to offer, but because of the memoization the current strategy reuses the previous result and calculates the probability wrongly.

The above example illustrates that it is erroneous to reuse the unfolding of nested tries that introduce cycles. It is not safe as different occurrences might give rise to different proofs. Actually, it depends on the context of the occurrence of the nested trie, in particular on the ancestor list of the occurrence. In order to maintain unfolding reusage for nested tries with cycles, we start from the following observation. When two occurrences of a reference to a nested trie have exactly the same ancestor list, then obviously the unfolding of the references will introduce exactly the same cycles and exactly the same ROBDD definitions.

Theorem 2 (Same ancestor and ROBDD definitions)**.** *Let t_1 with ancestors A be the first occurrence of a nested trie, and t_2 another occurrence with ancestors A. Both occurrences have the identical ROBDD definition unfolding.*

Proof. Indeed as t_1 and t_2 are the same nested trie and each occurrence has the same ancestor list A, exactly the same branches from t_1 and t_2 would be

Nested Trie	**Generated ROBDD definitions**	**Ancestor List**	**Memoized ROBDD**
t(p(1,5))	Follow reference: *t(p(2,5))*	[]	
t(p(2,5))	Found cycle: prune proof	[t(p(1,5))]	
t(p(2,5))	Follow reference: *t(p(3,5))*	[t(p(1,5))]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	Follow reference: *t(p(4,5))*	[t(p(1,5)),t(p(2,5))]	
t(p(4,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5)),t(p(3,5))]	
t(p(4,5))	1 `L1 = e(4,5)`	[t(p(1,5)),t(p(2,5)),t(p(3,5))]	L1
t(p(3,5))	2 `L2 = e(3,4) * L1`	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	3 `L3 = e(3,5) + L2`	[t(p(1,5)),t(p(2,5))]	L3
t(p(2,5))	4 `L4 = e(2,3) * L3`	[t(p(1,5))]	L4
t(p(1,5))	5 `L5 = e(1,2) * L4`	[]	
t(p(1,5))	Can not reuse L3	[]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5))]	
t(p(3,5))	Can not reuse L4	[t(p(1,5))]	
t(p(2,5))	Found cycle: prune proof	[t(p(1,5)),t(p(3,5))]	
t(p(2,5))	Found cycle: prune proof	[t(p(1,5)),t(p(3,5))]	false
t(p(3,5))	`e(3,2) * false`[1]	[t(p(1,5))]	
t(p(3,5))	Can not reuse L1	[t(p(1,5))]	
t(p(4,5))	Found cycle: prune proof	[t(p(1,5)),t(p(3,5))]	
t(p(4,5))	7 `L6 = e(4,5)`	[t(p(1,5)),t(p(3,5))]	L7
t(p(3,5))	8 `L7 = e(3,4) * L6`	[t(p(1,5))]	
t(p(3,5))	9 `L8 = e(3,5) + L7`	[t(p(1,5))]	L9
t(p(1,5))	10 `L9 = e(1,3) * L8`	[]	
t(p(1,5))	11 `L10 = L5 + L9`	[]	L11

[1]False simplifies the expression.

Table 3.4: ROBDD definitions for the Boolean formula of the nested tries of Figure 3.8 generated through memoization and ancestor list check.

pruned as cycles resulting to exactly the same ROBDD definitions. □

Returning to our running example and using Theorem 2 we present the ROBDD definitions generated for the nested tries of Figure 3.8 at page 44 in Table 3.4.

3.4.3 Optimization I: Subset

Unfortunately if we remain at the reusage stated at Theorem 2 then for many ProbLog programs that contain cycles the benefit gained by SLG resolution

will be lost during the generation of the ROBDD definitions. Fortunately, we can do better than that.

Theorem 3 (Ancestor subset check)**.** *Consider two references to the same nested trie, one occurrence t_1 has ancestor list A_1 and the other occurrence t_2 has ancestor list A_2 such that $A1 \subset A2$[3]. Then t_2 will at least participate in the same cycles as t_1.*

Proof. Obviously as A_2 contains all ancestors of A_1 (and possibly more) then all the cycles appearing at the unfolding of t_1 will appear at the unfolding of t_2 (and possibly more). □

As is shown by Theorem 1 unfolding the proofs of a cycle produces non-minimal proofs that do not alter the Boolean formulae. Thus from Theorem 3 and 1 we can conclude that the Boolean formula of t_1 is equivalent to that of t_2 and dynamic programming can reuse it instead of unfolding t_2 during the generation of ROBDD definitions.

Returning to the example of Figure 3.5, Table 3.5 shows the generation process of the ROBDD definitions in parallel with the memoization of the dynamic programming approach. As the third column we have added the ancestor list generated at each step of the generation. The [1] annotates a reusage of the nested trie *t(p(2,5))* which only occurs if ancestor subset check is used. At that location we can see that the ancestors for *t(p(2,5))* where $A_2 = [t(p(1,5)), t(p(3,5))]$ but the ROBDD definitions memoized had as ancestors $A_1 = [t(p(1,5))]$. As $A_1 \subset A_2$ the ROBDD definitions for *t(p(2,5))* are reused. If we had re-calculated *t(p(2,5))*, as if ancestor subset check was not used, the resulting ROBDD definition would be **false**. This result would have simplified the final ROBDD script. We address this issue later.

With this approach we managed to restore some reusage of the nested trie unfolding, which as shown in the experiments improves the performance significantly. Unfortunately, the complexity of the unfolding algorithm still depends highly on the presence of cycles and in the worst case scenario appears to be exponential.

3.4.4 Optimization II: Ancestor List Refine

We present a further optimization that aims at further reuse of nested trie unfolding. This optimization relies on the observation that only the nested tries that appear in the branches of a nested trie and the children of the children and

[3] An ancestor list can be interpreted as a set of ancestors.

Nested Trie	**Generated ROBDD definitions**	**Ancestor List**	**Memoized ROBDD**
t(p(1,5))	Follow reference: *t(p(2,5))*	[]	
t(p(2,5))	Found cycle: prune proof	[t(p(1,5))]	
t(p(2,5))	Follow reference: *t(p(3,5))*	[t(p(1,5))]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	Follow reference: *t(p(4,5))*	[t(p(1,5)),t(p(2,5))]	
t(p(4,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5)),t(p(3,5))]	
t(p(4,5))	1 `L1 = e(4,5)`	[t(p(1,5)),t(p(2,5)),t(p(3,5))]	L1
t(p(3,5))	2 `L2 = e(3,4) * L1`	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	3 `L3 = e(3,5) + L2`	[t(p(1,5)),t(p(2,5))]	L3
t(p(2,5))	4 `L4 = e(2,3) * L3`	[t(p(1,5))]	L4
t(p(1,5))	5 `L5 = e(1,2) * L4`	[]	
t(p(1,5))	Can not reuse L3	[]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5))]	
t(p(3,5))	6 `L6 = e(3,2) * L4`[1]	[t(p(1,5))]	
t(p(3,5))	Can not reuse L1	[t(p(1,5))]	
t(p(4,5))	Found cycle: prune proof	[t(p(1,5)),t(p(3,5))]	
t(p(4,5))	7 `L7 = e(4,5)`	[t(p(1,5)),t(p(3,5))]	L7
t(p(3,5))	8 `L8 = e(3,4) * L7`	[t(p(1,5))]	
t(p(3,5))	9 `L9 = e(3,5)+L6+L8`	[t(p(1,5))]	L9
t(p(1,5))	10 `L10 = e(1,3) * L9`	[]	
t(p(1,5))	11 `L11 = L5 + L10`	[]	L11

[1]Reusage of L4 as the memoized ancestor list of *t(p(2,5))* is a subset of the new ancestor list that *t(p(2,5))* would be produced with. Formally: [t(p(1,5))] $\subseteq$ [t(p(1,5)),t(p(3,5))].

Table 3.5: ROBDD definitions for the Boolean formula of the nested tries of Figure 3.8 generated through memoization and ancestor list subset check.

so forth can affect the generation of the ROBDD definitions. Thus, we could refine the ancestor list to a smaller subset that contains only those.

Theorem 4 (Descendants of nested tries)**.** *The ROBDD definitions of a nested trie T with ancestor list A and descendants list D depend only on the nested tries that appear both in A and D thus the ancestor list can be refined as $A' = A \cap D$.*

Proof. It is easy to prove the base case of Theorem 4, if a nested trie does not contain any reference to another nested trie then the resulting ROBDD definitions are always the same regardless of the ancestors. Assuming that all the descendant nested tries in D_{T_1} that are referred have no descendants themselves then the ROBDD definitions of T depend only on the appearance of

the D_{T_1} in the ancestor list. Either a descendant nested trie from D_{T_1} appears in the ancestor list and introduces a cycle reducing the ROBDD definitions, or it does not appear and unfolding it will produce further ROBDD definitions. Similarly if a descendant nested trie contains references to other descendants then the produced ROBDD definitions for this nested trie will depend to its descendant nested tries. Applying this reasoning recursively to all descendants then it is clear that the resulting ROBDD definitions are influenced only by the descendants that also appear in the ancestor list. □

From the implementation point of view this optimization imposes extra challenges. First it requires that we collect an exhaustive list of all descendants used and maintain it after backtracking, further it requires at each reuse of ROBDD definitions of a nested trie to include the descendants of that nested trie as descendants of the parent nested trie. Both issues are solved by using the refined ancestor list of the memoized nested trie as its descendants list.

We present in Table 3.6 an example of extra memoization reusage because of the ancestor list refinement. Notice that the ancestor list of *t(p(4,5))* shrinks from [t(p(1,5)),t(p(2,5)),t(p(3,5))] to [t(p(3,5))], this allows it to be reused at the second encounter of the same nested trie, as now the ancestor list of the memoized *t(p(4,5))* is a subset of the ancestor list for the second occurrence of *t(p(4,5))* which is [t(p(1,5)),t(p(3,5))].

Through the use of this optimization, further reusage was possible and significant performance benefits appear in many queries that contain cycles especially if the nested tries contain symmetries.

3.4.5 Optimization III: Pre-process Step

Another interesting observation is that while the ancestor subset check, increases the reuses of previous unfolded ROBDD definitions, it often uses more complex ROBDD definitions than necessary to describe the Boolean formulae. The pre-process optimization aims at improving exactly that.

The underlying idea is that an inexpensive pre-process step can generate the basic unfolding of each nested trie together with its minimal ancestor list. Then during the generation of ROBDD definitions, before reusing any memoized unfolding, we first check if any minimal unfolding can be reused to introduce a simpler Boolean formulae instead.

Table 3.7 presents the ROBDD definitions generated by the pre-process step, for the nested tries of Figure 3.8 at page 44. This optimization does not follow any references or checks for any cycles, it processes every nested trie once and

Nested Trie	**Generated ROBDD definitions**	**Refined Ancestor List**	**Memoized ROBDD**
t(p(1,5))	Follow reference: *t(p(2,5))*	[]	
t(p(2,5))	Found cycle: prune proof	[t(p(1,5))]	
t(p(2,5))	Follow reference: *t(p(3,5))*	[t(p(1,5))]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	Follow reference: *t(p(4,5))*	[t(p(1,5)),t(p(2,5))]	
t(p(4,5))	Found cycle: prune proof	[t(p(3,5))][1]	
t(p(4,5))	1 `L1 = e(4,5)`	[t(p(3,5))][1]	L1
t(p(3,5))	2 `L2 = e(3,4) * L1`	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	3 `L3 = e(3,5) + L2`	[t(p(1,5)),t(p(2,5))]	L3
t(p(2,5))	4 `L4 = e(2,3) * L3`	[t(p(1,5))]	L4
t(p(1,5))	5 `L5 = e(1,2) * L4`	[]	
t(p(1,5))	Follow reference: *t(p(3,5))*	[]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5))]	
t(p(3,5))	6 `L6 = e(3,2) * L4`[2]	[t(p(1,5))]	
t(p(3,5))	7 `L7 = e(3,4) * L1`[3]	[t(p(1,5))]	
t(p(3,5))	8 `L8 = e(3,5) +L7+L6`	[t(p(1,5))]	L8
t(p(1,5))	9 `L9 = e(1,3) * L8`	[]	
t(p(1,5))	10 `L10 = L5 + L9`	[]	L10

[1]Refinement of the ancestor list with the children nested tries of the current nested trie.
[2]Reusage of the generated ROBDD definitions as [t(p(1,5))] ⊆ [t(p(1,5)),t(p(3,5))].
[3]Reusage of the generated ROBDD definitions as [t(p(3,5))] ⊆ [t(p(1,5)),t(p(3,5))].

Table 3.6: ROBDD definitions for the Boolean formula of the nested tries of Figure 3.8 at page 44 generated through memoization, ancestor list subset check and ancestor list refinement.

it assumes *false* as the result of each branch that contains a reference to any other nested trie. In that way, it generates and memoizes the base case for each nested trie with an ancestor list that contains all referred children nested tries. By using the memoized base cases shown in Table 3.7 the full generation of the ROBDD definitions of the nested tries is shown in Table 3.8. Notice that the nested trie *t(p(4,5))* is immediately reused from the pre-processed memoizations. Also notice that, because the algorithm uses the first memoized unfolding that has a suitable ancestor list, the nested trie *t(p(2,5))* is reused from the pre-processed tries instead of the newly generated tries. This simplifies the ROBDD definitions imposing less work on the ROBDD generation; in our example the final ROBDD definitions shown in Table 3.8 contain 8 operations against the 10 and 9 operations from the ROBDD definitions generated respectively in Table 3.5 and Table 3.6.

Nested Trie	**Generated ROBDD definitions**	**Ancestor List**	**Memoized ROBDD**
t(p(1,5))		[t(p(2,5)),t(p(3,5))]	false
t(p(2,5))		[t(p(1,5)),t(p(3,5))]	false
t(p(3,5))	1 `L1 = e(3,5)`	[t(p(1,5),t(p(2,5)),t(p(4,5))]	L1
t(p(4,5))	2 `L2 = e(4,5)`	[t(p(3,5))]	L2

Table 3.7: ROBDD definitions generated from the pre-process step for the nested tries of Figure 3.8.

Nested Trie	**Generated ROBDD definitions**	**Ancestor List**	**Memoized ROBDD**
t(p(1,5))	Follow reference: *t(p(2,5))*	[]	
t(p(2,5))	Found cycle: prune proof	[t(p(1,5))]	
t(p(2,5))	Follow reference: *t(p(3,5))*	[t(p(1,5))]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	3 `L3 = e(3,4) * L2`[1]	[t(p(1,5)),t(p(2,5))]	
t(p(3,5))	4 `L4 = e(3,5) + L3`	[t(p(1,5)),t(p(2,5))]	L4
t(p(2,5))	5 `L5 = e(2,3) * L4`	[t(p(1,5))]	L5
t(p(1,5))	6 `L6 = e(1,2) * L5`	[]	
t(p(1,5))	Follow reference: *t(p(3,5))*	[]	
t(p(3,5))	Found cycle: prune proof	[t(p(1,5))]	
t(p(3,5))	`e(3,2) * false`[2]	[t(p(1,5))]	
t(p(3,5))	7 `L7 = e(3,4) * L2`[1]	[t(p(1,5))]	
t(p(3,5))	8 `L8 = e(3,5) + L7`	[t(p(1,5))]	L9
t(p(1,5))	9 `L9 = e(1,3) * L8`	[]	
t(p(1,5))	10 `L10 = L6 + L9`	[]	L10

[1]Reusage of a pre-processed nested trie *t(p(4,5))* as [t(p(3,5))] ⊆ [t(p(1,5)),t(p(2,5)), t(p(3,5))].
[2]Reusage of a pre-processed nested trie *t(p(2,5))* as [t(p(1,5)),t(p(3,5))] ⊆ [t(p(1,5)), t(p(3,5))]. False simplifies the expression.

Table 3.8: ROBDD definitions for the Boolean formula of the nested tries of Figure 3.8 generated through memoization, with ancestor list subset check, ancestor list refinement and using pre-processed ROBDD definitions.

In contrast to the previous ones this optimization comes at an added cost. It takes one extra iteration among all nested tries to generate the pre-processed ROBDD definitions. This cost is linear in the number of variables stored inside the nested tries; as the algorithm has a higher worst case complexity, this cost is insignificant. This is verified by the experiments where the cost is barely noticeable and in many cases the benefit much more significant.

3.4.6 Optimizing the Representation

As we already said, the unfolding of the nested tries heavily depends on the ancestor and the descendant lists. In order to perform set operations we handle both the ancestor and descendant lists as ordered lists that contain only unique nested trie references. To further improve the performance of the algorithm we use arbitrary long integers as bit-strings to represent both the ancestor and descendant lists as sets.

While the manipulation of the lists is not the main reason for the computational overhead in generating the ROBDD definitions, in big problems the maintenance of the lists in the correct form and the subset check has an observable cost. By using arbitrary long integers those operations become significantly faster providing a noticeable improvement in performance as we can see in the experiments.

Finally, we need to mention that Algorithm 4.4 which uses these optimizations, can be found and is further explained in Section 4.8. The algorithm is quite complex and uses several other algorithms as its components, therefore we urge the reader to continue reading and return to this section again at that point, if needed.

3.5 Implementation

ProbLog tabling has been realized through a program transformation. This final section presents two different transformations that implement ProbLog tabling. The first one is shown in Algorithm 3.1, can be used in a Prolog system without tabling but is limited to work only on fully grounded goals, while the second transformation is shown in Algorithm 3.2 takes advantage of the existing tabling of a Prolog system and can work with any type of goals.

3.5.1 Light Weight Tabling Implementation

In order to be able to explain the tabling implementation we mention that the predicates `nb_getval/2`, `b_setval/2` which handle global variables in Prolog [76] are available in several Prolog systems. For further information concerning global variables please see a Prolog manual [71]. Similarly, in the transformation we use the predicates `put_trie_entry/2`, `empty_trie/1`, `merge_trie/3`. These predicates are based on Yap's trie implementation. For more information please look Yap's manual [71]

For this approach we use a custom made, light weight tabling implementation. The transformation used is presented in Algorithm 3.1. This implementation has the advantage that it is more flexible and easier to integrate with any Prolog system. This approach is used in the ProbLog2010 implementation. We are not able to use the built-in approach together with the ProbLog2010 implementation because of limitations in the way inference collects information in its data structures when performing inference.

The main concept is to transform the body of the tabled predicate so that it first checks within a table whether the specific grounded goal of the predicate has been called before or not. This approach is viable only for ground goals because it memorizes only success or failure of a goal. It also memorizes the derivations used to prove the goal and constructs the nested tries.

The tabling mechanism is based on using a table for memoizing entries. These entries, when stored, use as a key the grounded head (`Head`) of the goal. This key is indexed similarly as in a hash table for fast access. In the table we store the trie identifiers `GoalTrie`, `Trie`, `SuspendedTrie` and the goal's completion state.

The `Trie` is used to store the successful derivations of the goal; the `SuspendedTrie` is used to store the derivations that resulted in a cycle; failed derivations are ignored. At the completion of a goal, if successful derivations exist for the goal, then all `Trie` entries and all `SuspendedTrie` entries are merged into `GoalTrie`. If there are no successful derivations, the completed entry is memoized and is reused as if it succeeded normally, but because of the empty `GoalTrie` it signifies failure instead of success. Every tabled goal has its own unique set of those three tries.

The `goal_lookup/5` predicate has two responsibilities: first responsibility is to return the stored entry under the key `Head`; its second responsibility is to initialize three empty tries with identifiers `GoalTrie`, `Trie`, `SuspendedTrie` for the key `Head` if there is not yet a stored entry for the key `Head`.

As we mentioned, the `goal_lookup/5` predicate is responsible for the read access

Algorithm 3.1 ProbLog tabling transformation for only ground goals without using built-in tabling support.

input A clause to be tabled (`Head :- Body`).
output A transformed clause (`Head :- NewBody`).

```
term_expansion((Head :- Body), (Head :- NewBody)):-
  NewBody = (
    goal_lookup(Head, GoalTrie, Trie, SuspendedTrie, Completion),
    (Completion = newgoal ->
      update_table(Head, GoalTrie, Trie, SuspendedTrie, ongoing),
      nb_getval(explanation, ParentExplanation),
      nb_getval(trie, ParentTrie),
      b_setval(explanation, []),
      (
        Body,
        nb_getval(suspended_tries, Susp),
        (memberchk(GoalTrie, Susp) ->
          b_setval(trie, SuspendedTrie)
        ;
          b_setval(trie, Trie)
        )
        nb_getval(explanation, CurrentExplanation),
        nb_getval(trie, CurrentTrie),
        put_trie_entry(CurrentTrie, CurrentExplanation),
        fail
      ;
        update_table(Head, GoalTrie,Trie,SuspendedTrie, finished)
      )
      \+ empty_trie(Trie),
      merge_trie(Trie, SuspendedTrie, GoalTrie),
      b_setval(trie, ParentTrie),
      b_setval(explanation, [GoalTrie|ParentExplanation])
    ; Completion = ongoing ->
      nb_getval(suspended_tries, Susp),
      b_setval(suspended_tries, [GoalTrie|Susp]),
      nb_getval(explanation, CurrentExplanation),
      b_setval(explanation, [GoalTrie|CurrentExplanation])
    ; Completion = finished
      nb_getval(explanation, CurrentExplanation),
      b_setval(explanation, [GoalTrie|CurrentExplanation])
    )
  ).
```

to the table and returns the appropriate tries (`GoalTrie`, `Trie`, `SuspendedTrie`) and completion status of the goal. There are three possible completion statuses for a goal: `newgoal` when a goal appears for the very first time, `finished` when all the choice points of a goal have been explored and `ongoing` when the goal has at least one open choice point.

When there is no entry for the key `Head` in the table, `goal_lookup/5` predicate returns the status `newgoal` and initializes three new tries (`GoalTrie`, `Trie`, `SuspendedTrie`) to return.

The predicate `update_table/5` is responsible for the write access to the table. Its first functionality is, to create an entry under the access key `Head` when no such entry already exists in the table; in that entry it stores `GoalTrie`, `Trie`, `SuspendedTrie`. The newly created entry starts with the the `newgoal` status. The second functionality of `update_table/5` is, when an entry under the key `Head` already exists to update the state of the goal. The predicate never creates a second entry under the same key `Head`.

When a new goal is encountered the transformed predicate reads the global variables (**explanation**, **trie**) of ProbLog2010 which are responsible for collecting the probabilistic information. It then alters them in order to create a nested trie for each goal. After each successful derivation the tabling mechanism checks whether the current goal is suspended in order to choose in which of the two tries (`Trie`, `SuspendedTrie`) to store the derivation.

When `goal_lookup/5` returns the `ongoing` state, it means that the resolution discovered a cycle. To handle the cycle we simply mark the goal as suspended and assume success of the derivation in order to continue the execution normally.

If by the end of the goal proving, there are no successful derivations then the cycle derivations stored in the `SuspendedTrie` should all fail, the goal fails and `GoalTrie` remains empty indicating failure.

Reusage in this tabling implementation is very simple. Whenever `goal_lookup/5` returns a completion status `finished` then `GoalTrie` is returned to be added in the collected derivations of the father goal. Regardless of success of the goal the `GoalTrie` is added. For this reason when we generate the ROBDD definitions from the nested tries we handle the empty tries as $false$ branches.

3.5.2 Built-in Tabling

Using the built-in tabling of a Prolog system has many benefits. First of all, the tabling mechanism of ProbLog now can be generalized to work also with non-ground goals. Second, we do not need to implement a complex

completion mechanism and to use suspended tries any-more. This approach still produces the nested tries and maintains the connection between the goal and the appropriate trie within the tables of the tabling mechanism. The problem with this approach is that it imposes extra requirements on the structure of the ProbLog inference algorithm. The built-in tabling approach is used in the new ProbLog implementation.

ProbLog tabling is again implemented as a program transformation which is shown in Algorithm 3.2. The transformed body of the original predicate calls a new predicate that is tabled by Prolog and that has two additional arguments. The first new argument is `EngineID` an identifier that is used for specifying to which ProbLog engine the goal exists. This argument protects the tables from repeated executions of the same goals with different inference methods. It is also used to check for reuse over compatible inference methods. The second argument is `SubTrie` which is the trie that stores the derivations that prove the goal.

The predicate `goal_lookup/2` is responsible to assign a unique `SubTrie` at each goal the first time that it is called. The relation among the trie `SubTrie` and the goal `Head` is stored in the arguments of the predicate and inside the internal table of the Prolog tabling mechanism.

This approach takes advantage of the ProbLog engine which is presented in Section 6.3. The ProbLog2010 implementation does not have an engine, nor states. The light weight tabling approach we used for ProbLog tabling in ProbLog2010 inspired the concepts of an engine and a state in ProbLog which then appeared in the new ProbLog2011 implementation.

3.6 Experiments

Our tabling directly interacts with the first two execution steps of ProbLog, SLD resolution and Boolean formulae preprocessing. The third step (ROBDD compilation) is affected indirectly. It is well-known that ROBDD packages use heuristics while constructing a ROBDD and that their behaviour depends on the input, i.e., different inputs describing the same Boolean formula can give rise to different results and/or execution times. We address the following questions:

1. How does our light weight tabling implementation perform both in time and in space for the SLD-resolution?
2. How do the nested tries compare with their flat equivalents during the Boolean formulae preprocessing and the ROBDD compilation?

Algorithm 3.2 ProbLog tabling transformation that uses Prolog built-in tabling support.

input A clause to be tabled (`Head :- Body`).

output A list containing two transformed clauses (`Head :- NewBody`, `NewHead :- NewBody2`).

```
term_expansion((Head :- Body), [(Head :- NewBody),
                                (NewHead :- NewBody2)]) :-
  Head ..= [Predicate|Arguments],
  atom_concat(Predicate, '_internal', NewPredicate),
  append(Arguments,[EngineID,SubTrie],NewArguments),
  NewHead ..= [NewPredicate|NewArguments],
  length(Arguments, Arity),
  NewArity is Arity + 2,
  table(NewPredicate/NewArity),
  NewBody = (
    get_problog_state(explanation, Explanation),
    get_problog_state(trie, Trie),
    set_problog_state(explanation, []),
    NewHead,
    set_problog_state(explanation, [SubTrie|Explanation]),
    set_problog_state(trie, Trie)
  ),
  NewBody2 = (
    Body,
    goal_lookup(Head, SubTrie),
    set_problog_state(trie, SubTrie),
    continuation_explanation
  ).
```

3. How do the nested tries with cycles perform and what are the effects of the ancestor subset check?

3.6.1 Benchmark Programs

Our benchmarks represent three different categories of problems that are typical for ProbLog. Our **weather** benchmark is a typical example of a (Hidden) Markov Model (HMM). In this type of problems the value of the current time state depends on the value of the previous time state. It is well known that time series problems, when naively implemented, are of exponential complexity. Using tabling for this type of problems we expect significant improvement as memoization reduces the complexity of the problem to linear[4]. The size of the weather problem is determined by the "Day" argument.

The **bloodtype** benchmark models a Bayesian network and infers the probability of a person's bloodtype based on its ancestors. The size of the problem depends on the number of generations and the interconnectivity of the nodes that represent the family. We took a randomly connected pedigree chart of fifteen generations and ask for the bloodtype of random persons from each generation.

From the link discovery [16] applications, we took a **graph** benchmark, namely a number of graphs from the biomine database [77]. This benchmark expresses connections between various types of objects such as genes, proteins, tissues, etc. and predicts relationships among them. We use the program of Figure 3.3(a) for the non-tabled version and the program of Figure 3.3(b) for the tabled version. For our experiments we used the first sample of graphs and the queries of [16]. The size is determined by the number of edges of the graph and the interconnectivity between the nodes. As these graphs are cyclic, cycle handling is necessary.

3.6.2 Results

All the experiments are performed on an Intel^R Core^{TM}2 Duo CPU at 3.00GHz with 2GB of RAM memory running Ubuntu 8.04.2 Linux under a usual load. The reported times are the averages of five runs from which we dropped the best and worst time and all times are in milliseconds. For the SLD resolution and the Boolean formulae preprocessing, we used a time-out of 1 hour, while for the ROBDD compilation we used a 1 minute time-out. One reason for that is

[4]The complexity statements regard only the SLD resolution step as it is always #P-Hard to assess the probability for a sum-of-products.

	Memory (Bytes)		SLD/SLG resolution		Boolean formulae preprocessing		ROBDD compilation	
Day	non-tab	tab	non-tab	tab	non-tab	tab	non-tab	tab
1	352	912	0	0	0	0	5	5
2	1048	2148	0	0	0	0	5	5
13	184941472	15744	115400	2	2584	4	802	37
14	554824408	16980	380011	3	7938	4	2380	36
167	-	206088	-	25	-	89	-	9728
1600	-	1977276	-	262	-	3587	-	-

Table 3.9: Results for the weather program. Times are in milliseconds.

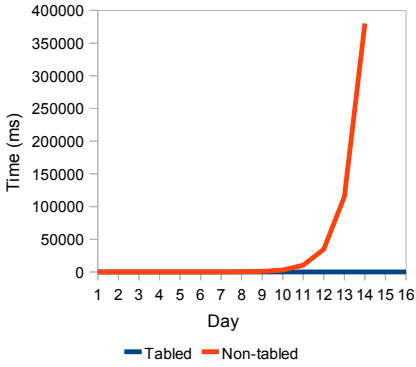
that we want to measure the effects on the first two steps and a second reason is that we know from experience that for our benchmarks most ROBDDs either will be built within one minute or run out of memory.

The results for the **weather benchmark** are in Figure 3.11 and in Table 3.9. Figure 3.11(a) shows that the SLD resolution execution times of the non-tabled version are exponential with respect to the "Day" argument, while the tabled version is linear. Figure 3.11(b) shows the scaling of the SLD resolution for queries that the non-tabled version fails to compute, as it exceeds the available memory.

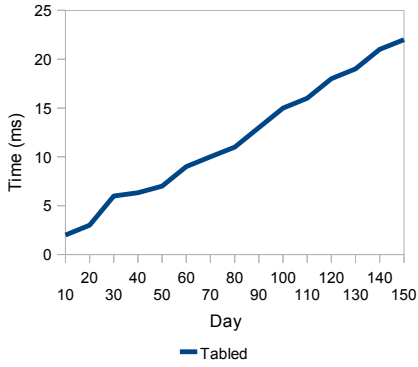
When we consider all three steps of ProbLog in Table 3.9, we see that the tabled version manages to compute "Day 167", while the non-tabled version stops at "Day 14". The tabled version is limited by the ROBDD compilation step that generates the ROBDD using a state-of-the-art ROBDD tool. Table 3.9 shows that the tabled version can continue the SLG-resolution step even further. For weather, the tabled version outperforms the non-tabled version in all stages including the ROBDD compilation.

Another interesting point is the memory usage to represent the proofs which goes from exponential to linear for the tabled version as shown in Figure 3.11(c). In Figure 3.11(d) we see the gain in memory is similar to the gain in time. This can be explained by the suffix sharing in the nested tries.

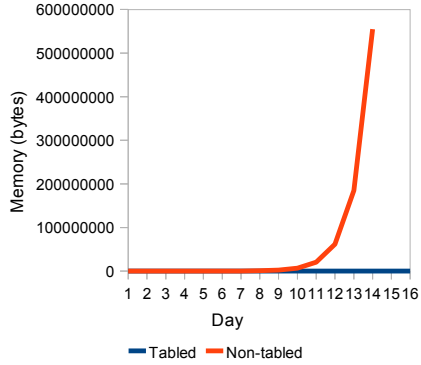
For the **bloodtype benchmark**, the results are in Table 3.10. The effect of increasing the input size, namely the number of generations, has also an impact on the interconnectivity and this clearly affects the results. For example, the run for 5 generations turns out to have less connectivity and thus faster execution times. The tabled version clearly outperforms the non-tabled version for the SLD-resolution and the Boolean formulae preprocessing. The slowdown for the ROBDD compilation for smaller problems is relative small. By tabling, we can collect the proofs even for the 15^{th} generation, where the non-tabled benchmark stops at the 3^{rd} generation.



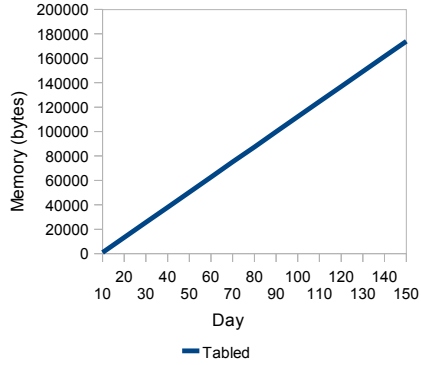
(a) Comparison of SLD/G resolution times for non-tabled versus tabled implementation.



(b) Day parameter impact on SLG resolution time in tabled implementation.

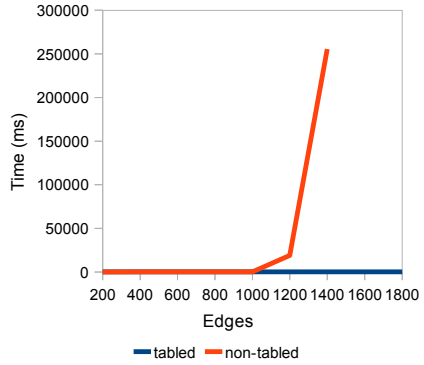


(c) Comparison of memory usage for non-tabled versus tabled implementation.

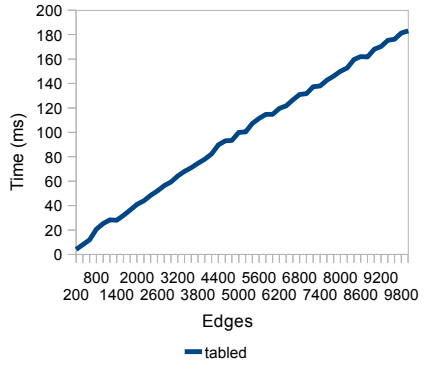


(d) Day parameter impact on memory consumption in tabled implementation.

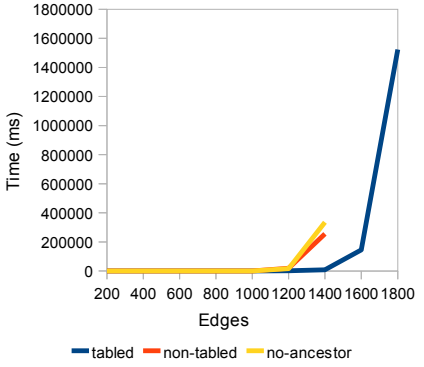
Figure 3.11: Tabling benchmarks for the weather program. ProbLog SLD/SLG resolution times are shown in (a), day parameter impact at time is shown in (b); memory consumption comparison is shown in (c), day parameter impact at memory is shown in (d).



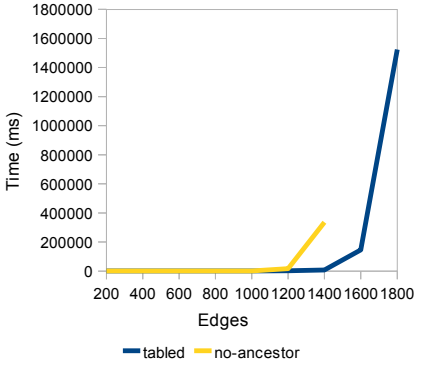
(a) Comparison of SLD/SLG resolution of non-tabled with tabled implementation



(b) Edges parameter impact on SLG resolution time in tabled implementation



(c) SLD/SLG resolution and Boolean formulae preprocessing comparison of non-tabled, tabled and tabled without ancestor subset check implementation



(d) Boolean formulae preprocessing comparison with/without ancestor subset check

Figure 3.12: Graph benchmark results.

Again ROBDD compilation is the limiting factor. This benchmark also illustrates that the difficulty in generating a ROBDD really depends on its input and not directly on the size of the problem.

In the **graph benchmark** we study the benefits of tabling in combination with cycle handling. As shown in Figure 3.12(a), the tabled version has a significant performance improvement for the SLD-resolution. Figure 3.12(b) shows that increasing the number of edges in the graph affects the SLG-resolution linearly.

	Memory (Bytes)		SLD/SLG resolution		Boolean formulae preprocessing		ROBDD compilation	
Gen	non-tab	tab	non-tab	tab	non-tab	tab	non-tab	tab
1	1048	1948	0	0	0	0	8	10
2	84568	9996	12	1	3	3	20	11
3	554824408	26092	125978	2	9566	6	80	166
4	-	34140	-	3	-	7	-	342
5	-	9996	-	1	-	2	-	8
6	-	130716	-	13	-	27	-	-
7	-	179004	-	17	-	38	-	44432
8	-	170956	-	16	-	35	-	40900
9	-	179004	-	17	-	37	-	-
10	-	187052	-	18	-	39	-	41916
11	-	179004	-	17	-	36	-	-
12	-	227292	-	22	-	46	-	-
13	-	235340	-	22	-	48	-	-
14	-	219244	-	21	-	45	-	-
15	-	211196	-	20	-	43	-	-

Table 3.10: Results for the bloodtype program. Times are in milliseconds.

Table 3.11 displays the results of running three different queries: Query 1: `path('HGNC_983','HGNC_582')`, Query 2: `path('HGNC_582','HGNC_983')` and Query 3: `path('HGNC_620','HGNC_582')` on the same graphs. The figures in Figure 3.12 correspond to the first query. The effect of tabling on the memory usage is a bit different now. Using nested tries for tabling favours suffix sharing rather than prefix sharing. It seems that in some graphs prefix sharing is more important for memory compaction than suffix sharing. However, in bigger graphs, the nested tries are again improving significantly memory consumption. Another observation is that the tabled version requires to construct the nested tries even for goals that fail or succeed without probabilistic facts. This introduces a significant minimum memory cost to the implementation.

Unfortunately, we notice in Figure 3.12(c) that all the versions behave exponential when summing up the SLD/SLG resolution times and the Boolean formulae preprocessing times. Figure 3.12(d) presents the times for generating the ROBDD scripts from the nested tries with cycles and the performance gain of the ancestor subset check. While the tabled version without the ancestor subset check is less efficient (total time) than the non-tabled version, the version with the ancestor subset check shows significant performance gains in all cases. Using the ancestor subset check, sometimes causes a small slowdown during the execution of the ROBDD scripts. This is due to non-minimal proofs introduced by reused cycles.

In all our benchmarks without tabling, we see that the cost of the first two steps is usually higher than the cost of the third step. But, while tabling makes the first step scale linearly and similarly improves the second step for some benchmarks, we notice sudden exponential blow ups for the third step, as is to be expected for a #P-hard problem.

Finally, we present in Table 3.12 the results of experiments related with the several different optimizations presented in Section 3.4. For these experiments we used the same experimental setting as we did for the graph based experiments, but in this case we use the built-in tabling implementation of Section 3.5.2 and we compare the optimizations related with the generation of ROBDD definitions from the nested tries. As before, we notice that having no optimization at all is the worst strategy and that using the first optimization: subset check is very important. With that in-mind all optimization experiments are made in conjunction with subset checking. Furthermore, we notice that generated ROBDD definitions with subset checking perform substantially worse than the non-optimized ROBDD definitions.

The **ancestor list refine** optimization aimed at increasing the reuse of unfolded nested tries and at decreasing the Boolean formulae preprocessing time. Comparison of the results of subset checking without and with ancestor list

		Memory (Kilobytes)		SLD/SLG resolution		Boolean formulae preprocessing			ROBDD compilation		
Query	Edges	non-tab	tab	non-tab	tab	non-tab	tab	no-anc	non-tab	tab	no-anc
	800	< 1	192	19	17	0	22	37	3	3	4
	1000	32	239	245	20	1	107	258	36	33	37
1	1200	3303	286	18928	25	28	1982	16844	119	214	196
	1400	39020	333	255546	28	473	7832	335853	455	454	278
	1600	-	380	-	33	-	145669	-	-	26789	-
	1800	-	426	-	37	-	1523948	-	-	-	-
	800	< 1	< 1	0	0	0	0	0	3	3	3
	1000	22	230	1253	20	1	86	3816	29	35	40
2	1200	2150	276	143276	24	17	62388	3367873	178	930	109
	1400	21545	323	1374688	28	203	58818	-	441	1712	-
	1600	-	368	-	33	-	584502	-	-	-	-
	1800	-	414	-	37	-	3425065	-	-	-	-
	800	< 1	192	23	16	0	19	25	5	3	4
	1000	16	239	99	20	0	68	203	5	5	8
3	1200	825	286	4160	25	6	1330	12198	125	406	232
	1400	3251	333	18745	29	24	1412	-	380	397	-
	1600	-	380	-	35	-	24037	-	-	28978	-
	1800	-	426	-	39	-	237624	-	-	-	-

Table 3.11: Results for the graph program. Times are in milliseconds.

refine confirm that the optimization achieves its target. Notice in Table 3.12 that there is a significant improvement in several queries for the Boolean formulae preprocessing time; also we notice an improvement in ROBDD compilation time.

As was previously explained, the pre-process optimization targets in reclaiming the lost performance, compared with the no optimization and subset check at the ROBDD compilation step. We notice that the performance for ROBDD compilation improves significantly comparing the subset check column with our without pre-process. In parallel, we also see that the Boolean formulae preprocessing time is improved. Depending on the query, pre-process performs better than ancestor list refine and vice-versa. Most of the times, we notice that using the ancestor list refine performs a bit better than using the pre-process and, because of that, we say that overall it is a bit better. As both optimizations are independent from each other, we can use both. Unfortunately, we do not really see an improvement compared from when we are using only one of the two. From that we conclude that the two optimizations are not compatible and possibly take advantage the same source of inefficiency.

The last column of Table 3.12 presents the results for using a different representation for the ancestors. We used the ancestor list refine optimization setting as it is slightly better in average than the other ones to see the impact of the different representation. The different representation obviously only affects the Boolean formulae preprocessing time and we can notice that there is an improvement. Especially for query 2 we can notice that this improvement is significant.

Finally, we want to mention that the results in Table 3.11 and Table 3.12, while they are from the same benchmarks and are using the same queries, origin from two different implementations. Namely our light weighted implementation of tabling and the built-in implementation of tabling. For that reason, the performance varies a bit. An important difference among the two implementations is that our light weighted tabling has a special treatment of cycle introducing derivations. This treatment orders differently the cycle introducing branches of nested tries. We have noticed that in specific cases such as the 2nd query for 1000 edges the order of the branches might play a big role in performance.

3.7 Conclusions

In this chapter we successfully identified the requirements for a tabling mechanism for ProbLog and presented two different program transformations in order

Query	Edges	No Optimization	Subset Check	Ancestor List Refine	Pre-process	Ancestor List Refine & Pre-process	Ancestor List Refine & Representation
		Boolean formulae preprocessing					
	1000	285	81	76	96	98	39
1	1200	5336	1380	504	608	618	295
	1400	72733	7891	497	657	633	250
	1000	3806	1050	670	827	828	243
2	1200	-	49459	11350	13545	13692	5409
	1400	-	58422	12100	14288	14408	5737
	1000	244	72	66	87	86	40
3	1200	59997	1175	431	525	529	274
	1400	-	994	437	550	556	286
		ROBDD compilation					
	1000	11	98	18	18	19	19
1	1200	386	1035	508	502	503	511
	1400	410	1904	569	585	584	569
	1000	90	375	217	89	90	219
2	1200	-	3322	2757	1726	1728	2740
	1400	-	7336	3129	2885	2854	3002
	1000	17	18	18	18	18	18
3	1200	645	1119	511	571	531	510
	1400	-	1057	667	682	585	665

Table 3.12: Results for the graph program by the use of different optimization options. Times are in milliseconds.

to extend ProbLog with a tabling mechanism. The first transformation actually realizes a light weight tabling mechanism while the second transformation uses the built-in tabling mechanism of the underlying Prolog system.

We also introduced a new data structure namely nested tries where ProbLog tabling records the relevant part of the SLD tree of grounded goals, namely the probabilistic facts that are used in the proofs. In ProbLog terms, the tries of clause identifiers are tabled. When a goal has to be re-computed, ProbLog just reuses the tabled trie creating a nested structure. Nested tries establish prefix and suffix sharing. Furthermore, we presented how cycle handling can be performed using the nested tries.

Our experiments have shown that tabling is definitely beneficial for the SLD-resolution step, where the memory and time consumption can go from exponential to linear. Nested tries establish prefix and suffix sharing which are crucial for good results.

Tabling also affects the next ROBDD related steps of ProbLog. For benchmarks without cycles, tabling further reduces the execution times, as these steps also benefit from the compaction by the sharing. In the graph benchmark, we see that tabling improves the overall performance of the system. While the improvement for SLD resolution is remarkable, the work is partly transferred to the Boolean formulae preprocessing step. Our approach encodes the cycles of the SLD resolution by the nested tries. The ancestor subset check is indispensable during the Boolean formulae preprocessing.

Finally, we presented further optimizations related with the generation of ROBDD definitions and we did a short experimental evaluation of them. We can see that while both ancestor list refinement and pre-process optimizations provide a significant improvement, unfortunately they appear not to co-operate for an even greater gain. A further improvement, that is important for performance, is the representation chosen for the ancestors. We saw that we can have a significant performance gain when the task is hard.

Chapter 4

Preprocessing I: Boolean Formulae to ROBDD Definitions

This chapter focuses on the different approaches for generating ROBDD definitions from a Boolean formula. First, we present three different approaches namely: naive, decomposition [66] and recursive node merging [35] in Sections 4.2, 4.3 and 4.4 respectively. All three approaches generate a script of ROBDD definitions from a single trie that represents a DNF Boolean formula. In Section 4.5, we present a new structure named depth breadth trie (dbtrie) that allows further optimization of ROBDD definitions before the script generation. We combine the recursive node merging method with the dbtrie structure. Section 4.6 presents a complexity analysis of all presented preprocessing algorithms. We verify the benefits and claims experimentally in Section 4.7. Finally, Section 4.8 generalizes our approaches to be usable with the previously presented nested tries and with Boolean formulae different from DNFs through the introduction of negation.

As is shown in the experimental part in Section 4.7, all of the presented approaches significantly outperform the naive method. But unfortunately depending on the structure of the Boolean formula to be processed there is no single best method. Figure 4.1 presents graphically the ProbLog parts that are modified by preprocessing.

This chapter is based on the publication [47]. New with respect to [47] is the extension of the approaches to nested tries and the support for general negation.

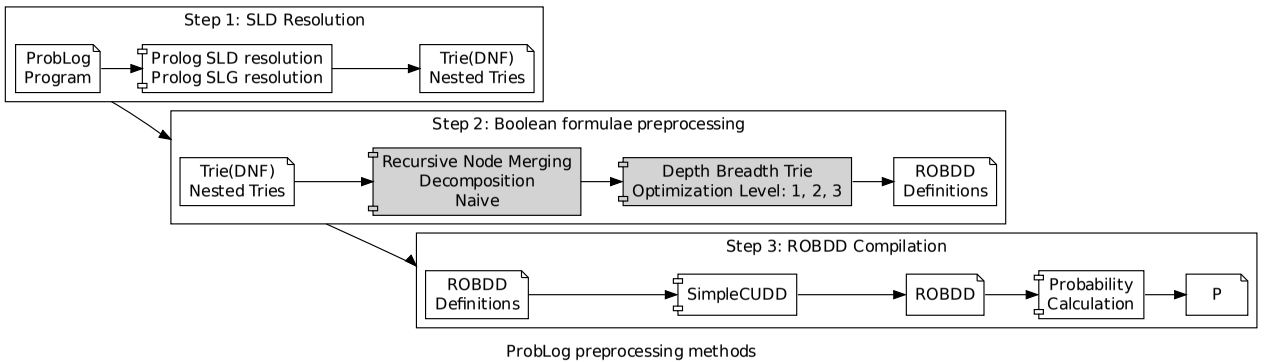
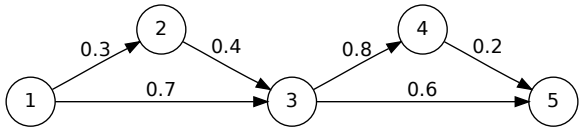


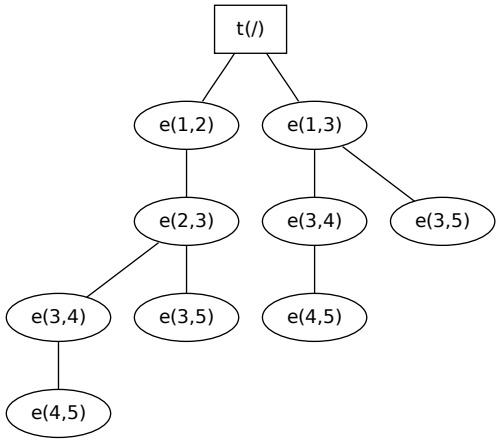
Figure 4.1: The diagram of the three steps of ProbLog with the preprocessing additions. Shaded blocks mark the modifications from the block figure of Figure 2.2.



(a) A probabilistic graph without cycles.

$$
\begin{array}{ll}
e(1,2) \wedge e(2,3) \wedge e(3,4) \wedge e(4,5) & \vee \\
e(1,2) \wedge e(2,3) \wedge e(3,5) & \vee \\
e(1,3) \wedge e(3,4) \wedge e(4,5) & \vee \\
e(1,3) \wedge e(3,5) &
\end{array}
$$

(b) The collected proofs for the graph by the query: `path(1,5)` in their DNF Boolean formula form.



(c) The collected trie for the graph by the query: `path(1,5)`.

Figure 4.2: The graph together with the Boolean formula in DNF form and the collected trie by the query: `path(1,5)`, used as an example.

```
L1 = e(1,2)*e(2,3)*e(3,5)
L2 = e(1,2)*e(2,3)*e(3,4)*e(4,5)
L3 = e(1,3)*e(3,5)
L4 = e(1,3)*e(3,4)*e(4,5)
L5 = L1+L2+L3+L4
```

(a) Naive

```
L1 = e(3,4)*e(4,5)
L2 = e(3,5)+L1
L3 = e(2,3)*L2
L4 = e(3,4)*e(4,5)
L5 = e(3,5)+L4
L6 = e(1,3)*L5
L7 = e(1,2)*L3
L8 = L7+L6
```

(b) Decomposition

```
L1 = e(3,4)*e(4,5)
L2 = e(3,5)+L1
L3 = e(1,3)*L2
L4 = e(2,3)*L2
L5 = e(1,2)*L4
L6 = L3 + L5
```

(c) Recursive Node Merge

```
L1 = e(3,4)*e(4,5)
L2 = e(3,5)+L1
L3 = e(1,3)*L2
L5 = e(1,2)*e(2,3)*L2
L6 = L3+L5
```

(d) Depth Breadth Trie

Table 4.1: ROBDD Definitions of the Boolean Formula of Figure 4.2 generated from each preprocessing method.

4.1 Preprocessing Example

The running example of this chapter is shown in Figure 4.2. The graph used is in Figure 4.2(a), the DNF Boolean formula is in Figure 4.2(b) and finally the collected trie is in Figure 4.2(c).

Table 4.1 presents the resulting ROBDD definitions generated by the four different methods used for the example in Figure 4.2.

4.2 Naive

The naive method directly mirrors the structure of the DNF Boolean formula by first constructing all conjunctions of the DNF Boolean formula and then combining those in one big disjunction. Table 4.1(a) shows, at the top left, the resulting script for the proofs of our example.

4.3 Decomposition

The decomposition method [66] recursively divides a DNF Boolean formula into smaller ones until only one variable remains. To do so, it first chooses a Boolean variable from the formula, the so-called decomposition variable dv, and then breaks the formula into three subformulae. The first subformula f_1' contains the conjunctions that include dv, the second subformula f_2' those that include the negation of dv and the third subformula f_3 those that include neither of the two. Then, by applying the distribution axiom, the original Boolean formula can be re-written as $f = f_1' \vee f_2' \vee f_3 = (dv \wedge f_1) \vee (\neg dv \wedge f_2) \vee f_3$. Up to now we have not presented examples where negated literals like $\neg dv$ occur; later in Section 4.9.2 we discuss further about negated literals and also in Chapter 6 we fully explain general negation and give examples where it is used.

As all three new subformulae f_1, f_2 and f_3 do not contain dv, they can be decomposed independently. The most simple choice for the decomposition variable is the first variable of the current formula, however, various heuristic functions can be used as well, cf. [66]. Algorithm 4.1 formalizes the decomposition method, while Table 4.1(b) again shows, at the top right, the result for our example query. All definitions resulting from the same decomposition step are written as a block at the end of that step, omitting those equivalent to false to avoid unnecessary ROBDD operations.

For example, if for the DNF of Figure 4.2(b) we choose as a decomposition variable $e(1,2)$ then the DNF is broken in three parts: $f_1 = (e(2,3) \wedge e(3,4) \wedge e(4,5)) \vee (e(2,3) \wedge e(3,5))$, $f_2 = false$ and $f_3 = (e(1,3) \wedge e(3,4) \wedge e(4,5)) \vee (e(1,3) \wedge e(3,5))$. The algorithm first further decomposes f_1 by choosing $e(2,3)$ as the decomposition variable to: $f_1 = (e(3,4) \wedge e(4,5)) \vee e(3,5)$, $f_2 = false$ and $f_3 = false$. At the next recursion variable $e(3,4)$ is chosen and the produced formulae are: $f_1 = e(4,5)$, $f_2 = false$ and $f_3 = e(3,5)$. Finally, the first ROBDD definition is written as: $L_i = e(3,4) * f_1$, $L_j = e(3,4) * f_2$, $L_k = L_i + L_j + f_3 \rightarrow L_i = e(3,4) * e(4,5)$, $L_j = e(3,4) * false$, $L_k = L_i + Lj + e(3,5) \rightarrow L1 = e(3,4) * e(4,5)$ and $L2 = e(3,5) + L1$. Now the algorithm can return to the previous step and similarly produce $L3 = e(2,3) * L2$ and so forth.

4.4 Recursive Node Merging

The approach followed in ProbLog, as described in [35], exploits the sharing of both prefixes – as directly given by the tries – and suffixes, which have

Algorithm 4.1 Decomposition method.

input A DNF Boolean formula and a start index i for ROBDD definitions.

output The ROBDD definitions for the Boolean formula in a script file, the next available ROBDD definition index j and L_n the last ROBDD definition written.

```
function DECOMPOSE(DNF, i)
  if DNF = true or DNF = false or DNF = {{dv}} then
    return (DNF, i)
  f1 := false; f2 := false; f3 := false
  dv := CHOOSE_DECOMPOSITION_VARIABLE(DNF)
  for all conj ∈ DNF do
    if dv ∈ conj then
      drop dv from conj to obtain c
      f1 := f1 ∨ c
    else if ¬dv ∈ conj then
      drop ¬dv from conj to obtain c
      f2 := f2 ∨ c
    else
      f3 := f3 ∨ conj
  (L_f1, j) := DECOMPOSE(f1, i)
  (L_f2, k) := DECOMPOSE(f2, j)
  (L_f3, l) := DECOMPOSE(f3, k)
  write L_l = dv * L_f1
  write L_{l+1} = ¬dv * L_f2
  write L_{l+2} = L_l + L_{l+1} + L_f3
  return (L_{l+2}, l + 3)
```

to be extracted algorithmically. We will call this approach **recursive node merging**.

Recursive node merging traverses the trie representing the DNF bottom-up. In each iteration it applies two different operations that reduce the trie by merging nodes. The first operation (**depth reduction**) creates the conjunction of leaf nodes that are a single child with their parents. The second operation (**breadth reduction**) creates the disjunction of all children nodes of a node, when these children nodes are all leaves. In our new implementation of this method, depth reduction can be applied to an arbitrary chain of ancestor nodes with a single child.

Algorithm 4.2 shows the details for this method and Figure 4.3 illustrates its step-by-step application to the example trie in Figure 4.2(c). The resulting

script can be found on the bottom left in Table 4.1(c).

Algorithm 4.2 Recursive node merging. REPLACE(T, C, L_i) replaces each occurrence of C in T by L_i.

input A trie T representing the DNF Boolean formula.
output The ROBDD definitions for the Boolean formula in a script file.

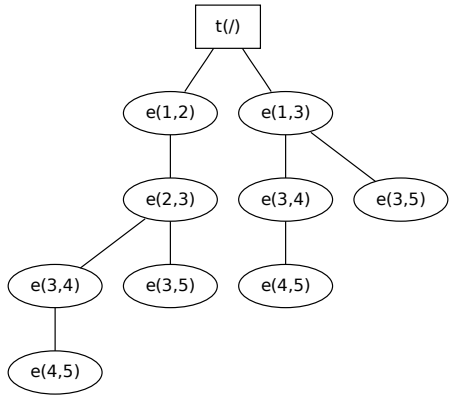
```
function RECURSIVE_NODE_MERGING(T)
  i := 1
  while ¬leaf(T) do
    S_∧ := {(C, P)|C leaf in T and single child of P}
    for all (C, P) ∈ S_∧ do
      write L_i = C * P
      T := REPLACE(T, (C, P), L_i)
      i := i + 1
    S_∨ := {([C_1, ..., C_n], P)| leaves C_j are all the children of P in T, n > 1}
    for all ([C_1, ..., C_n], P) ∈ S_∨ do
      write L_i = C_1 + ... + C_n
      T := REPLACE(T, [C_1, ..., C_n], L_i)
      i := i + 1
```

For both reduction types, a subtree that occurs multiple times in the trie is reduced only once, and the resulting conjunction/disjunction is used for all occurrences of that subtree (procedure REPLACE() in Algorithm 4.2). Recursive node merging thus performs some suffix sharing.

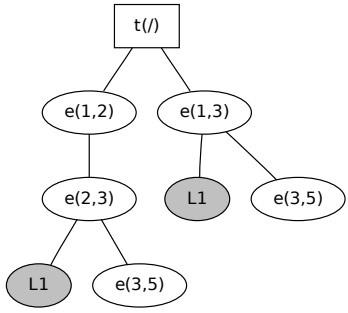
Because of the internal structure of tries it is not necessary to explicitly collect the leaf nodes at each iteration (represented by the terms $S_\wedge, S_\vee$); we can instead just ask for the next leaf of the trie. Note however, that the REPLACE() procedure fully traverses the trie to search for repeated occurrences of subtrees and that can be quite costly. Actually, in the worst case scenario when no replacement occurs the procedure increases the complexity for each operation from $O(N)$ to $O(N^2)$. Furthermore, the original implementation of the recursive node merging algorithm worked on a list based representation of tries which required the transformation of the trie to a set of lists before performing the algorithm it self.

4.5 Depth Breadth Trie

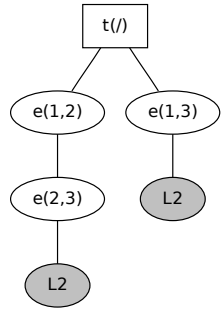
In this section, we introduce our new approach to implement recursive node merging. The initial implementation explicitly performed the costly REPLACE()



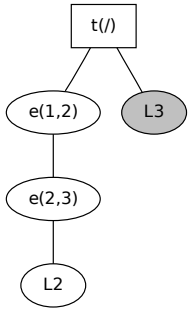
(a) Original Trie.



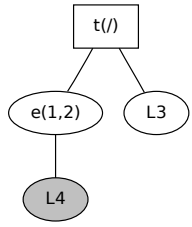
(b) Step 1.



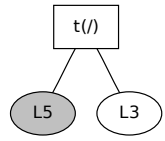
(c) Step 2.



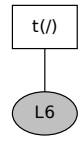
(d) Step 3.



(e) Step 4.



(f) Step 5.



(g) Step 6.

Figure 4.3: Execution of the recursive node merging shown in Algorithm 4.2 for the example trie of Figure 4.2.

procedure of Algorithm 4.2. The new approach avoids this by storing all ROBDD definitions in a central data structure. For each new definition, we first check if it is already present in the data structure, and if so, reuse the corresponding reference L_i. As such a check/insert operation can be done in a single pass for tries, we introduce an additional and specific trie configuration for this purpose, that we named **depth breadth trie**. Apart from this improvement, the depth breadth trie has the additional advantage of allowing one to easily identify common prefixes on the level of ROBDD definitions, which was not possible before. As we will see, this leads to the definition of three new (optional) optimizations that can be performed during recursive node merging to further reduce the number of Boolean operations to be performed in ROBDD construction.

A depth breadth trie is divided in two parts corresponding to the two reduction types of recursive node merging: the **depth part** collects the conjunctions, the **breadth part** the disjunctions. This separation is achieved by two specific functors of arity two, `depth/2` and `breadth/2`. Their first argument is a Prolog list containing the literals that participate in the formula, the second argument is the unique reference L_i assigned to the corresponding ROBDD definition. For example, the terms `depth([e(3,4),e(4,5)],L1)` and `breadth([e(3,5),L1],L2)` represent definitions $L1 = e(3,4) * e(4,5)$ and $L2 = e(3,5) + L1$ respectively. Note that reference `L1` introduced by the first term is used in the second term to refer to the corresponding subformula. At the same time, those references provide the order in which ROBDDs are defined in the script. Figure 4.4 shows the complete depth breadth trie built by recursive node merging for our example DNF, which produces the ROBDD definitions shown in Table 4.1(c).

In the following, we introduce the three new optimizations that can be exploited with the depth breadth trie. The motivation behind these optimizations is to decrease the amount of operations performed in ROBDD construction. The optimizations are illustrated in Figure 4.5. Figure 4.5(a) presents the initial trie used in all cases, Figures 4.5(b), 4.5(c) and 4.5(d) correspond to Optimizations I, II and III, respectively.

Optimization I (Contains Prefix): The first optimization occurs when a new formula $[p_1, \ldots, p_n]$ to be added to the depth breadth trie contains as prefix an existing formula $[p_1, \ldots, p_i]$, $i \geq 2$, with reference L_n. In this case, the existing formula will be reused and, instead of inserting $[p_1, \ldots, p_n]$, we will insert $[L_n, p_{i+1}, \ldots, p_n]$ and assign a new reference to it.

Optimization II (Is Prefix): The second optimization considers the inverse case of the first optimization. It occurs when a new formula $[p_1, \ldots, p_i]$,

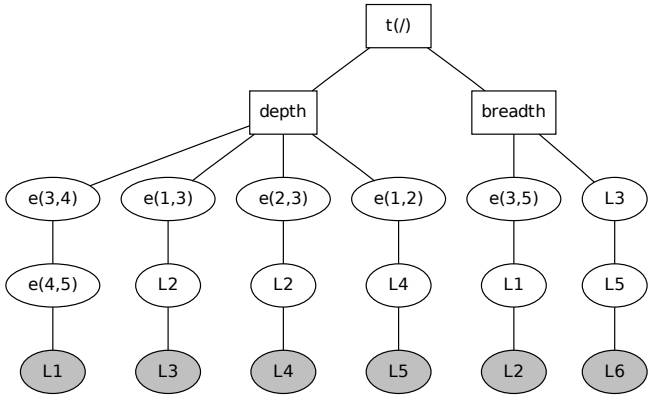
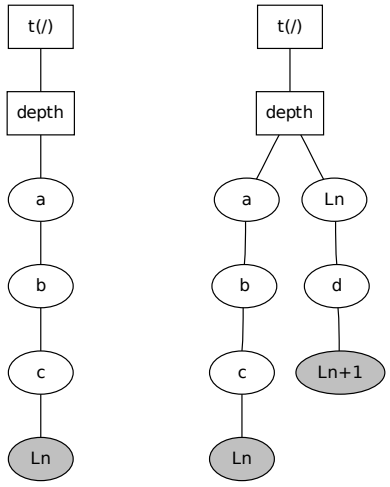


Figure 4.4: The ROBDD definitions generated by using recursive node merging as in Algorithm 4.2 collected in a depth breadth trie.

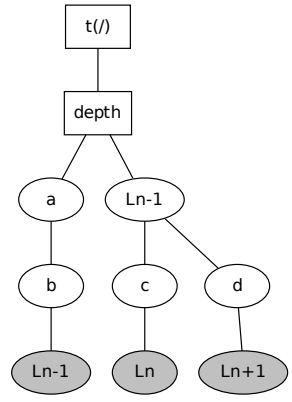
$i \geq 2$, to be added to the depth breadth trie is a prefix of an existing formula $[p_1, \ldots, p_n]$ with reference L_n. In this case, we split the existing subformula representing $[p_1, \ldots, p_n]$ in two: one representing the new formula $[p_1, \ldots, p_i]$ with a new reference L_{r-1}, the other representing the existing formula, but modified to re-use the new reference L_{r-1}, that is, $[p_1, \ldots, p_n]$ is replaced by $[L_{r-1}, p_{i+1}, \ldots, p_n]$.

Optimization III (Common Prefix): The last optimization generalizes the two previous ones. It occurs when a new formula $[p_1, \ldots, p_n]$ being added to the depth breadth trie has a common prefix $[p_1, \ldots, p_i]$, $i \geq 2$, with an existing formula $[p_1, \ldots, p_i, p'_{i+1}, \ldots, p'_m]$ with reference L_n. In this case, first, the common prefix is inserted as a new formula with reference L_{r-1}, triggering the second optimization, and second, the original new formula is added as $[L_{r-1}, p_{i+1}, \ldots, p_n]$ using L_{r-1} as in the first optimization.

Each repeated occurrence of a prefix of length P identified by one of the optimizations decreases the total number of operations required by $P - 1$. Consider that M and N are the lengths of formulae f_M and f_N, respectively, and that f_N is a prefix of f_M. Then, optimizations I and II decrease the amount of operations from $N - 1 + M - 1$ to $N - 1 + M - N = M - 1$, this is a total decrease of $(N - 1 + M - 1) - (M - 1) = N - 1$ and as f_N was the prefix formula then $N = P$. For Optimization III, consider two formulae f_M and f_N of length M and N respectively with common a prefix f_P of



(a) Initial trie (b) Adding $[a, b, c, d]$ (c) Adding $[a, b]$



(d) Adding $[a, b, d]$

Figure 4.5: Examples that trigger (b) Optimization I, (c) Optimization II and (d) Optimization III for the depth breadth trie.

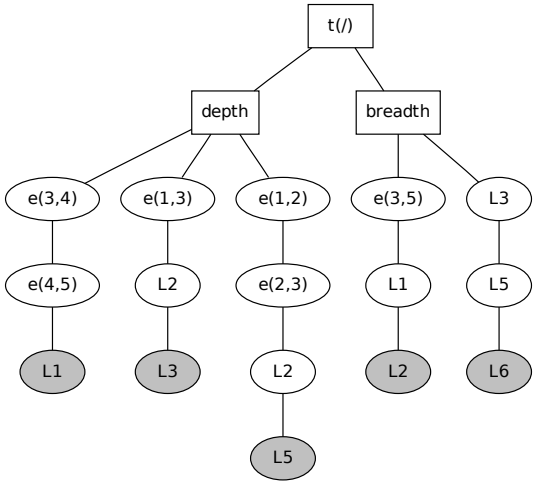


Figure 4.6: The ROBDD definitions generated by using recursive node merging with multiple depth node reduction collected in a depth breadth trie.

length P, the number of operations decreases from $(N-1)+(M-1)$ to $(P-1)+(N-P)+(M-P) = N+M-P-1$, and if a third formula f_K shares the same prefix, the number of operations it requires again reduces to $(K-1)-(P-1) = K-P$.

Algorithm 4.3 formalizes the implementation of these three optimizations. One should notice that with the depth breadth trie, the references L_i are no longer incremented by one but by the length of the formula being added. This is necessary as Optimizations II and III insert additional subformulae that have to be created before the current formula is added, and thus need to be assigned a smaller reference. As our formulae always contain at least two elements, using the length of the formula to increment L_i is sufficient to leave enough free places for later use with Optimizations II and III. Figure 4.6 presents the depth breadth trie of the same DNF by taking in account multiple depth node reductions and having longer than one increment for references. The ROBDD definitions for this depth breadth trie are shown at Table 4.1(d).

The order given by those references will therefore ensure that subformulae will be generated before being referred to. Moreover, as we are using tries, these optimizations can be performed while adding the new formulae. During insertion, if the new definition branches off an existing one after two or more steps, the

insertion of the new formula is frozen while the appropriate optimization is performed and resumed afterwards.

Our implementation in fact offers four choices of optimization level: no optimizations, Optimization I only, Optimizations I and II, or all three optimizations. Furthermore, the minimal length of common prefixes can be adapted. Note that depending on the order in which the formulae are inserted, different optimizations might trigger and the resulting trie might be slightly different.

Finally, we want to mention that all different preprocessing methods (naive, decomposition, recursive node merging) can be easily modified to use a depth breadth structure in order to store the generated ROBDD definitions.

4.6 Complexity of Preprocessing Methods

This section analyses the worst case complexity of our preprocessing algorithms. We use N to denote the number of proofs or conjunctions and M to denote the number of probabilistic facts or Boolean variables. From our experience, $N >> M$ usually holds for typical ProbLog programs.

4.6.1 Naive Method

The naive approach writes N conjunctions, each with a maximum of M Boolean variables, and one disjunction of length N. Thus, its worst case complexity is $O(N \cdot M + N) = O(N \cdot M)$.

4.6.2 Decomposition Method

The decomposition method repeatedly scans a DNF to assign conjunctions to subformulae which are then in turn decomposed further. If decomposing on the first variable of the current formula, the decomposition variable is determined in constant time. In the worst case, the N conjunctions of the initial DNF are of length M each. Thus, the first decomposition step needs to scan the entire formula, corresponding to a cost of $N \cdot M$. This then results in three recursive calls for each part, respectively, with N_1, N_2 and N_3 conjunctions subject to the constraint $N = N_1 + N_2 + N_3$, with maximal length $M - 1$ for each conjunction. Decomposing the i^{th} part in turn requires scanning with cost $N_i \cdot (M - 1)$, that is, the total cost of the second level is

Algorithm 4.3 Depth breadth trie optimizations. $COUNTER$ is a global counter and REPLACE$(Literals, C, L_i)$ replaces C in $Literals$ by L_i.

input A trie T representing either the depth or breadth part of the depth breadth trie and a list $Literals$ with the literals to be added.

output Updates T to contain the $Literals$ and returns the reference L_i assigned to $Literals$ in T.

```
function UPDATE_DEPTH_BREADTH_TRIE(T, Literals)
  if (Literals, L_i) ∈ T then
    return L_i
  for all (List, L_i) ∈ T do
    if List is prefix of Literals then
      /* Optimization I */
      Literals := REPLACE(Literals, List, L_i)
      return UPDATE_DEPTH_BREADTH_TRIE(T, Literals)
    if Literals is prefix of List then
      /* Optimization II */
      T := REMOVE((List, L_i), T)
      T := ADD((Literals, L_{i-LENGTH(Literals)}), T)
      List := REPLACE(List, Literals, L_{i-LENGTH(Literals)})
      T := ADD((List, L_i), T)
      return L_{i-LENGTH(Literals)}
    if Literals, List have a common prefix Prefix and LENGTH(Prefix) > 1
    then
      /* Optimization III */
      L_j := UPDATE_DEPTH_BREADTH_TRIE(T, Prefix)
      Literals := REPLACE(Literals, Prefix, L_j)
      return UPDATE_DEPTH_BREADTH_TRIE(T, Literals)
  Counter := Counter + LENGTH(Literals)
  T := ADD ((Literals, L_Counter), T)
  return L_Counter
```

$N_1 \cdot (M-1) + N_2 \cdot (M-1) + N_3 \cdot (M-1) = (N_1 + N_2 + N_3) \cdot (M-1) = N \cdot (M-1)$. Again, it then results in three recursive calls for each part, with N_{i1}, N_{i2} and N_{i3} conjunctions respectively, with maximal length $M-2$ for each conjunction, and subject to the constraint $N_i = N_{i1} + N_{i2} + N_{i3}$. In general, due to the fixed number of conjunctions to be assigned to subformulae, the total cost of the j-th decomposition level is $N \cdot (M+1-j)$. As the number of such levels is bound by the number of variables M that can be used for decomposition, the overall cost is $\sum_{j=1}^{M} N \cdot (M+1-j)$ and thus $O(N \cdot M^2)$.

4.6.3 Recursive Node Merging

Recursive node merging complexity strongly depends on the amount of prefix sharing in the initial trie. In the following analysis, we will consider our new approach using the depth breadth trie.

In the first iteration of the algorithm, it is possible that no conjunctions (depth reductions) are found, in which case the size of the trie is not reduced. However, in later iterations, depth reduction will always reduce the depth of the trie by at least one. Likewise, breadth reduction will always reduce the number of leaves by at least one, that is, after $O(min(M, N))$ iterations, either the depth or the breadth of the trie will be reduced to one and the algorithm will terminate with the next iteration. In each iteration, both depth and breadth reductions will touch each leaf, leading to a complexity of $O(N)$ for each iteration.

Viewed over all iterations, depth reduction will scan and reduce each branch of the trie once, starting from the leaves, and process each node once leading to a complexity of $O(\#nodes) = O(N \cdot M)$.

Obtaining a reference for each depth and breadth reduction corresponds to a check/insert operation on the depth breadth trie, which is linear in the number of nodes the reduction term contains, plus the number of sibling nodes visited. Note however that searching through a chain of sibling nodes that represent alternative paths in the trie could be too expensive if we have a large number of nodes. To avoid this problem, a threshold value (8 in our implementation) controls whether to dynamically index the sibling nodes through a hash table, hence providing direct node access and optimizing search. Further hash collisions are reduced by dynamically expanding the hash tables. As each node in the proof trie participates in one reduction, for the total execution, the cost of obtaining references is thus bound by $O(8 \cdot \#nodes) = O(N \cdot M)$.

Therefore, the overall complexity of recursive node merging using a depth breadth trie is $O(min(M, N)) \cdot O(N) + O(\#nodes) + O(\#nodes) = O(N \cdot M)$. While the constant factor is higher than for the naive method, in practice, this

is typically outweighted by the fact that prefix sharing causes the number of nodes in the proof trie to be far less than $N \cdot M$.

4.7 Experimental Results

We report on experiments comparing the four preprocessing methods: naive, decomposition (**dec**), recursive node merging as described in [35] (**rnm**) and recursive node merging with the depth breadth trie (**dbt**). The environment for our experiments was a C2Q 2.83 GHz 8 GB machine running Linux. We use Yap Prolog 6.0 and the CUDD ROBDD package [80] with the group sifting algorithm for variable reordering [53].

As **rnm** has been developed to exploit structure sharing in the trie, whereas **dec** is a general purpose method, we consider two benchmarks that show the difference in this aspect. The first is a three-state Markov model, where we query for the probability of a sequence of length N ending in a given state. Proofs for such queries share large amounts of structure. The second is the domain of connectivity queries in biological graphs that originally motivated ProbLog. In this case, we consider two different ProbLog inference methods which lead to different types of formulae. Exact inference as discussed in Chapter 2 produces formulae with high sharing in both prefixes and suffixes of proofs (which correspond to paths starting and ending at specific nodes). On the other hand, upper bound formulae as encountered in bounded approximation [35] typically contain small numbers of proofs and large numbers of so-called **stopped derivations**, i.e. partial proofs cut off at a probability threshold. While the latter still share prefixes, their suffixes are a lot more diverse. This type of formulae has been observed to be particularly hard for ProbLog. In our experiments, we use a 144 edge graph extracted from the Biomine network also used in [35], and query for acyclic paths between given pairs of nodes. We use a set of 45 different queries, some chosen randomly, and some maximizing the degrees of both nodes.

We thus set up experiments to study the following questions:

Q1: How do the different preprocessing methods compare on more, or less structured problems?

Q2: What is the impact of each optimization and which parameters affect them?

Table 4.2 presents the results for the Markov model. In this benchmark, no extra optimizations are triggered for **dbt**. Times are given in milliseconds

	Preprocessing				ROBDD compilation			
N	**naive**	**dec**	**rnm**	**dbt**	**naive**	**dec**	**rnm**	**dbt**
1	2	2	0	0	6	6	6	6
2	3	3	0	0	6	6	6	6
3	3	5	1	0	6	6	6	6
4	5	14	3	1	7	7	7	7
5	12	48	10	1	22	27	27	24
6	36	187	39	2	95	33	27	24
7	94	518	93	5	251	45	28	25
8	286	1,934	219	8	786	87	28	25
9	974	7,142	676	25	3,698	188	31	25
10	3,313	26,293	2,369	92	20,854	539	35	29
11	11,109	109,407	7,316	346	256,330	1,638	43	31
12	37,745	442,805	22,759	1,076	/	5,024	96	31
13	/	-	73,631	2,733	/	-	205	48
14	/	-	-	10,100	/	-	-	75

Method **dbt** is implemented in C, other preprocessing methods in Prolog.

Table 4.2: Average runtimes for preprocessing and ROBDD compilation on three-state Markov model, for sequence length N.

and are averages over three runs. ROBDD compilation uses a timeout of 600 seconds. When a method reaches this timeout, it is not applied to larger problems. These cases are marked with (/). Cases marked with (-) fail due to memory during script preprocessing; this also occurs when using **dbt** at length 15. Note that **dbt** is implemented in C, while the other preprocessing methods are implemented in Prolog, which partly accounts for the time differences. In this experiment, both trie based methods clearly outperform the naive and **dec** methods, and **dbt** also clearly outperforms **rnm**. In particular, preprocessing with **rnm** is one order of magnitude slower than with **dbt**, and also leads to somewhat slower ROBDD compilation. As a first answer to **Q1**, we thus conclude that for structured problems, **dbt** is indeed the first choice.

For the graph domain, we use a timeout of 300 seconds, and a cutting threshold $\delta = 0.05$ for obtaining upper bound formulae. As the naive method always performs worst, it is excluded from the following discussion. Typically, all optimizations for the **dbt** method are triggered, with type I being most frequent (note that type III increases type I usage).

In exact inference, cf. Table 4.3, ROBDD compilation times for all methods are very close and rarely exceed 4 seconds. However, preprocessing time for **dec** is one and two orders of magnitude higher than for **rnm** and **dbt**, respectively. Again, this is due to the high amount of suffix sharing in those tries, which is

Method	Preprocessing		ROBDD compilation		Sum	
	avg	sdev	avg	sdev	avg	sdev
dec	40,076	38,417	2,235	1,313	42,312	39,543
rnm	3,694	3,632	1,844	1,150	5,538	4,110
dbt	124	117	1,998	1,318	2,123	1,355
dbt1	125	118	1,891	1,697	2,016	1,708
dbt2	125	118	1,481	630	1,607	667
dbt3	128	120	1,446	769	1,575	800

Average and standard deviation of times over 45 queries.
Method **dbt**i uses all optimization levels $l \leq i$.

Table 4.3: ROBDDs for exact inference over the graph domain.

exploited by our method, but causes repeated work during construction for the decomposition method. This also explains the slightly higher ROBDD execution times for **dec**. Again these results clearly enforce our first conclusions about **Q1**. Regarding **Q2**, these results show that our optimizations are incrementally effective in reducing construction time without introducing costs in preprocessing time.

For upper bound ROBDDs, the results are more diverse. Here, we focus on the comparison between **dec** and **dbt** with different optimization levels, where **dbt**i refers to **dbt** using all optimization levels $l \leq i$. For presentation of results in Table 4.4, we partition queries in categories by using two thresholds on ROBDD compilation time ($t < 15,000ms$ for **Easy**, $15,000ms \leq t < 50,000ms$ for **Medium** and $t \geq 50,000ms$ for **Hard**), and use majority vote among the methods different preprocessing methods to decide the query's class (as one single test query finishes in few milliseconds, it is omitted from the results). The last column gives the number of queries reaching the timeout in ROBDD compilation.

Compared to exact inference, preprocessing times for **dec** are lower, as significant parts of the tries are unique and only small parts cause redundant work. On average, ROBDDs obtained by **dec** have smaller construction times than those obtained from **dbt**, even though variation is high. Table 4.5 compares methods by counting the number of queries for which they achieve fastest upper bound ROBDD compilation. Together, those results provide the second part of the answer to **Q1**: for problems with less suffix sharing, scripts obtained from **dec** often outperform those obtained from **dbt**.

Concerning the optimization levels, Table 4.4 indicates that for the graph case, their effect varies greatly. For all levels, we observe cases of improvement as well as deterioration. We suspect that optimizations are often performed to greedily.

Group	Method	Preprocessing avg	Preprocessing sdev	ROBDD compilation avg	ROBDD compilation sdev	Sum avg	Sum sdev	Time outs
Easy 19/44	**dec**	1,648	261	9,351	2,323	10,999	2,371	0
	dbt	17	2	10,148	2,192	10,165	2,192	0
	dbt1	16	2	10,714	2,389	10,730	2,389	0
	dbt2	16	2	14,417	3,263	14,433	3,263	0
	dbt3	17	2	15,311	4,055	15,328	4,055	0
Medium 14/44	**dec**	2,086	414	21,785	5,197	23,871	5,325	0
	dbt	23	3	29,719	4,029	29,743	4,029	0
	dbt1	24	3	39,914	9,084	39,938	9,083	0
	dbt2	21	3	28,522	3,165	28,543	3,163	0
	dbt3	23	3	46,263	19,231	46,286	19,230	0
Hard 11/44	**dec**	2,066	179	28,979	9,172	31,045	9,093	0
	dbt	22	2	62,612	16,350	62,635	16,353	3
	dbt1	21	2	121,442	29,052	121,464	29,053	2
	dbt2	20	3	94,454	28,753	94,476	28,755	3
	dbt3	22	3	122,150	37,751	122,172	37,753	3

Average and standard deviation of times over 44 queries queries grouped into categories according to runtimes. Method **dbt**i uses all optimization levels $l \leq i$.

Table 4.4: ROBDDs for upper bounds at threshold 0.05.

Method	All	rnm + dbt	rnm + All dbt	All dbt
dec	26	-	-	-
rnm	2	14	6	-
dbt	6	30	13	16
dbt1	2	-	9	10
dbt2	5	-	9	9
dbt3	3	-	7	9

Method **dbt**i uses all optimization levels $l \leq i$.

Table 4.5: Upper bound ROBDDs: number of queries where method leads to fastest ROBDD compilation, for various sets of methods.

Initial experimentation on artificially created formulae indicate that several factors influence performance of optimizations, among which are: (i) the length of the shared prefix; (ii) the number of times it occurs; (iii) the structure of the subformulae occurring in the prefix; and (iv) the structure of the suffixes sharing the prefix. While the latter three are harder to control during preprocessing, we performed a first experiment where we only trigger the optimizations for shared prefixes of minimal length $n > 2$. Results on upper bound formulae confirm that this parameter indeed influences ROBDD compilation, again to the better or the worse. We conclude that, while we identified certain parameters influencing the success of optimizations in synthetic data, in the case of less regular data, the answer to **Q2** remains an open issue for further investigation.

4.8 Nested Tries

In Chapter 3 we presented the notion of nested tries and their generation through tabling. We also presented how the nested tries are converted to ROBDD definitions and discussed several optimizations (subset check, ancestor list refine and pre-process); we refer to these optimizations as the nested trie optimizations. The complete algorithm for processing nested tries with the nested trie optimizations as presented in Chapter 3 is shown in Algorithm 4.4. The main complication for this algorithm as mentioned in Chapter 3 is related with the cycle handling. Furthermore, it is important to maintain an efficient preprocessing of the nested tries. In order to achieve that, we have presented the nested trie optimizations that improve different performance aspects. Here we present the generalized algorithm that can use any of the preprocessing method presented in this chapter to generate the ROBDD definitions from nested tries.

Algorithm 4.4 is presented in three parts. The first part, which is the function

NESTED_TRIE() is responsible for initializing the data structures needed by the algorithm. The most important part of the algorithm is included in function NESTED_TRIE_RECURSIVE() and finally the third part shown in Algorithm 4.5 is the pre-process optimization presented in Section 3.4.5.

Algorithm 4.4 can be used in conjunction with any of the different preprocessing algorithms presented in this chapter. In Algorithm 4.4, the chosen preprocessing method is called by the function call PREPROCESSING_METHOD(). The preprocessing methods in order to be integrated in Algorithm 4.4 need to function as follows:

1. The preprocessing algorithm needs to create a copy of the trie that is about to be processed. This is important, as the same trie can be processed several times with different ancestors sets. Different ancestor sets could produce different ROBDD definitions.

2. The preprocessing algorithm must reduce the trie at each iteration by removing any processed part of the Boolean formula, similarly as the recursive node merging method does. This is important in order to ensure termination.

3. Whenever the preprocessing algorithm detects a nested trie, it stops its process and returns control to the nested trie Algorithm 4.4 which is responsible to replace the nested trie with a ROBDD definition reference (for example: $L10$, $false$, etc.). After, the preprocessing algorithm is called again to continue the generation of ROBDD definitions from the point where it had stopped. Note, that only one copy of the trie is at all times kept; this copy is reduced while it is processed. In that way the algorithm knows exactly where its processing had stopped. A second occurrence of the same trie is resolved as a cycle.

Algorithm 4.4 starts with calling one of the preprocessing methods as shown at line 1. Whenever a descendant nested trie (T_{nested} in Algorithm 4.4) is found, the preprocessing method is stopped for Algorithm 4.4 to handle the nesting. First, Algorithm 4.4 at line 6 checks if the descendant nested trie introduces a cycle in order to prune the branch by replacing T_{nested} with $false$ as shown in line 7, otherwise it checks at line 8 if the descendant nested trie is already memoized with the appropriate ancestor list ($A_{T_{nested}}$ or superset of that list). If so, the algorithm re-uses the memoized result as shown in lines 9-11. Finally, if the nested trie is neither a cycle nor memoized, Algorithm 4.4 recursively executes the generation processing for the descendant nested trie in order to get a suitable ROBDD definition reference to replace T_{nested} which is shown in lines 13-16.

The memoization for this algorithm is performed by using as key the trie identifier. We can memoize a result whenever it is fully processed, which means that no more descendant nested tries have been left to be processed. Then for each trie identifier we use a bucket which contains the different possible results. Those results are scanned linearly to find one that is suitable for re-usage depending on the ancestor list.

Algorithm 4.5 formalizes the pre-process optimization presented in Section 3.4.5. It operates over each nested trie by assuming that all references to other nested trie are pruned. In that way it generates and memoizes base cases which can be re-used by Algorithm 4.4. As it is shown the algorithm uses the preprocessing methods in a similar way as Algorithm 4.4 uses them. Note that the memoized ancestor list is the descendant list of the nested trie, as this ancestor list would generate these ROBDD definitions.

4.9 Negation

4.9.1 Negated Literals

Handling negated literals for ProbLog is relatively simple. A negated literal is produced by negating a single probabilistic fact. This means that the negated literal succeeds when the probabilistic fact does not exist in the possible world, thus the probability for the negated literal is $P(not(pf)) = 1 - P(pf)$. One can simply interpret the negation of a single probabilistic fact as negating the literal in the Boolean formula.

4.9.2 General Negation

With the introduction of nested tries, one can handle general negation for ProbLog. As it is argued in [36] negation as defined in the well-founded semantics [20] is more suitable for ProbLog programs. Using nested tries one can implement negation as in the well-founded semantics for ProbLog.

When negating a goal in ProbLog the expected result is the probability that the goal fails instead of succeeds, thus $P(not(G)) = 1 - P_{success}(G)$. On the Boolean formula level we negate the DNF part of the goal at hand. As a nested trie actually corresponds to the proofs required for a specific goal, negating the nested trie results in negating the specific part of the DNF and thus achieves our goal.

Algorithm 4.4 The generalized nested trie to ROBDD definitions approach that uses all presented preprocessing methods and optimizations.

input A root trie T of the nested tries to generate ROBDD definitions and an empty depth breadth trie DBT to store the generated ROBDD definitions.

output Updates DBT to contain the generated ROBDD definitions for T and returns the reference L_{end} that represents the complete Boolean formula.

```
1: function NESTED_TRIE(T, DBT)
2:   Tries := COLLECT_TRIES_IN(T)
3:   L_begin := PRE-PROCESS(Tries, DBT, 1)
4:   (L_end, _) := NESTED_TRIE_RECURSIVE(T, DBT, ∅, ∅, L_begin)
5:   return L_end
```

input A trie T of the nested tries to generate ROBDD definitions, the depth breadth trie DBT to store the generated ROBDD definitions, the ancestor tries A_T and any previously known descendant tries D_T of the trie T.

output Updates depth breadth trie DBT to contain the ROBDD definitions generated for T. Returns for trie T the representative reference L_{end} and all its descendant tries as D_T.

```
1: function NESTED_TRIE_RECURSIVE(T, DBT, A_T, D_T, L_begin)
2:   (L_end, T_nested) := PREPROCESSING_METHOD(T, DBT, L_begin)
3:   if T_nested ≠ null then {T contains a nested trie T_nested}
4:     A_{T_nested} := A_T ∪ {T}
5:     D_T := D_T ∪ {T_nested}
6:     if T_nested ∈ A_{T_nested} then {T_nested introduces a cycle}
7:       Replace the occurrence of T_nested in trie T with false.
8:     else if IS_MEMOIZED(T_nested, A_{T_nested}) then
9:       (L_{T_nested}, D_{T_nested}) := GET_MEMOIZED_RESULT(T_nested, A_{T_nested})
10:      Replace the occurrence of T_nested in trie T with L_{T_nested}.
11:      D_T := D_T ∪ D_{T_nested}
12:    else {T_nested is not a cycle, neither is memoized.}
13:      (L_{T_nested}, D_{T_nested}) := NESTED_TRIE_RECURSIVE(T_nested, DBT,
           A_{T_nested}, ∅, L_end + 1)
14:      Replace the occurrence of T_nested in trie T with L_{T_nested}.
15:      D_T := D_T ∪ D_{T_nested}
16:      L_end := L_{T_nested}
17:    return NESTED_TRIE_RECURSIVE(T, DBT, A_T, D_T, L_end + 1)
18:  else {T is fully processed}
19:    A_T := A_T ∩ D_T {Ancestors Refine}
20:    MEMOIZE(T, A_T, D_T, L_end)
21:    return (L_end, D_T)
```

Algorithm 4.5 The algorithm that performs the pre-process optimization in all Tries.

input A trie collection $Tries$ to be pre-processed, the depth breadth trie DBT to store the generated ROBDD definitions and the starting ROBDD definition reference L_{begin}.

output Updates DBT to contain ROBDD definitions for the pre-processed tries and returns the reference $L_{end}+1$ that represents the next free ROBDD definition reference.

```
function PRE-PROCESS(Tries, DBT, L_begin)
  L_end := L_begin
  for all T ∈ Tries do
    D_T := ∅
    (L_end, T_nested) := PREPROCESSING_METHOD(T, DBT, L_end)
    while T_nested ≠ null do
      Replace the occurrence of T_nested in trie T with false.
      D_T := D_T ∪ {T_nested}
      (L_end, T_nested) := PREPROCESSING_METHOD(T, DBT, L_end)
    memoize(T, D_T, D_T, L_end)
  return L_end + 1
```

Complications arise when considering cycles. We have shown that a cycle does not contribute to the probabilities mass and can be dropped. At that point we where considering cycles in which both goals (the one proving and the one introducing the cycle) are not negated. It is easy to show that the same reasoning applies when both goals are negated. But when only one of the two goals that form a cycle is negated, special handling is required.

The problem in this specific situation is that a paradox is defined which states: that a goal is proven if the negation of the goal is proven ($p \leftarrow \neg p$). These type of theories lie in the undecided part of the well-founded semantics which has no practical use for ProbLog. To tackle the problem we assume that all undecided atoms do not provide any probabilistic information and we can remove them. In Chapter 7, we present an application of general negation where we need to perform model verification and inconsistent clauses such as the previous one are detected and calculated to provide a normalization factor. Specifically for that application we used Stickel's [81] approach for proving inconsistencies. This results to a logic very similar to that of Nilson [52].

While, in general, nested tries are only used for tabled goals, we implemented ProbLog negation for goals to generate their own nested tries regardless whether the goal is tabled or not. A consequence of general negation and this approach

is, that the produced Boolean formulae are no more in DNF form. Fortunately, our approaches for computing the probability are not limited to DNF formulae and the issue is easily resolved.

4.10 Conclusions and Future Work

We introduced depth breadth tries as a new data structure to improve preprocessing in ProbLog, and compared the resulting method and its variations to the method used so far as well as to the decomposition method presented by [66]. Our experiments with the three-state Markov model and with exact inference confirm that our trie based method outperforms the other methods on problems with a high amount of suffix sharing between proofs. At the same time they reveal that the decomposition method is more suited if this is not the case, and thus is a valuable new contribution to ProbLog. Finally, we presented how our preprocessing methods generalize to handle nested tries and negation. Though that our approaches are inspired by ProbLog they can be used by other applications that use Boolean formulae regardless their form.

While the three depth breadth optimizations, aimed at reducing the number of ROBDD operations, can greatly improve performance in some cases, in others, they have opposite effects. Initial experiments suggest that those optimizations should be applied less greedily. Future research could be in the line of a more in depth study of the factors influencing the effects of these optimizations. Also further investigation of the respective strengths of our trie based approach and the decomposition method, and to exploit those in a hybrid preprocessing method could be an interesting research topic.

Chapter 5

Preprocessing II: Variable Compression

ProbLog uses ROBDDs to compute the success probability of a query. While the complexity of the calculation of the probability is linear in terms of the size of the ROBDD, the generation of the ROBDD is an NP-hard task.

It is well-known that the variable ordering used to construct the ROBDD for a Boolean formula has an impact on the size of the ROBDD. The orderings that give rise to smaller ROBDDs are called good orderings: constructing smaller ROBDDs takes less time and space and also the computation of the probability is faster. State-of-the-art ROBDD tools use heuristics to decide about the variable ordering, which search space is exponential.

In this chapter we present an approach that reduces the search space for the variable ordering by decreasing the number of variables in the Boolean formula, namely by replacing subsets of variables by new representative variables. We call this **variable compression**. In the context of ProbLog we can perform variable compression if the calculated probability of the new ROBDD remains the same.

In order to do variable compression before the ROBDD generation, we need to detect these variable subsets in the Boolean formulae, or in the case of ProbLog at the level of the DNF. We discovered sets of variables in the DNF for which we can compute the contribution of such a set of variables to the probability of the ROBDD independently from the rest of the ROBBD and replace these sets by new representative variables. We name these sets AND/OR-clusters.

The AND/OR-clusters are determined for a particular DNF and as such they are query-dependent. Remember that a DNF contains all the proofs of a query. Figure 5.1 presents graphically the ProbLog parts that are modified by variable compression.

In this chapter, we first give the cluster definitions and present how the clusters can be used to compress variables in Section 5.2. Next, we show how to discover AND/OR-clusters in DNF Boolean formulae in Section 5.3 and Section 5.4 respectively. Finally, we present some experiments for the AND-clusters in Section 5.5 and a complexity analysis of the algorithms in Section 5.6.

This chapter is based on publication [44]; novel is the discussion of OR-clusters and the complexity analysis of all algorithms.

5.1 Example

For this chapter we use the example ProbLog program of Figure 5.2. This program represents the graph of Figure 5.3(a). The edges in the graph are labelled with the probabilities and with the Boolean variables (x_i) that will be used in the ROBDD. We use the query `path(1, 3)` which collects the proofs shown as a trie in Figure 5.3(b). The collected trie represents the DNF Boolean formula: $(x_0 \wedge x_2) \vee (x_0 \wedge x_3 \wedge x_6) \vee (x_1 \wedge x_4 \wedge x_5 \wedge x_2) \vee (x_1 \wedge x_4 \wedge x_5 \wedge x_3 \wedge x_6)$. Figure 5.4 presents three possible ROBDDs, (a) with a bad ordering for the variables, (b) with a good ordering for the variables and (c) with the ordering by ProbLog. The success probability of the query is $P = 0.498296$.

Comparing the three ROBDDs of Figure 5.4 it is clear that we want to avoid generating ROBDDs with a bad ordering. The heuristics used by the ROBDD package do not take into consideration special properties (such as annotated disjunctions, chains of probabilistic facts, graph structure, etc.) that arise from specific domains like the probabilistic one.

5.2 Cluster Definitions

We define two kinds of clusters and prove that their compression does not affect the final probability. We define the clusters in terms of the ROBDDs, as the idea of looking for patterns can also be valuable for other application areas that use ROBDDs.

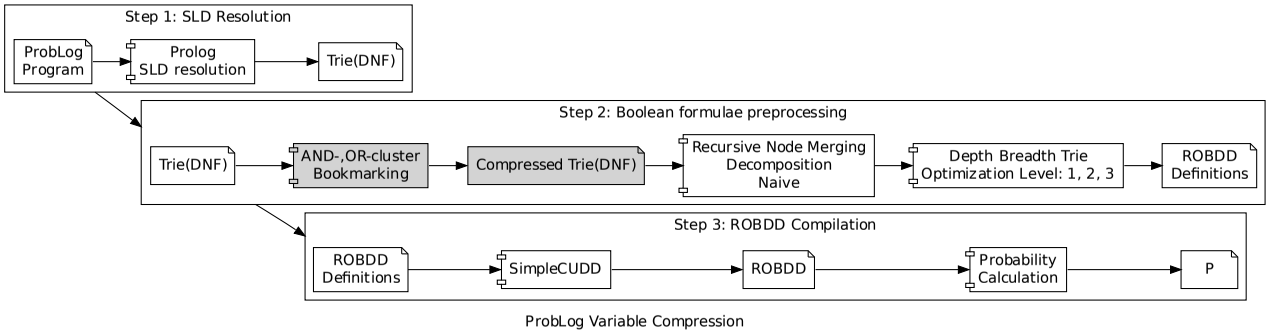


Figure 5.1: The diagram of the three steps of ProbLog with the variable compression additions. Shaded blocks mark the modifications from the block figure of Figure 2.2.

Probabilistic Facts:

```
0.5::edge(1, 2).
0.4::edge(1, 4).
0.7::edge(2, 3).
0.8::edge(2, 6).
0.9::edge(4, 5).
0.7::edge(5, 2).
0.4::edge(6, 3).
```

Background Knowledge:

```
path(X, Y):-
  path(X, Y, [X]).
path(X, Y, _):-
  edge(X, Y).
path(X, Y, V):-
  edge(X, Z),
  Y \== Z,
  \+ member(Z, V),
  path(Z, Y, [Z|V]).
```

Figure 5.2: ProbLog example program. The probabilistic facts `edge/2` and for background knowledge the predicate `path/2`.

Definition 11. *Let F be a Boolean formula with variables $v_1, \ldots, v_l$. The variables $\{x_1, \ldots, x_k\} \subseteq \{v_1, \ldots, v_l\}, k > 1$, form an **AND-cluster** if there exists a variable ordering such that the ROBDD R of F*

1. *has only one node n_i for variable $x_i, 1 \leq i \leq k$,*
2. *node n_j has as its only incoming edge the high edge of node $n_{j-1}, 2 \leq j \leq k$,*
3. *and the low edges of the nodes $\{n_1, \ldots, n_k\}$ connect to the same node in R.*

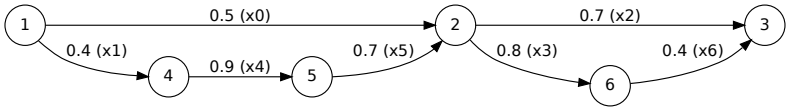
An example of AND-clusters can been seen in Figure 5.5(a).

Definition 12. *Let F be a Boolean formula with variables $v_1, \ldots, v_l$. The variables $\{x_1, \ldots, x_k\} \subseteq \{v_1, \ldots, v_l\}, k > 1$, form an **OR-cluster** if there exists a variable ordering such that the ROBDD R of F*

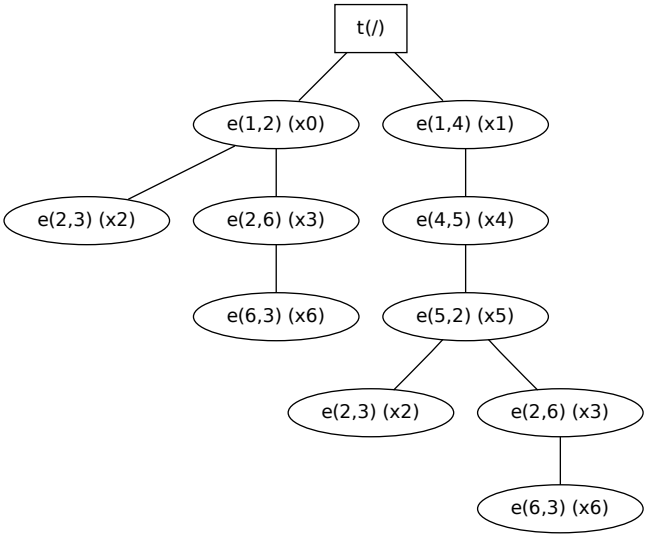
1. *has only one node n_i for variable $x_i, 1 \leq i \leq k$,*
2. *node n_j has as its only incoming edge the low edge of node $n_{j-1}, 2 \leq j \leq k$,*
3. *and the high edges of the nodes $\{n_1, \ldots, n_k\}$ connect to the same node in R.*

An example of OR-clusters can been seen in Figure 5.5(b).

In a probabilistic framework like ProbLog that uses ROBDDs to calculate probabilities, each ROBDD variable has an assigned probability. To be able to compress the clusters of variables to new representative variables, we need to

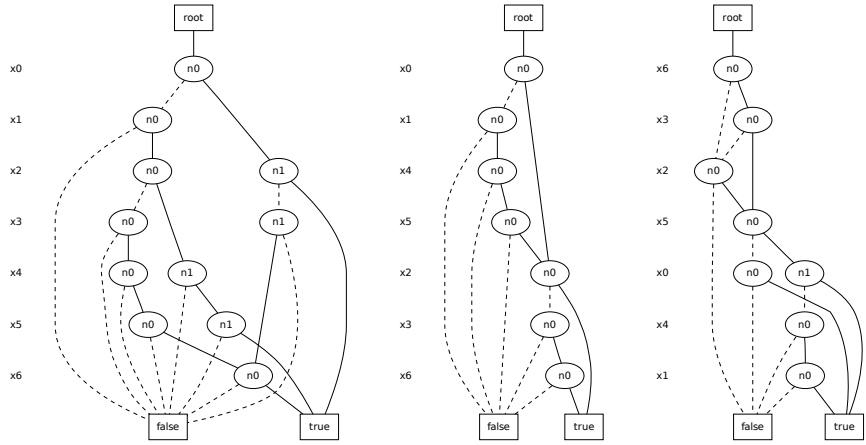


(a) The probabilistic graph of the example ProbLog program.



(b) The collected trie for the query: `path(1,3)`.

Figure 5.3: The probabilistic graph of the example ProbLog program shown in Figure 5.2 and collected trie for the query: `path(1,3)`.



(a) Bad ordering: (x0-x1-x2-x3-x4-x5-x6)

(b) Good ordering: (x0-x1-x4-x5-x2-x3-x6)

(c) ProbLog ordering: (x6-x3-x2-x5-x0-x4-x1)

Figure 5.4: ROBDD for the query: `path(1, 3)` of the example ProbLog program shown in Figure 5.2. ROBDD notation is explained at Section 2.3.4

compute the probabilities of the representative variables such that the probability we compute for the ROBDD as a whole does not change.

Definition 13 (Replacing a cluster by a representative variable)**.** *Replacing a cluster of variables* $\{x_1, \ldots, x_n\}$ *by a representative variable* V *is the equivalent of removing all nodes of the variable in ROBDD and replacing them by one. All ingoing edges to the cluster would be ingoing edges to the representative variable. Similarly, the representative variable node outgoing edges will go to the same nodes where the cluster's outgoing edges are going.*

Theorem 5 (Probability of AND-cluster)**.** *Replacing an AND-cluster* $\{x_1, \ldots, x_n\}$ *by a representative variable* V *with probability* $P_V = P_{AND}(\{x_1, \ldots, x_n\}) = \prod_{i=1}^{n} P(x_i)$ *does not change the probability of the ROBBD as a whole.*

Proof. First consider the simple case of a ROBDD that consists of exactly one AND-cluster, $\{x_1, \ldots, x_n\}$. The probability of this ROBDD is $P(x_1) \cdot P(x_2) \cdot \ldots \cdot P(x_i) \cdot \ldots \cdot P(x_n) \cdot 1 + (1 - P(x_1) \cdot P(x_2) \cdot \ldots \cdot P(x_i) \cdot \ldots \cdot P(x_n)) \cdot 0 = \prod_{i=1}^{n} P(x_i)$. But in general, an AND-cluster has an outgoing high edge to a part T with P_T and its low edges connect to a part F with P_F. The probability of the ROBDD part that includes the AND-cluster can be generalised as $P = P(x_1) \cdot P(x_2) \cdot \ldots \cdot P(x_i) \cdot$

$... \cdot P(x_n) \cdot P_T + (1 - P(x_1) \cdot P(x_2) \cdot ... \cdot P(x_i) \cdot ... \cdot P(x_n)) \cdot P_F = P_T \cdot \prod_{i=1}^{n} P(x_i) +$ $P_F - P_F \cdot \prod_{i=1}^{n} P(x_i) = (P_T - P_F) \cdot \prod_{i=1}^{n} P(x_i) + P_F$. If we replace the AND-cluster with a new representative variable V with P_V and calculate the probability, we get $P = P_V \cdot P_T + (1 - P_V) \cdot P_F = P_V \cdot P_T + P_F - P_V \cdot P_F = (P_T - P_F) \cdot P_V + P_F$. Therefore $P_V = P_{AND}(\{x_1, \ldots, x_n\}) = \prod_{i=1}^{n} P(x_i)$. □

Theorem 6 (Probability of an OR-cluster)**.** *Replacing an OR-cluster* $\{x_1, \ldots, x_n\}$ *by the representative variable* V *with probability* $P_V = P_{OR}(\{x_1, \ldots, x_n\}) = P(x_1) + (1 - P(x_1)) \cdot P_{OR}(\{x_2, \ldots, x_n\})$ *and* $P_{OR}(\{x_n\}) = P(x_n)$ *does not change the probability of the ROBBD as a whole.*

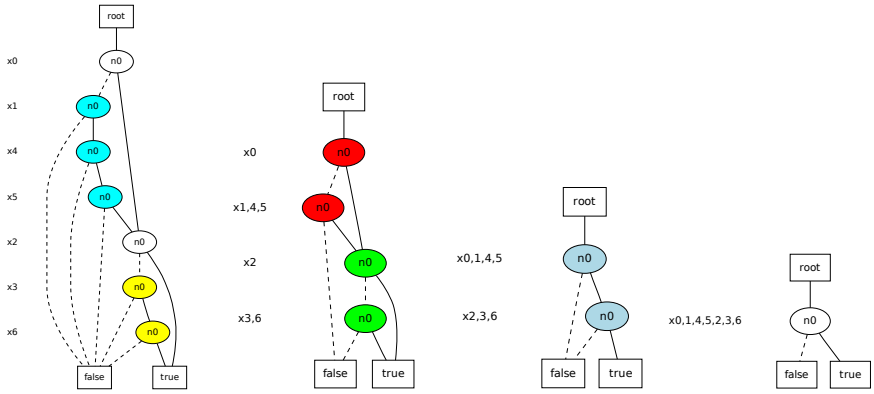
Proof. First consider the simple case of a ROBBD that consists of exactly one OR-cluster, $\{x_1, \ldots, x_n\}$. The probability of this ROBBD is $P(x_1) \cdot 1 + (1 - P(x_1)) \cdot (P(x_2) \cdot 1 + (1 - P(x_2)) \cdot \ldots \cdot (P(x_i) \cdot 1 + (1 - P(x_i)) \cdot \ldots \cdot (P(x_n) \cdot 1 + (1 - P(x_n)) \cdot 0)) \ldots) = P(x_1) + (1 - P(x_1)) \cdot P_{OR}(\{x_2, \ldots, x_n\})$. But in general an OR-cluster has its high edges to a part T with P_T and an outgoing low edge to a part F with P_F. The probability can be generalised as $P = P(x_1) \cdot P_T + (1 - P(x_1)) \cdot (P(x_2) \cdot P_T + (1 - P(x_2)) \cdot \ldots \cdot (P(x_i) \cdot P_T + (1 - P(x_i)) \cdot \ldots \cdot (P(x_n) \cdot P_T + (1 - P(x_n)) \cdot P_F)) \ldots) = (P(x_1) + (1 - P(x_1)) \cdot P(x_2) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{i-1})) \cdot P(x_i) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{n-1})) \cdot P(x_n)) \cdot P_T + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_n)) \cdot (P_F / P_T)$. If we replace the OR-cluster with a new representative variable V with P_V and calculate the probability, we get $P = P_V \cdot P_T + (1 - P_V) \cdot P_F$ if we replace $P_V = P(x_1) + (1 - P(x_1)) \cdot P(x_2) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{i-1})) \cdot P(x_i) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{n-1})) \cdot P(x_n)$ then we need to prove that $1 - P_V = (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_n)) \Rightarrow 1 - (P(x_1) + (1 - P(x_1)) \cdot P(x_2) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{i-1})) \cdot P(x_i) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{n-1})) \cdot P(x_n)) = (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_n))$. Finally by using the distribution rule we see that the previous formula is a tautology. Therefore $P_V = P_{OR}(\{x_1, \ldots, x_n\}) = P(x_1) + (1 - P(x_1)) \cdot P_{OR}(\{x_2, \ldots, x_n\})$. □

5.2.1 Using the Clusters for Variable Compression

We illustrate the variable compression with our `path(1, 3)` example. In Figure 5.5(a) we have two AND-clusters, $\{x_1; x_4; x_5\}$ and $\{x_3; x_6\}$. After compression we obtain the ROBBD in Figure 5.5(b) with two new Boolean variables $x_{1,4,5}$, $x_{3,6}$ and their associated probabilities $P(x_{1,4,5})$[1], $P(x_{3,6})$[2]. After

[1] $P(x_{1,4,5}) = P_{AND}(\{x_1; x_4; x_5\}) = 0.4 \cdot 0.9 \cdot 0.7 = 0.252$

[2] $P(x_{3,6}) = P_{AND}(\{x_3; x_6\}) = 0.8 \cdot 0.4 = 0.32$



(a) AND-clusters. (b) OR-clusters. (c) AND-cluster. (d) Compressed.

Figure 5.5: Compressing ROBDD for the query: `path(1,3)`. Notation: coloured nodes represent clusters.

AND-compression we have two OR-clusters, $\{x_0; x_{1,4,5}\}$ and $\{x_2; x_{3,6}\}$ as shown in Figure 5.5(b); by further compressing them we get the ROBDD in Figure 5.5(c) with two new Boolean variables $x_{0,1,4,5}$, $x_{2,3,6}$ and their probabilities $P(x_{0,1,4,5})$[3], $P(x_{2,3,6})$[4]. Finally, by compressing the single AND-cluster $\{x_{0,1,4,5}; x_{2,3,6}\}$ of the ROBDD in Figure 5.5(c) we end up with the ROBDD in Figure 5.5(d) that has a single Boolean variable $x_{0,1,4,5,2,3,6}$ and probability $P(x_{0,1,4,5,2,3,6})$[5].

Not all ROBDDs can be compressed to a single variable, but iterating AND/OR-cluster based variable compression can lead to an easier to construct ROBDD. We want to use variable compression to be able to deal with queries that caused timeouts. So, we are willing to pay a certain cost to detect the clusters. In order to use our variable compression in practice, we first have to detect AND-clusters in DNFs.

5.3 Discovering AND-clusters

To be able to benefit from AND-cluster compression, we need to identify them before the ROBDD generation. Fortunately, AND-clusters also appear in the DNF representing the proofs: either all the probabilistic facts of an AND-cluster

[3] $P(x_{0,1,4,5}) = P_{OR}(\{x_0; x_{1,4,5}\}) = 0.5 + (1 - 0.5) \cdot 0.252 = 0.626$
[4] $P(x_{2,3,6}) = P_{OR}(\{x_2; x_{3,6}\}) = 0.7 + (1 - 0.7) \cdot 0.32 = 0.796$
[5] $P(x_{0,1,4,5,2,3,6}) = P_{AND}(\{x_{0,1,4,5}; x_{2,3,6}\}) = 0.626 \cdot 0.796 = 0.498296$

appear in a proof, or none of them. A naive approach to detect AND-clusters is to find longest common subsequences (LCS) in the conjunctions of the DNF, however this is an NP-hard problem [39]. As our problem is a special case of the LCS we can do better.

Theorem 7. *Every set of Boolean variables* $\{v_1, \ldots, v_n\}$ *in a set of clauses* $\{cl_1, \ldots, cl_m\}$ *satisfying:* $\forall v_i \in \{v_1, \ldots, v_n\}$ $occur(v_i) = (\bigcap_{v_i \in cl_j} cl_j) \cap (\bigcap_{v_i \notin cl_j} \overline{cl_j}) = \{v_1, \ldots, v_n\}$*, forms an AND-cluster; where* $\overline{cl_j}$ *denotes the complement of the set* cl_j *with respect to the set of the Boolean variables in all clauses.*

Proof. Indeed, if in a DNF Boolean formula we take all clauses $cl_1, \ldots, cl_k \in \{cl_1, \ldots, cl_m\}$ where v_i appears and in those and only those all $\{v_1, \ldots, v_n\}$ appear, then by the distribution property we can create a single clause of the form $(v_1 \wedge \cdots \wedge v_n) \wedge (cl'_1 \vee \cdots \vee cl'_k)$ with $cl'_i \wedge (v_1 \wedge \cdots \wedge v_n) = cl_i$. □

In ProbLog setting we call the Boolean variables as probabilistic facts, and the clauses as proofs. Thus, the first part of $occur(pf_i)$ is the set of probabilistic facts that occur in each proof in which pf_i occurs. The second part is the set of probabilistic facts that do not occur in proofs that do not contain pf_i. The first set is a possible AND-cluster for pf_i but it might also contain probabilistic facts that occur in proofs that do not contain pf_i. In order to exclude the latter ones, the possible AND-cluster has to be restricted to probabilistic facts that only occur in proofs containing pf_i.

5.3.1 The Book Marking Algorithm for AND-clusters

Based on Theorem 7, the Book Marking Algorithm 5.1 deals with all the proofs one by one and ensures that for all probabilistic facts pf_i seen by the algorithm so far $occur(pf_i)$ is computed. The algorithm encodes a proof by a bit string. We order the probabilistic facts by their chronological appearance in the proofs. The i^{th} probabilistic fact is denoted by $pf(i)$. The i^{th} bit encodes whether the probabilistic fact $pf(i)$ is used in the proof. We refer to the bit string as the **occurrence number** (ON) of the proof.

We use a two dimensional matrix (MA) of bits to represent the AND-clusters. Row k corresponds to the probabilistic fact $pf(k)$ and represents $occur(pf(k))$. Column l represents the probabilistic fact $pf(l)$. The element l of a row k indicates whether $pf(l)$ forms an AND-cluster with $pf(k)$. This matrix grows incrementally as we deal with the proofs one by one and the size of each dimension is equal to the number of different, already seen probabilistic facts.

Dealing with a new proof involves computing ON and then computing its impact on the AND-clusters already in MA according to Theorem 7 using the following three operations:

1. for each previously seen probabilistic fact i which appears in this proof, we compute $MA[i] = MA[i] \wedge ON$;
2. for each previously seen probabilistic fact i that does not occur in this proof, we compute $MA[i] = MA[i] \wedge \neg ON$; and
3. we grow MA to include AND-clusters for the probabilistic facts that were not seen before.

After all proofs of the DNF have been dealt with, all the rows in MA with more than one active bit (i.e. set to 1) represent an AND-cluster.

Algorithm 5.1 The Book Marking algorithm for AND-clusters.

input A DNF Boolean formulae (`DNF`) and the count of different variables in the Boolean formulae (`Count`).

output A two dimensional bit matrix represented as a matrix of integers containing all AND-clusters (`Matrix`).

```
function BOOKMARKING(DNF, Count)
  MatrixSize := 0
  CREATE(Matrix, Count)
  for all Proof ∈ DNF do
    OccurrenceNumber := BIT_ENCODE(Proof)
    for (i := 0; i < MatrixSize; i++) do
      if (2^i & OccurrenceNumber) > 0 then {Existing matrix row and pf_i
      in OccurrenceNumber - operation 1}
        Matrix[i] := Matrix[i] & OccurrenceNumber
      else {Existing matrix row and pf_i not in OccurrenceNumber -
      operation 2}
        Matrix[i] := Matrix[i] & not(OccurrenceNumber)
    for (i := MatrixSize; i < BIT_LENGTH(OccurrenceNumber); i++)
    do {Add a new matrix row - operation 3}
      Matrix[i] := not(2^MatrixSize − 1) & OccurrenceNumber
    MatrixSize := MAX(BIT_LENGTH(OccurrenceNumber), MatrixSize)
  return Matrix
```

After all proofs of the DNF have been processed, the resulting matrix can be processed in a straightforward way: iterate over all matrix rows starting the count from 0, and every row i that has a value greater than $2^{(i+1)}$ is

representing an AND-cluster. Because the row of each probabilistic fact that is a member of an AND-cluster fully represents that AND-cluster, we have multiple appearances of the same AND-cluster. As the matrix is symmetric with respect to its diagonal we can check if any bit before the i^{th} is active and that informs us about duplicates.

5.3.2 An Example of the Book Marking Algorithm for AND-clusters

As an example for the Book Marking algorithm we use the proofs of `path(1,3)`: [x0, x2], [x0, x3, x6], [x1, x4, x5, x2], [x1, x4, x5, x3, x6]. Each row of Table 5.1 corresponds to a single proof (PR), and has the probabilistic fact order (OL), the occurrence number (ON) and the matrix (MA).

For the first proof, the algorithm uses the order $x0 < x2$ to compute 11 as the occurrence number of the proof. As initially the matrix MA is empty, operation 3 uses the ON to construct a MA with two rows and two columns with all bits activated.

For the second proof the algorithm adds $x3$ and $x6$ to the order which becomes $x0 < x2 < x3 < x6$. The algorithm computes $ON = 1101$; note that we are reading the bit strings from right to left. Operation 1 computes the conjunction of 1101 with the row of $x0$: $11 \wedge 1101 = 0011 \wedge 1101 = 0001$. This operation sets the bit corresponding to $x2$ to 0 as $x0$ and $x2$ are no longer an AND-cluster. For the row of $x2$, operation 2 computes $11 \wedge neg(1101) = 0011 \wedge 0010 = 0010$ and sets the bit for the probabilistic fact $x0$ to 0. Finally, the algorithm extends MA by two new rows and columns for the probabilistic facts $x3$ and $x6$ with, as values of the rows, $neg(11) \wedge 1101 = 1100 \wedge 1101 = 1100$. Note that also the existing rows are expanded with new columns set to 0.

When all proofs are dealt with, the Book Marking Algorithm has found two different AND-clusters, namely $\{x1; x4; x5\}$ and $\{x3; x6\}$. Without variable compression, ProbLog generates the ROBDD of Figure 5.4(c), which has a size in between the sizes of the other two ROBDDs in Figure 5.4. After compressing the variables of the AND-clusters to a representative variable $x1, 4, 5$ with $P(x1, 4, 5) = 0.252$ and $x3, 6$ with $P(x3, 6) = 0.32$, we get the compressed proofs: [x0, x2], [x0, x3,6], [x1,4,5, x2], [x1,4,5, x3,6]. For the compressed proofs, ProbLog generates the ROBBD of Figure 5.5(b).

The algorithm as presented here only tackles proofs that contains either positive or negative occurrences of each probabilistic fact and not both. If in one proof a probabilistic fact is positive and in an other is negative, this probabilistic fact is not part of an AND-cluster.

Proof (PR)	=	x0, x2							1	1	x2			
Order List (OL)	=	[x0, x2]							1	1	x0			
Occurrence Number (ON)	=	11 = 3							x2	x0				
Matrix (MA)	=	[3, 3]												
PR	=	x0, x3, x6							1	1	0	0	x6	
OL	=	[x0, x2, x3, x6]							1	1	0	0	x3	
ON	=	1101 = 13							0	0	1	0	x2	
MA	=	$[3 \wedge 13, 3 \wedge neg(13), neg(3) \wedge 13, neg(3) \wedge 13]$							0	0	0	1	x0	
MA	=	[1, 2, 12, 12]							x6	x3	x2	x0		
PR	=	x1, x4, x5, x2	1	1	1	0	0	0	0	x5				
			1	1	1	0	0	0	0	x4				
OL	=	[x0, x2, x3, x6, x1, x4, x5]	1	1	1	0	0	0	0	x1				
ON	=	1110010 = 114	0	0	0	1	1	0	0	x6				
MA	=	$[1 \wedge neg(114), 2 \wedge 114, 12 \wedge neg(114),$	0	0	0	1	1	0	0	x3				
		$12 \wedge neg(114), neg(15) \wedge 114,$	0	0	0	0	0	1	0	x2				
		$neg(15) \wedge 114, neg(15) \wedge 114]$	0	0	0	0	0	0	1	x0				
MA	=	[1, 2, 12, 12, 112, 112, 112]	x5	x4	x1	x6	x3	x2	x0					
			1	1	1	0	0	0	0	x5				
PR	=	x1, x4, x5, x3, x6	1	1	1	0	0	0	0	x4				
OL	=	[x0, x2, x3, x6, x1, x4, x5]	1	1	1	0	0	0	0	x1				
ON	=	1111100 = 124	0	0	0	1	1	0	0	x6				
MA	=	$[1 \wedge neg(124), 2 \wedge neg(124), 12 \wedge 124,$	0	0	0	1	1	0	0	x3				
		$12 \wedge 124, 112 \wedge 124, 112 \wedge 124, 112 \wedge 124]$	0	0	0	0	0	1	0	x2				
MA	=	[1, 2, 12, 12, 112, 112, 112]	0	0	0	0	0	0	1	x0				
			x5	x4	x1	x6	x3	x2	x0					

Table 5.1: An example of the Book Marking algorithm for AND-clusters shown in Algorithm 5.1.

5.4 Discovering OR-clusters

Similarly to AND-clusters, in order to benefit from OR-cluster compression, we need to identify them before the ROBDD generation. Fortunately, also OR-clusters appear in the DNF representation of the proofs. By using the distributive property so that each probabilistic fact of an OR-cluster appears only in a single clause, then the formed clauses are identical (except the probabilistic fact belonging to the OR-cluster). Again, our detection problem is similar to the LCS problem. Fortunately, again it is a special case which we can solve in polynomial time.

Theorem 8. *Every set of Boolean variables* $\{v_1, ..., v_n\}$ *in a set of clauses* $\{cl_1, ..., cl_m\}$ *satisfying* $\forall v_i \in \{v_1, ..., v_n\}$*, (1)* $v_i \in cl_i$ *and (2)* $v_i \notin cl_j$ *with* $i \neq j$*,* $1 \leq i, j \leq n$ *and (3)* $(cl_i \setminus \{v_i\}) = (cl_j \setminus \{v_j\})$ *forms an OR-cluster.*

Proof. Indeed, if in a DNF Boolean formula we take all the clauses $cl_1, ..., cl_k \in \{cl_1, ..., cl_m\}$ where $v_1, ..., v_n$ appear, by the distribution property we can create a single clause of the form $(v_1 \vee \cdots \vee v_n) \wedge cl'$ where cl' is the common part of $cl_1, ..., cl_k$. □

In ProbLog setting we call the Boolean variables as probabilistic facts, and the clauses as proofs. Thus, using the above observation we separately apply the distribution property for each probabilistic fact, collecting all "common" clauses $cl_1, \ldots, cl_k$. Finally, we compare all the clauses among them to identify the variables that have identical clauses. Those variables that have identical clauses participate in the same OR-cluster.

5.4.1 The Book Marking Algorithm for OR-clusters

Based on the previous observation and on Theorem 8, the Book Marking Algorithm 5.2 first collects all clauses for each variable by using the distributive property. As we apply the distribution property in the whole DNF formula each clause might contain both disjunctions and conjunctions. For that reason, we encode the conjunctions as a bit string generating like before an $OccurrenceNumber$ and then each encoded conjunction is being stored in an ordered set. This way the order of variables and duplicate conjunctions will not affect later the comparison to determine whether two clauses are identical.

After all variables have been processed then we need to compare all the generated clauses among them. For this final step we compare all steps and we modify a Matrix which will finally contain all OR-clusters. Notice in Algorithm 5.2

that by using the bitwise OR operator we mark all the identical clauses in the Matrix.

Finally, the resulting matrix can be processed in a straight forward way: iterate over all matrix rows starting the count from 0, and every row i that has a value greater than $2^{(i+1)}$ is representing an OR-cluster. Because the row of each probabilistic fact that is a member of an OR-cluster fully represents that AND-cluster, we have multiple appearances of the same OR-cluster. As the matrix is symmetric with respect to its diagonal we can check if any bit before the i^{th} is active, and that informs us about duplicates.

Algorithm 5.2 The Book Marking algorithm for OR-clusters.

input A DNF Boolean formulae (`DNF`) and the count of different variables in the Boolean formulae (`Count`).

output A two dimensional bit matrix represented as a matrix of integers containing all OR-clusters (`Matrix`).

```
function BOOKMARKING(DNF, Count)
  CREATE(Matrix, Count)
  CREATE(Clause, Count)
  for (i := 0; i < Count; i++) do
    Matrix[i] := 2^i
  for (Variable := 0; Variable < Count; Variable++) do
    Clause[Variable] := DISTRIBUTION_PROPERTY(Variable, DNF)
  for (i := 0; i < Count; i++) do {Clause comparison process}
    for (j := i + 1; j < Count; j++) do
      if (Clause[i] = Clause[j]) then
        Matrix[i] := Matrix[j] := Matrix[i]|Matrix[j] {Where | denotes
        the bitwise OR operator}
  return Matrix
function DISTRIBUTION_PROPERTY(Variable, DNF)
  CREATESET(Clause)
  for all (Proof ∈ DNF) do
    OccurrenceNumber := BIT_ENCODE(Proof)
    if (2^Variable & OccurrenceNumber > 0) then
      OccurrenceNumber := OccurrenceNumber − 2^Variable
      Clause := ADDTOSET(OccurrenceNumber, Clause)
  return Clause
```

5.4.2 An Example of the Book Marking Algorithm for OR-clusters

To finish our example for the Book Marking Algorithm, we now continue from the compressed proofs that we stopped earlier: [x0, x2], [x0, x3,6], [x1,4,5, x2], [x1,4,5, x3,6]. The Table 5.2 is broken into three parts: first is the encoding of the proofs in integer numbers, where we also present the order list (OL) used for encoding; the second part is performing the distributive property for each variable over the DNF; finally we present the clause comparison process that actually discovers the OR-clusters in the DNF formulae.

The second step of our example shown at Table 5.2 presents the final result of the encoding for the clause. Each conjunctive clause is encoded to a bit string. In our example there are only single literal conjunctive clauses but assume that we had the conjunction $1 \wedge 2$, this gets encoded by activating bits 1,2 (110) from a zero indexed bit string and would result in the number 6. We perform this encoding as it is easy to insert, sort the resulted integers and keep efficiently at all times sorted sets which later we can efficiently compare.

For the final step, we present the initial Matrix for each iteration of the comparison and the concluding Matrix for the same iteration. We only present the two iterations where the clauses actually are identical and the matrix is being manipulated. The final return Matrix for our example is $MA = [9, 6, 6, 9]$.

Finally, we can see that the algorithm detected two OR-clusters, namely: {x0; x1,4,5}, {x2; x3,6} and will compress the two OR-clusters into two new probabilistic facts x0,1,4,5 and x2,3,6 respectively. The new probabilities for these facts are $P(x0, 1, 4, 5) = 0.626$ and $P(x2, 3, 6) = 0.796$. After this compression ProbLog generates the ROBDD shown in Figure 5.5(c).

5.5 Experiments for AND-clusters

We implemented the variable compression method using only AND-clusters within ProbLog. To judge the practicality and the impact, we use ProbLog benchmarks that discover links in real biological networks [77]. Graphs model probabilistic links between concepts such as genes, proteins, etc. The first benchmark consists of a graph of concepts related to the Alzheimer disease that has $23,060$ edges; because of the size, inference for this graph soon becomes intractable. We query for the existence of a path between two given nodes, to control the problem size we limit the maximum path length. For the second benchmark, we take the experiments (the same data sets and the same

First Step: Numeric Encoding of Proofs

x0, x2		0, 1
x0, x3,6	$\Longrightarrow$	0, 2
x1,4,5, x2		3, 1
x1,4,5, x3,6		3, 2

Order List (OL) = [x0, x2, x3,6, x1,4,5]

Second Step: Perform Distributive Property for each Variable

Variable	Boolean Clause	Set Encoded Clause
0	$0 \wedge (1 \vee 2)$	Clause[0]=[2,4]=[0010, 0100]
1	$1 \wedge (0 \vee 3)$	Clause[1]=[1,8]=[0001, 1000]
2	$2 \wedge (0 \vee 3)$	Clause[2]=[1,8]=[0001, 1000]
3	$3 \wedge (1 \vee 2)$	Clause[3]=[2,4]=[0010, 0100]

Third Step: Clause Comparison Process

Starting Matrix $\Longrightarrow$ Concluding Matrix

1	0	0	0	0		1	0	0	1	0
0	1	0	0	1		0	1	0	0	1
0	0	1	0	2	$\xrightarrow[Clause[i]=Clause[j]]{i=0;j=3}$	0	0	1	0	2
0	0	0	1	3		1	0	0	1	3
0	1	2	3			0	1	2	3	

Starting Matrix $\Longrightarrow$ Concluding Matrix

1	0	0	1	0		1	0	0	1	0
0	1	0	0	1		0	1	1	0	1
0	0	1	0	2	$\xrightarrow[Clause[i]=Clause[j]]{i=1;j=2}$	0	1	1	0	2
1	0	0	1	3		1	0	0	1	3
0	1	2	3			0	1	2	3	

Table 5.2: An example of the Book Marking algorithm for OR-clusters shown in Algorithm 5.2.

queries) from [16]. All the graphs are fragments of the same network [77]. The experiments should give answers to the following questions:

Q1: What is the compression ratio in a real life data set?

Q2: How does compression improve the performance of generating an ROBDD?

Q3: In which cases is the variable compression beneficial?

The default setting of ProbLog is to use CUDD's [80] group sifting [53] dynamic reordering during ROBDD compilation. CUDD uses the following memory-time trade-off. It starts by consuming memory without reordering the variables; once the memory usage passes a threshold, it starts reordering the variables and as a

consequence it consumes time. While CUDD is implemented in C, our Book Marking algorithm is implemented in Yap Prolog [71].

When we increase the problem size for the first benchmark, we see that the ROBDD compilation time is the limiting factor. We executed three different queries with a timeout of 1 hour for the ROBDD compilation. Each query was executed 5 times and we present the averaged times for the ROBDD compilation. Table 5.3 presents the comparison of executing the queries with four different settings. The results are presented grouped by the three queries (a), (b) and (c). The first and second column use the dynamic reordering strategy; the third and the fourth use the order in which probabilistic facts appear in the proofs as a static ordering; the first and third column use variable compression of AND-clusters. Our experiments confirm that for big ProbLog problems dynamic reordering performs better than static ordering. Variable compression improves the ROBDD compilation times and has the expected effect both for dynamic and static orderings.

The second part of Table 5.3 presents the compression statistics which are independent of the reordering method: the time to do variable compression, the number of AND-clusters found, the number of variables before compression (ovars), the number of variables after compression (cvars), and finally the variable compression ratio[6]. We note that the time cost for doing the variable compression is much less than the time gained during ROBDD compilation. More importantly, the time for finding the AND-clusters is polynomial (as shown in the next section), while that for ROBDD compilation is exponential. Because our constant costs are relatively high, we notice that in small problems variable compression needs more time than we gain during ROBDD compilation, but those problems are solved very fast either way. The benefit of variable compressing is far more significant for larger problem sizes.

In order to confirm the positive results for the compression ratio and the better performance of the ROBDD compilation, we use the larger set of experiments of our second benchmark. We study the impact of variable compression in combination with dynamic reordering as it was confirmed to be the better option for ProbLog. In this benchmark, all the queries can be computed without variable compression. The behaviour of queries is diverse, as some spent most of the time in the ROBDD compilation and others in SLD resolution. Among the 360 queries of [16], 100 queries do not use any probabilistic facts. We divided the other 260 queries into 3 groups: the first group contains 92 queries that generate tiny ROBDDs with less than 20 variables; the second group contains 152 queries that generate small ROBDDs with 20 or more variables, but less than 100; and finally the third group contains the queries that generated relatively big

[6]Ratio = (ovars - cvars) / ovars

		ROBDD compilation Times				Compression Statistics				
	Path Length	Reordering Compressed	Reordering Only	Static Compressed	Static	Time	Clusters Discovered	Original Var.	Compr. Var.	Compr. Ratio
	8	4	4	5	5	7	11	34	23	32%
	9	51	97	7	9	22	17	91	71	22%
(a)	10	153	297	10	12	32	25	137	110	20%
	11	24,830	90,529	*	*	336	76	337	254	25%
	12	3,083,750	-	-	-	835	92	479	378	21%
	8	5	4	4	5	5	7	26	17	35%
(b)	9	282	417	24,904	47,000	72	49	170	119	30%
	10	1,035	1970	*	*	91	53	226	169	25%
	11	1,019,588	-	-	-	966	104	528	410	22%
	4	4	4	4	4	0	3	13	10	23%
(c)	5	95	246	18	23	64	42	135	91	33%
	6	224	497	74	122	33	45	180	131	27%
	7	58,917	2,488,793	*	*	385	92	455	350	23%

The reported times are in milliseconds. Longer path lengths timeout.
A **-** indicates a timeout.
A ***** indicates that the system runs out of memory.

Table 5.3: AND-cluster experimental results for the first set of benchmarks.

ROBDDs with more than 100 variables.

For the 'Tiny' group we obtain an average compression ratio of 42%. Their ROBDD compilation times and the variable compression times are too small to draw any conclusions. For the other two groups, we compute averages for each group and for both groups together. The results are in Table 5.4. We give the variable compression ratio, the time gain realised for the ROBDD compilation, and the variable compression time relative to the SLD resolution time.

Query Group	ROBDD Compr. Ratio	ROBDD Com. Time Gain	Compression Time Ratio
Small	$(28 \pm 11)\%$	$(40 \pm 36)\%$	$(26 \pm 41)\%$
Big	$(27 \pm 5)\%$	$(47 \pm 23)\%$	$(69 \pm 107)\%$
All	$(28 \pm 10)\%$	$(41 \pm 36)\%$	$(32 \pm 53)\%$

The reported times are in milliseconds. All presented results are average and with their standard deviation.
Compr. Ratio = (original variables - variables after compression) / original variables.
ROBDD Com. Time Gain = (ROBDD time without compression - ROBDD time with compression) / ROBDD time without compression.
Compression Time Ratio = (SLD time with book marking algorithm - SLD time without) / SLD time without.

Table 5.4: AND-cluster experimental results for the second set of benchmarks.

While the results might be specific for the application, they confirm the actual presence of AND-clusters in real world datasets. In this real dataset we encounter a compression ratio that ranges from 7% to 61% with an average of 28%. The compression ratio results are similar to the ones of the first benchmark.

In the 'Small query' group 44% of the queries have a small number of variables and they do not need variable reordering neither before nor after compression: their time gain is near 0. Most of the other queries in the 'Small query' group need reordering before and no reordering after compression, so they have a huge time gain up to 87%. On average we end up with a gain of 40%.

For the 'Big query' group the average gain is larger, namely 47%, but the variation is less as all the queries need reordering before and after compression. Here the gain comes from having less variables that have to be dealt with during the reordering by the state-of-the-art tool.

Comparing the variable compression time with the SLD resolution time shows that the former is smaller than the latter, but its cost is relatively higher for the 'Big' queries.

Our experiments[7] yield promising results, answering our initial questions by showing that there is in real life ProbLog applications a role for variable compression, as it improves significantly the performance of the ROBDD compilation.

5.6 Complexity Analysis

The Book Marking algorithm for AND-clusters shown in Algorithm 5.1 has a worst case complexity of $O(M \cdot N^2)$ where M is the number of proofs seen and N is the number of different probabilistic facts; usually for ProbLog applications $M >> N$. For all M proofs, we do a bitwise encoding, and then we modify the matrix $Matrix$.

By using an indexed table, the encoding of a proof is done in $O(N)$ time and requires $O(N)$ space to remember the index for each probabilistic fact encountered. Each proof needs to modify $Matrix$ which is an NxN bit table. Activating or deactivating a bit in the table is done in constant time, but in the worst case all bits needs to be processed resulting in $O(N^2)$ operations. So, the total time complexity is $O(M \cdot (N + N^2)) = O(M \cdot N^2)$ and the total space complexity $O(N + N^2) = O(N^2)$.

One can take advantage of the symmetry and other properties of the NxN bit matrix $Matrix$ to avoid some computations. These optimisations reduce the constant times rather than the complexity. One such optimisation is that we use arbitrary precision integers to represent each row of $Matrix$.

The worst case complexity of the Book Marking algorithm for OR-clusters shown in Algorithm 5.2 is similar $O(M \cdot N^2 + N^2) = O(M \cdot N^2)$. In more detail, the first part of the algorithm requires N steps to collect, for each probabilistic fact, the clauses from the M proofs in the DNF formulae in which the probabilistic fact appears. The maximum size for each clause is N. Thus, in the worst case scenario, N steps are required to collect MxN sized proofs. The second part of the algorithm which compares all collected clauses takes N^2 steps. Unfortunately, the total space complexity of this algorithm is worse than the previous one. Collecting the clauses can require memory up to O($M \cdot N^2$); while the second step requires an NxN matrix resulting to a final $O(M \cdot N^2 + N^2) = O(M \cdot N^2)$ memory complexity.

[7]For our experiments we used an Intel^R Core^{TM}2 Duo CPU at 3.00GHz with 2GB of RAM memory running Ubuntu 8.04.2 Linux.

5.7 Related Work and Conclusions

We exploit regularities, AND-clusters and OR-clusters, observed in ROBDDs to improve the generation of ROBDDs for DNFs in ProbLog. Variable compression based on these clusters reduces the number of variables in the DNFs. This results in smaller ROBDDs, whose generation uses less time and memory, and as such we can deal with ProbLog queries that used to cause timeouts. Our method is a preprocessing step that detects clusters of Boolean variables. Taking into account the probabilistic setting, variable compression is feasible and can be followed by any other variable ordering heuristic. For other applications, one might be able to find different meaningful compressions, or one might just use our clusters as input to existing variable ordering heuristics.

Variable ordering heuristics also exploit structural properties of the problem modelled by the ROBDD such as connected variables [2,53]. Heuristics designed for one application area might perform poorly in another context [50]. We are not aware of variable ordering heuristics being used in a probabilistic context.

Hintsanen [28] argues that structural properties are important for finding the most reliable subgraph. He calculates the probability of subgraphs connecting two nodes and searching for the subgraph with the maximum probability. The paper identifies as a special case the series-parallel subgraphs for which they can compute the probability polynomially. These series-parallel subgraphs have similarities with our AND/OR-clusters.

We have presented a polynomial algorithm for detecting the AND-clusters and we have obtained promising results for an application using a real database. For ProbLog the best results are obtained by combining AND-cluster variable compression with the group sifting dynamic variable ordering of CUDD. By using variable compression we managed to answer more queries. We showed that AND-cluster based variable compression is beneficial for more complex ROBDD.

For a future implementation of the Book Marking algorithm, C would be a better choice than Prolog both for time efficiency as for space. This would reduce the hidden constant factors of Prolog and would also save Prolog garbage collector executions. It is worth noting that the first iteration to discover AND-clusters could be computed in parallel with the SLD resolution.

A further task would be to implement the presented OR-cluster discovery algorithm and verify experimentally its benefits. In addition to the technical improvements, a challenging task is to investigate how we can take advantage of the AND/OR-clusters in other Boolean forms like CNF. Unfortunately, discovering AND/OR-clusters in nested tries imposes fundamental difficulties

as both bookmarking algorithms presented here are based on properties of the DNF Boolean form. For example, collecting the clauses required for the OR-clusters would be the equivalent of unfolding the nested trie structure to a DNF. Finally, the goal would be to generalise the method and to be able to compress repeated structures in the ROBDD such as XOR-like clusters. The size of the ROBDDs is one of the limits that is currently reached when executing ProbLog programs. We think that an approach based on variable compression can push this limit.

Chapter 6

A New Implementation: MetaProbLog

In this chapter we present technical details of the ProbLog2011 implementation. The motivation for this implementation is the inability to extend the ProbLog2010 implementation with support for features such as: meta-programming, higher-order probabilistic calls, modularity issues, memory management. Also, it is more elegant in ProbLog2011 to perform general negation for non-tabled goals and to use the built-in tabling mechanism of a Prolog system as shown in Section 3.5.2.

In order to achieve these new features we introduce the abstract ProbLog engine concept. Our solution allows the engine to suspend the resolution of a query q_c with its relevant probabilistic information, to start the resolution of a new query q_n and to compute its desired result, and then use the result of q_n when resuming the inference of q_c. Our approach is based on ProbLog engines. A ProbLog engine has a set of parameters and a state. Every different instantiation of these parameters implements a different ProbLog inference method. By suspending the execution of the current ProbLog engine and creating a new one, we are able to support nested inference. To suspend and resume ProbLog engines we use a stack. The contributions of this work are: (1) a general solution for nesting ProbLog queries; (2) the implementation of new primitives; (3) a ProbLog that abides to the Prolog module system; (4) better memory management.

In Prolog, we typically write a program and then we formulate goals in terms of the program predicates. The program predicates can call new Prolog goals, effectively nesting them. A Prolog system uses SLD resolution to compute the

failure or success of the goals and in case of success the answer substitutions. During SLD resolution goals are nested in a very natural way: in order to prove a particular goal the conjunction of nested goals must be proven. Results of goals are always used in the same way: failure causes backtracking and on success we use the answer substitution. Meta-programming makes it possible to construct some of the nested goals at run-time, but once the Prolog system has mapped such a callable term to a goal, execution continues as if the query was known at compilation time. This feature of Prolog allows the programmer to easily form programs that include higher-order functionality.

For Problog, in order to compute the probability of a query, the inference methods need to do some bookkeeping about the probabilistic facts used for proving the query. This bookkeeping interferes with meta-programming and as a consequence does not directly allow higher-order functionality. Current probabilistic logic programming languages such as PHA [61], PRISM [73, 74], PITA [68, 69], do not support nested inference[1]. We are presenting an approach for nested inference. Nested inference in a probabilistic logic programming is required for implementing higher-order probabilistic calls. Nested inference in ProbLog interferes with the necessary bookkeeping performed. Moreover, ProbLog queries can return different kinds of results. For a ground query, ProbLog can compute its success probability. For a non-ground query, ProbLog can compute the different answers for the query together with their respective probabilities.

In the new approach, instead of computing the probability, ProbLog can also return detailed information about the annotated facts used during the proofs of the query, for example the corresponding DNF. Therefore, we present a more general ProbLog predicate that has the form `problog_inference(Inference, Query, ResultType, Result)`. The variable *Inference* is used to specify the used inference method; *Query* the user's query to be proven; *ResultType* the type of return result we are interested in, for example the probability, or the corresponding DNF(nested tries); and finally *Result* is the returned result.

In most Prolog systems programming in the large is achieved through the use of modules. The ProbLog2010 implementation was not taking into account modularity. One was unable to write a program that would be encapsulated by a module: when ProbLog2010 would load it would make assumptions that all ProbLog programs where situated in the *user namespace*. Furthermore, it would assert information about probabilistic facts, and internal structures in the *user namespace*. The behaviour of ProbLog2010 implementation would prevent other Prolog based extensions to cooperate harmonically. ProbLog2011

[1] We use the term "nested inference" when we refer to a higher-order call that contains probabilistic information.

implementation fully conforms to the module system of Yap Prolog, allowing users to make ProbLog programs that lie in different namespaces than the *user namespace*. The ProbLog2011 implementation is fully encapsulated, allowing the user to safely import it as a separate library without interfering with other Prolog extensions.

Prolog systems base their memory management on the use of an automated garbage collector. ProbLog2010 creates several structures where the automated garbage collector of Prolog is not aware of. That results in used data that remain after the termination of a query. In order to tackle these memory leaks, ProbLog2011 completely alters the way it manages memory compared with ProbLog2010. The new approach keeps track of used data structures over each query and is able to fully remove unnecessary data structures on the termination of queries.

The main focus of this chapter, is to introduce the ProbLog engine abstraction used to achieve nested inference and explain how technically it can be used to implement several different inference methods. We also explain in detail the new memory management. Then we explain the new higher-order primitives that we have implemented and illustrate their functionality with some examples. We finally describe the modularity improvements we made.

Section 6.1 presents the limitation of the existing system and motivates the need for probabilistic meta-calls. We briefly present the running example in Section 6.2. Then we propose a way to overcome these limitations by the usage of ProbLog engines in Section 6.3. Then follows some new primitives and some examples of their usage in Section 6.4. Section 6.5 presents the ProbLog2011 implementation modularity improvements. A few experiments are presented in Section 6.6 and finally, Section 6.7 concludes.

This chapter is been based on [45].

6.1 Why Probabilistic Meta-calls

Many real life applications use probabilistic inference to make decisions about a task. For a probabilistic logic system, to fully support decision making, the nesting of its inference methods is required. Consider for example the problem of inferring the semantic similarity between two words. While there are many approaches to tackle this problem, there is no best one. A reasonable approach is to infer the word similarity in different independent ways and then use a combination model. One could represent the synonym relation between words as a probabilistic graph and write a ProbLog program to infer the probability

of two words having the same meaning. This technique does not perform well if spelling errors appear in the words. One could write another ProbLog program to find the probability of a spelling error. There are several ways to use these results in a probabilistic model. The final model that uses probabilistic inference during probabilistic inference, can be looked at as a higher order model.

The existing ProbLog implementation does not support nested inference. Moreover, once we start using nested inference we also want to determine at run-time the actual ProbLog query. We call the proposed extensions probabilistic meta-calls.

In Prolog, goals are nested all the time as the basic step of SLD resolution proves a goal by proving the goals in the body of a unifying clause. Moreover, goals can be constructed as Prolog terms at run-time and then Prolog's support for meta-calls transforms the terms into executable goals.

As we have presented, the ProbLog2010 system can compute a probabilistic query of the form `problog_inference(Inference, Query, Result)` for a given *Query*, with a chosen $Inference$ method and returns the success probability at *Result*. Note that during this inference no nested calls to `problog_inference/3` are allowed as they interfere with the bookkeeping of the use of probabilistic facts.

In this chapter we generalise the `problog_inference/3` predicate, to allow nested inference and to support meta-call features. In addition of determining at run-time the inference method and the query, we also want to specify what kind of result we want: the success probability of the query, or a specific representation of the bookkeeping information on which the probability computation is based. The generalised ProbLog query will call `problog_inference( Inference, Query, ResultType, Result)` and these calls can be nested. We introduce the notion of a ProbLog engine that allow us to implement the general `problog_inference/4` meta-predicate.

6.2 Example

To illustrate the features of this chapter we use the example ProbLog program in Figure 6.1. The program encodes a probabilistic graph which is a typical example in ProbLog. In this type of programs the usual queries are for the probability that a path exists between two nodes in the graph; or a query that contains an unbound variable for a node asks what the probability that a path exists which starts or ends at the node that was given as an input; finally, one can ask what is the probability that a path exists in the graph by leaving both

Probabilistic Facts:

```
0.3::edge(1, 2).
0.7::edge(1, 3).
0.4::edge(2, 3).
0.8::edge(3, 4).
0.6::edge(3, 5).
0.2::edge(4, 5).
```

Background Knowledge:

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

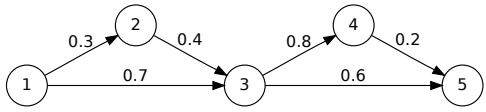


Figure 6.1: An example of a probabilistic graph and the corresponding ProbLog program.

arguments unbound. In the ProbLog2010 implementation it is not possible to query about the answers of a query, thus with which binding the query succeeded.

6.3 Technical Details

Before describing the ProbLog engine, we want to point out that for the exact inference of Section 2.3, ProbLog collects the probabilistic facts used in a success branch of the SLD tree of the query in a list (a so-called explanation or proof), and also collects all explanations as a DNF, which is typically represented by a trie. This trie is then transformed into a ROBDD in order to compute the correct success probability. On the other hand, program sampling as in Section 2.4.1 samples a possible world which is kept in a list[2] and counts the successful derivations.

Some shortcoming of the previous system are: the lack of intermix different inference methods; each inference method needs its own SLD-resolution kernel; the difficulty to use alternative data structures and to extend or modify current functionality. We identified the need for an abstract framework that provides a common SLD-resolution kernel, that can be instantiated to realise different inference methods and/or different design options.

[2]Or an equivalent data structure like an array.

6.3.1 ProbLog Engine

The ProbLog engine is an abstraction that allows dynamic modifications of Prolog's resolution to uniformly implement the different bookkeeping methods needed by the different inference methods. By parametrisation of the ProbLog engine, one parametrises the resolution.

The basic functionality of the parametrised ProbLog engine is the resolution of the query together with the bookkeeping about the probabilistic facts that are used during this execution. By setting the parameters of the ProbLog engine, different instances are created that correspond to different inference methods and to different result types.

The difference with the ProbLog2010 implementation is the parametrised design, but also the organisation of the data-structures of the ProbLog engine. Each instance of the ProbLog engine has its own unique identifier, which is used when working with instance specific data. This instance-based organisation is necessary for the nested inference.

In the rest of this section we explain the relevant parameters of the ProbLog engine and how the nesting is supported.

6.3.2 Parameters of the ProbLog Engine

In order to define an instance of a ProbLog engine that implements an inference method, we specify two "continuation" predicates that deal with the construction of the explanations and the construction of the DNF. We also decide about the kind of data structures that are used to represent the explanations and the DNF. The instance of the ProbLog engine uses two "registers" to refer to the two data structures.

More specifically, we use two "continuation" predicates that are used during the SLD resolution to implement the adequate bookkeeping for the probabilistic facts, which are called annotated facts in this context: a fact annotated with a probability is a probabilistic fact[3]. Both predicates are used to perform inference specific tasks and are different from inference method to inference method.

The first dynamic "continuation" predicate is `continuation_af/2` (af stands for annotated fact) which is called every time the proven goal is an annotated fact. The first argument is the unique identifier of the annotated fact and

[3]We use annotated facts instead of probabilistic facts, for the low level implementation, in order to support different type of annotations such as the ones used in Decision Theory ProbLog [86] and Algebraic ProbLog [38] extensions.

the second argument the annotation of the fact, typically its probability. The second dynamic "continuation" predicate is `continuation_explanation/0` and is called every time the SLD resolution reaches a successful derivation. In addition to the two predicates, each instance of a ProbLog engine has two "registers". These registers contain a reference to the instance specific data structures in which the information about the usage of annotated facts is collected. The first register (actually the referred data structure) is used by `continuation_af/2` and the second by `continuation_explanation/0`. These registers are part of the state of the ProbLog engine that has to be saved and reset for nested inference.

6.3.3 Inference Method: Exact

In Figure 6.2(a) we present the parametrisations that implements the exact inference method. The `continuation_af/2` predicate is responsible for collecting the identifiers of the used annotated facts in a list, i.e., the ID of the used annotated fact is added to the current explanation (referred to by the first register). The `continuation_explanation/0` predicate is responsible for collecting the explanations in a trie: it adds the current completed explanation to the trie (referred to by the second register). The first register refers to the current (partial) explanation, and the second to the trie under construction.

6.3.4 Inference Method: Program Sampling

In Figure 6.2(b) we present the parametrisations that implement the program sampling inference method which we briefly explained in Section 2.4.1. For program sampling, the `continuation_af/2` predicate is responsible for checking whether the annotated fact or its negation is in the current sampled possible world, and if not to sample it and add it or it's negation. The `continuation_explanation/0` predicate does not need to do anything. In this example we represent a partial possible world by a list but it could be represented by an array or another data structure. The first register refers to the current partial possible world. The equivalent bookkeeping for the second register would be the complete possible world, but because it is not required to sample the complete possible world we only need and keep a unique identifier which refers to the sample.

```
Continuation_af = (continuation_af(ID, _Probability):-
                     add_to_explanation(ID)),
Continuation_explanation = (continuation_explanation:-
                              add_to_trie(completed)),
problog_engine_init(exact,
                      continuations(Continuation_af,
                                    Continuation_explanation),
                      state(list, trie)).
```

(a) ProbLog engine parametrisation for exact.

```
Continuation_af = [(continuation_af(ID, _Probability):-
                       in_possible_world(ID, Result),
                       !, call(Result)),
                    (continuation_af(ID, Probability):-
                       sample(Probability, Result),
                       add_possible_world(ID, Result),
                       call(Result))
                  ],
Continuation_explanation = (continuation_explanation),
problog_engine_init(program_sampling,
                      continuations(Continuation_af,
                                    Continuation_explanation),
                      state(list, identity)).
```

(b) ProbLog engine parametrisation for program sampling.

Figure 6.2: Modifying the SLD resolution for exact, program sampling inference methods.

6.3.5 ProbLog Memory

Together with the ProbLog engine strategy we also redesigned the way ProbLog handles memory. There where several drawbacks in the ProbLog2010 implementation. Most characteristic was that plenty of execution related data would not be properly destroyed after the faulty termination of a query. Similarly, even a correct termination of a query could not ensure that all related data had been removed from memory. In order to address these issues the ProbLog2011 implementation creates a memory space for each ProbLog engine. Together at the destruction of the ProbLog engine the ProbLog garbage collector is responsible to remove all used data from memory.

To structure ProbLog's memory more efficiently we distinguished the memory

into two different types, static memory and dynamic memory. ProbLog's static memory contains data that are usually constructed by the compilation of a ProbLog program and are not affected by the execution of a query. For example, the annotations of facts are all stored in ProbLog's static memory. The focus of this memory is to provide fast access to the data. The second and more interesting part is ProbLog's dynamic memory.

All data that are stored to the dynamic memory are marked to belong to a specific ProbLog engine. This is easily achieved by adding to the stored data the ProbLog engine identifier. Obviously, data for the dynamic memory can only be produced when a ProbLog engine is active. Furthermore, any memory access of data must belong to the same ProbLog engine that stored it. In this way, data of a ProbLog engine are protected from another ProbLog engine. This is a crucial feature for meta calls and for properly using the built-in tabling mechanism.

The most obvious benefit of this approach is, that it is easy to reclaim used memory from a ProbLog engine after it is destroyed. In that way we can ensure, both for the proper or faulty termination of a query, that no traces of the execution are left in memory. We need to mention that the built-in tabling mechanism does not permit to partially abolish tabled results. To overcome this limitation and to ensure that all tables are cleared, the ProbLog system schedules the abolishment of tabled results after the termination of all nested engines. After the termination of a query, a ProbLog finalization step is called which fully "cleans" the memory. This finalization step optionally triggers the host Prolog garbage collector in order to ensure that non referred Prolog terms are removed too.

Prolog uses an automated memory management through the use of a garbage collector. The implementation of ProbLog uses several structures that Prolog's garbage collector cannot automatically remove. For example, it asserts facts, creates tries, arrays or generates records. Furthermore, the unpredictable nature of a garbage collector generates latency and can cause several problems. ProbLog's finalization step, instead of being triggered periodically, is activated at the termination of each engine. By also triggering Prolog's garbage collector, we reduce the influence of terminating a query on the performance of the next query.

The presented strategy showed significant improvement especially in the ability of the system to call several queries. Optionally, the garbage collector can be configured to keep information which future queries can reuse in order to save execution time. This implemented but experimental feature potentially improves the execution of a query by reusing previous results, similarly to tabling, instead of recalculating the query. Also the feature allows the reuse of

tabled calls from compatible engines extending the tabling benefits over meta calls. Programs that call several times similar queries or sub-parts of a query, (e.g., for learning) can have huge benefits from an approach like this.

6.3.6 Nesting ProbLog Engines

The nesting of ProbLog engines, requires a suspension/resumption mechanism for which we use a stack. The active ProbLog engine is called the current engine.

When a new engine is initialised, it first pushes the current engine on the top of the stack and then becomes the active engine. When an engine has finished all its computations, it ends by popping the stack of engines.

When pushing an engine on the stack, we save the engine parameters: `continuation_af/2`, `continuation_explanation/0`, the two registers, and the unique identifier of the engine. This information is sufficient to implement the nesting of engines using a stack discipline. Note that the information kept in the stack is related to the probabilistic bookkeeping.

6.3.7 Calling the ProbLog Engine

Probabilistic inference in ProbLog is invoked through `problog_inference/3`. Given the inference method and the query one retrieves the probability that the query succeeds. When the given query is non-ground, `problog_inference/3` computes as success probability the success probability of all the instances of the query without binding the variables. Later at Section 6.4.3 we present a new inference method that returns the answers through backtracking by binding the non-ground variables.

An instance of ProbLog engine is used to execute the `problog_inference/2`, `problog_inference/3`, and `problog_inference/4` predicates. The four arguments of the predicate `problog_inference/4` are $Inference$, *Goal*, *ResultType*, *Result*. The argument $Inference$ is used to define which ProbLog inference method is used and its possible values are *pure*, *exact*, *program_sampling*, and *current*, where *pure* denotes that the engine behaves as pure Prolog and *current* instructs the engine to use the same inference method as was being used. The argument *Goal* denotes the goal that needs to be proved by the call. *ResultType* denotes that we are interested in the success probability of the goal or in the explicit representation of the collected information and its possible values are *probability* and $info$. Finally, the

argument *Result* is the returned probability or the detailed information that the engine returns.

The following predicates `problog_inference/3` and `problog_inference/2` are used as syntactic sugar.

```
problog_inference(Inference, Goal, Probability):-
  problog_inference(Inference, Goal, probability, Probability).
problog_inference(Goal, Probability):-
  problog_inference(current, Goal, probability, Probability).
```

6.4 Nested Inference

With our ProbLog engine based approach we can realise several interesting extensions as is described in the following subsections. Nested inference allows us to compute the success probability of a new child query that was possibly defined at run-time, and use the success probability during the execution of the parent ProbLog query. The nested inference allows us to interleave any combination of different inference methods or inference tasks. Furthermore, returning the explicit information instead of the success probability can be used to formulate the counterpart of the `\+/1` predicate of Prolog. Finally, we use our approach to support non-ground queries.

6.4.1 Nested Inference Returning Success Probability

Our approach allows us to perform nested inference. The nested inference computes the correct results as every call to `problog_inference` suspends the previous ProbLog engine, starts a new independent ProbLog engine and uses the result when the previous one is resumed. In the example of Figure 6.3, ProbLog inference is used to decide which route will be taken.

6.4.2 Nested Inference Returning Information & ProbLog Negation

The implementation of negation as failure in Prolog uses a meta-call as shown in Figure 6.4(a). ProbLog negation [36] is a more complex task due to bookkeeping issues. In the ProbLog2010 implementation, probabilistic facts could be easily negated as their integration is very simple and, as presented in Section 4.9.2, also tabled goals can be negated, but no other goals. Our approach allows to

```
?- Input =..., problog_inference(exact, model(Input), Psucc).

model(Input):-
  some_computation(Input, From),
  decide_route(From).

decide_route(From):-
  problog_inference(path(1, From), P),
  (P < 0.3 ->
    path(From, 5)
  ; P < 0.6 ->
    path(From, 4)
  ;
    path(From, 3)
  ).
```

Figure 6.3: An example of inference within inference using the ProbLog program of Figure 6.1.

negate all probabilistic goals, namely goals that use in their proofs probabilistic facts.

Different inference methods require different implementations of negation. We illustrate the difference using the exact and program sampling inference methods.

For exact inference, proving the negation of a probabilistic fact means to add to the explanation the complementary probabilistic fact[4]. As probabilistic facts are represented by Boolean variables, we simply mark them negated. This negation clearly does not alter the representation of the DNF formula.

The success probability of a probabilistic goal *Goal* is computed from a representation of its corresponding DNF. In order for the negation of a probabilistic goal `problog_not(goal)` to succeed, the negation of the corresponding DNF should hold or the corresponding CNF should hold.

Now consider the contribution of the subgoals to the final DNF: their corresponding annotated facts are scattered over the different explanations in the DNF. If we allow `problog_not` in ProbLog, we need to do something special to incorporate the CNFs in a correct way. Moreover the calls to `problog_not` can be nested. Our solution uses the suspend/resume mechanism to compute the DNF for the negated probabilistic goal. We use nested tries to represent the

[4]Probabilistic facts are represented as random variables, negating them results to the complementary random variable with probability $1 - P$.

```
'\+'(Goal):-
  call(Goal), !, fail.
'\+'(_).
```

(a) Negation as Failure in Prolog.

```
problog_not(exact, Goal):-
  problog_inference(current,Goal,info,DNF),
  continuation_af(not(DNF), _).
problog_not(program_sampling, Goal):-
  \+ Goal.
```

(b) ProbLog Negation for exact and program sampling inference methods.

Figure 6.4: Negation

DNF: by collecting the explanations of the subgoals separately we store them in nested tries (as used for tabling, see [42, 43] and Chapter 3) and in these nested tries we can indicate which parts need to be negated.

In the case of program sampling, things are very different. Instead of proof collection, we count in how many samples the query succeeds. In the process of constructing a possible world, we sample each probabilistic fact that we need to prove. At each sample, a probabilistic fact either succeeds or fails. Negating a probabilistic fact means inverting success with failure and failure with success. Similarly, a probabilistic goal succeeds or fails depending on the possible world sampled, and its negation again means the inversion on success and failure. For that reasons one can use the negation as failure as defined in Prolog for ProbLog programs when doing program sampling.

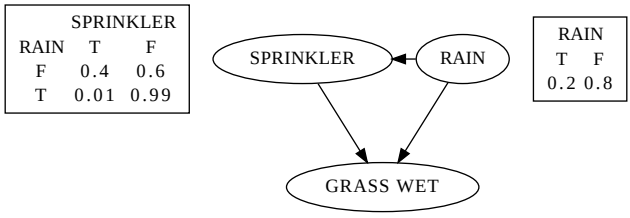
ProbLog negation has many uses in modelling. First of all it can be used to calculate the probability of a query not succeeding. Further on, it has been used to model annotated disjunctions which are needed for many probabilistic models such as a hidden Markov models, Bayesian networks and other. Some example uses are shown in Figure 6.5.

6.4.3 Nested Inference Returning Answers & ProbLog Answers

Finding all the answers of a non-ground query in Prolog is done through backtracking. In ProbLog, we also need to calculate the success probability of the query having a particular answer.

```
% This example encodes a coin toss.
0.50::heads(_Number).
toss(Number, heads) :- heads(Number).
toss(Number, tails) :- problog_not(heads(Number)).

% This example encodes the following Sprinkler/Rain
% Bayesian network from wikipedia.
```



	SPRINKLER	
RAIN	T	F
F	0.4	0.6
T	0.01	0.99

		GRASS WET	
SPRINKLER	RAIN	T	F
F	F	0.0	1.0
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

```
0.20::rain.
0.01::sprinkler_on(rain).
0.40::sprinkler_on(no_rain).
0.80::grass_wet(rain).
0.90::grass_wet(sprinkler).
0.99::grass_wet(both).
sprinkler :- rain, sprinkler_on(rain).
sprinkler :- problog_not(rain), sprinkler_on(no_rain).
grass_wet :- problog_not(sprinkler), rain, grass_wet(rain).
grass_wet :- sprinkler, problog_not(rain),grass_wet(sprinkler).
grass_wet :- sprinkler, rain, grass_wet(both).
```

Figure 6.5: Example uses of ProbLog negation

```
problog_answers(Inference, Goal, P):-
  init_inference(pure_prolog_engine),
  call(Goal),
  problog_inference(Inference, Goal, P),
  (suspend_engine; (resume_engine, fail)).
```

Figure 6.6: Simplified ProbLog Answers

We implemented this task by using Prolog to find the answers of the query. Once we have an answer, we have a grounding of the query and we can do probabilistic inference for this ground query. A simplification of the actual code implementing ProbLog answers is shown in Figure 6.6. With this extension we can return answers to non-ground queries tupled with their success probability. We call this extension *ProbLog answers*.

The `call(Goal)` goal is using Prolog's backtracking mechanism to enumerate all possible answers by fully ignoring any probabilistic information related with the query. When an answer is found, `problog_inference(Inference,Goal,P)` uses the appropriate inference method to calculate the probability of the answer.

This simplified code calls `problog_inference/3` also when a particular answer occurs again. This inefficiency is solved easily by memoizing the calculated answers.

We need to add some functionality to the stack discipline. After dealing with one answer, we need to re-activate the parent engine[5] for continuing the previous goal, but when execution backtracks back to `call(Goal)` we need to again activate the pure Prolog engine that returns us the answers. To solve this we implemented a special suspension mechanism which swaps the order of engines in the stack.

Finally, the last difficulty is that the different engines need separate garbage collectors which must be triggered when an engine is not needed anymore, thus we need a mechanism to tell us when the `call(Goal)` is completed or the user commits to an answer[6]. This problem is solved with the help of YAP's `setup_call_cleanup/3` built-in predicate.

ProbLog answers has many uses. Non-ground queries are needed to find which nodes are connected in probabilistic graphs and with which probability. One can use it in combination with other higher-order calls such as Prolog's `findall/3`, `forall/2`, etc. answering even more complex queries such as which node is more

[5]The outer engine that called ProbLog answers.
[6]When for example the choice points are cut.

```
connected_node(From, To, P):-
  problog_answers(path(From, To), P).

find_most_probable_node(From, MaxNodeTo, MaxP):-
  findall(To-P, problog_answers(path(From, To), P), Tuples),
  findmax(Tuples, MaxNodeTo, MaxP).
```

Figure 6.7: Example uses of ProbLog answers

probable to be connected with a node in a probabilistic graph. See example in Figure 6.7.

6.5 Modularity

It is easy to argue that a module system is an essential feature of a programming language. It facilitates the easy safe reuse of existing code and the development of general purpose libraries. Yap Prolog has a module system [71] based on the Quintus Prolog module system [29] and as ProbLog has been developed as an extension of Yap Prolog it should at least abide by its module system.

The ProbLog2010 implementation violates the rules of the module system, resulting in programmers being unable to write modular ProbLog programs. Furthermore, the ProbLog2010 implementation compiles code without taking into account other modules or other possible Prolog extensions that might be working in together, creating several conflicts. The ProbLog2011 implementation takes in account these issues and addresses the underlying problems of the existing implementation.

In addition of achieving full modularity for the new ProbLog we also suggest and implement the extension of the module system with four new predicates that are useful in the probabilistic domain and especially for machine learning applications. The new predicates namely: `problog_import/2`, `problog_forget/2`, `problog_import_only/2`, `problog_forget_all/1`. They are inspired by the example set system of hipP Prolog [31] which was extensively used in the TILDE/ACE data mining system [6, 7].

The idea behind hipP's example set system was to be able to load/unload different example sets (Prolog facts) to make more clear and modular experiments for data mining. Unfortunately, in Yap Prolog it is not possible to abolish compiled facts. But with the predicates `problog_forget/2` and

`problog_forget_all/1` we restrict the access of compiled probabilistic facts from a module.

Figure 6.8 presents a modular ProbLog program. At Figure 6.8(a) the ProbLog background theory is shown; at Figure 6.8(b), Figure 6.8(c) two different example sets are presented and finally, Figure 6.8(d) illustrates how the modules are used to execute two different experiments which each use a different dataset.

The predicate `problog_import/2` is responsible for importing one or more predicates from a module giving access to the module which it was called from. Predicate `problog_import_only/2`, ensures that the specified predicates will be loaded only from the specified module. The predicate `problog_forget/2` removes one or more imported predicate from a specific module denying access to the calling module and similarly `problog_forget_all/1` removes the specified imported predicates regardless the module that they where imported from. It is important to note that the module system of Prolog has precedence over the four ProbLog predicates. For that reason one must not import any predicates while loading a module otherwise the imports override the functionality of the four presented predicates.

6.6 Experiments

Our experiments aim to measure the meta-call overheads. For the experiments we used a prototype[7] implementation of ProbLog that is implemented using the ProbLog engine approach. All the experiments are performed on an Intel Core 2 Duo CPU at 3.00GHz with 2GB of RAM memory running Ubuntu 8.04.2 Linux under a usual load using Yap 6.2.0 [71].

To measure the overhead we used a typical ProbLog application namely an Alzheimer graph from [37] with a `path/2` predicate that defines paths between nodes; we consider the pair of nodes `'HGNC_582'`, `'HGNC_983'` which is a query that has both many failing and succeeding derivations. We executed three different benchmarks, all of them nested with meta-calls exactly 10 times. The first benchmark is performing all the nesting first and then the goal is proved. The query for the first benchmark is of the form: $exact((exact((exact((...)), G1)), G1))$, where exact/2 is the abbreviation for `problog_inference(exact,Goal,_P)`. This benchmark measures the overhead of the created engines. For these experiments we used a different sample taken from the same graph as the ones used in the experiments for Chapters 3 and 4.

[7] The prototype implementation is available at: `https://lirias.kuleuven.be/bitstream/123456789/316482/3/metaproblog.tar.gz`

```
:- module(path, [path/2]).
path(X, Y):-
  edge(X, Y).
path(X, Y):-
  edge(X, Z),
  Y \== Z,
  path(Z, Y).
```

(a) ProbLog background knowledge.

```
:- module(small_graph, [edge/2]).
0.3::edge(1, 2).
0.7::edge(1, 3).
0.4::edge(2, 3).
```

(b) The probabilistic facts of the small graph shown in Figure 2.1.

```
:- module(directed_graph, [edge/2]).
0.3::edge(1, 2).
0.7::edge(1, 3).
0.4::edge(2, 3).
0.8::edge(3, 4).
0.6::edge(3, 5).
0.2::edge(4, 5).
```

(c) The probabilistic facts of the example graph shown in Figure 6.1.

```
:- use_module(problog).
:- use_module(path, [path/2]).
:- use_module(small_graph, []).
:- use_module(directed_graph, []).

experiment1:-
  problog_forget(directed_graph, [edge/2]),
  problog_import(small_graph, [edge/2]),
  problog_exact(path(1,3), P).
experiment2:-
  problog_forget(small_graph, [edge/2]),
  problog_import(directed_graph, [edge/2]),
  problog_exact(path(1,5), P).
```

(d) The definition of two different experiments using modules.

Figure 6.8: A ProbLog program using the module system to separate the background, two different data sets and the experiments.

The second benchmark has the goal before the nesting. In this way, an engine consumes resources before starting a nested engine. To avoid executing the nested call after each successfull derivation of the goal, `path('HGNC_582', 'HGNC_983')`, we transform the goal into `((path('HGNC_582','HGNC_983'), fail);true)`[8]. The query for this benchmark is of the form: $exact((G2,\ exact((G2,\ exact((...)))))))$. This benchmark measures the the impact of previously proven goals to newer engines.

Our final benchmark is the combination of the two above. The query is of the form: $exact((G2, exact((G2, exact((...)), G1)), G1))$.

Executing the presented queries with no nested meta-calls we achieve the following execution times: 19069, 12080, 31191 milliseconds respectively for the first, second and third query.

The results of our benchmarks are presented in Table 6.1. The left hand side of the table presents the average times of 10 executions for each call at each nesting. The right hand side presents the average time any nested goal took at each execution. First thing we notice is all our averages are very close and that the standard deviation is low. From this observation we can safely claim that the depth of nesting does not impose any significant loss of time. On the other hand we do notice a very small overhead around 1% going from no nested calls to any nested call. The path query we use is a representative ProbLog path query. Furthermore, from the nesting desing we know that the overhead for nesting ProbLog queries does not vary among different queries. So we can safely state that a similar overhead should appear regardless the query.

6.7 Conclusions

In this chapter we briefly presented the ProbLog2011 implementation based on the ProbLog engines strategy. This implementation is able to perform nested inference. Furthermore, it achieves modularity and properly performs the needed memory management. The underlying idea is to abstract the required information for the probabilistic bookkeeping in a parametrised ProbLog engine. By storing the engines in a stack, we achieve probabilistic meta-calls. We introduced the general probabilistic query `problog_inference/4`, which allows us to perform nested inference and, further more, we presented how to implement `problog_not/1` and `problog_answers/3` with meta-calls. We also briefly illustrated the functionality of meta-calls in probabilistic modelling. Also we presented a more structural way to handle memory, which ensures no

[8] This will also reduce somewhat the work load but still retain it suitable for our experiment.

Query: exact((exact((exact((...)),G1)),G1))					
Depth	Avg Time (msec)	Standard Deviation	Run	Avg. Time (msec)	Standard Deviation
1	19347.7	191.72	1	19447.2	125.89
2	19281.3	120.29	2	19382.4	312.21
3	19380.7	202.51	3	19287.2	216.96
4	19279.2	114.64	4	19253.2	85.79
5	19318.3	215.61	5	19231.9	144.35
6	19240.1	155.73	6	19226.2	190.78
7	19435.6	175.61	7	19561.5	103.54
8	19268.2	99.21	8	19320.0	140.81
9	19338.0	112.58	9	19285.8	89.11
10	19361.3	340.75	10	19255.0	91.08
Query: exact((G2,exact((G2,exact((...))))))					
Depth	Avg Time (msec)	Standard Deviation	Run	Avg. Time (msec)	Standard Deviation
1	12202.5	124.22	1	12252.6	154.03
2	12301.2	145.41	2	12269.5	73.19
3	12269.2	130.31	3	12275.9	141.36
4	12294.7	134.93	4	12240.8	108.59
5	12256.9	163.27	5	12218.3	188.52
6	12189.6	152.83	6	12272.3	176.31
7	12169.9	140.61	7	12259.7	196.27
8	12148.4	134.18	8	12234.1	141.92
9	12312.4	119.22	9	12257.6	165.59
10	12346.9	95.93	10	12210.9	76.06
Query: exact((G2,exact((G2,exact((...)),G1)),G1))					
Depth	Avg Time (msec)	Standard Deviation	Run	Avg. Time (msec)	Standard Deviation
1	31509.1	250.17	1	31494.3	300.57
2	31749.2	362.68	2	31444.8	205.96
3	31667.5	242.61	3	31529.5	138.27
4	31416.1	121.02	4	31406.4	218.04
5	31379.6	145.27	5	31453.1	363.23
6	31331.7	270.83	6	31680.6	433.29
7	31930.7	301.64	7	31562.7	121.24
8	31466.0	322.14	8	31686.2	330.58
9	31588.4	121.49	9	31482.0	203.66
10	31331.8	108.03	10	31630.5	397.61

Table 6.1: Experimental results.

garbage remains after the termination of a query. Finally, the ProbLog2011 implementation is module aware and abides to the rules of Yap Prolog's module system. As a result of that the ProbLog2011 implementation is easier to be used as a library and possible to be used for programming in the large. To verify our approach for nested inference, we performed some experiments and measured an overhead of approximately 1% for our strategy. This overhead is independent from the type of query.

Chapter 7

Applications

In this chapter we present two applications that use some of the features we presented. Both applications would have been very difficult to address without the added features. The difficulties lie both in computational and expressive power.

The two applications presented were motivating examples for ProbLog both in order to expand the system and in order to stress test its applicability in real life. The first application as we will present is a very difficult computationally task. In order to solve the problem a first attempt was made using external from ProbLog tools [88]. In order to tackle the newly discovered difficulties we extended ProbLog's tabling to fully handle cycles and abide to the well found semantics. Further, we implemented some syntactic sugar to define different atoms which are the negation of each other. For example: male and female.

The second application was a stress test for ProbLog both in expressive power and in scalability. We analysed a network protocol using ProbLog to model its behavior. The challenges lied both in the modelling of the protocol as in making a scalable analysis application. Using several of ProbLog's features such as tabling, general negation and dynamic probabilistic facts, we implemented a scalable analysis application that provides interesting extra information in addition of that from the state-of-the-art simulation techniques provided until now.

7.1 First Order ProbLog

The first application of the features that we presented previously, was to extend the expressibility of ProbLog to full first order logic. This section presents joint work with Maurice Bruynooghe, Angelika Kimmig, Bernd Gutmann, Joost Vennekens, Gerda Janssens and Luc De Raedt published in [8, 88]. The main contributions of the author are the implementation of the necessary ProbLog extensions to handle first order logic such as handling cycles with negation, providing syntax for marking atoms as the negation of each other, automatically calculating the needed normalization factor. As well as performing the experimental part. The paper [8] is directly presented here.

7.1.1 Introduction

A problem for languages that combine probability theory with *expressive* logical formulas is that probabilistic and logical knowledge might interact in complex ways, leading to a semantics that is too complicated to understand, and inference/learning that is too slow to be of practical use. Initial research into Probabilistic Logic Learning therefore mainly focused on adding probabilities to restricted logical languages, such as definite clause logic [33, 73]. Since then, the trend has been to extend the expressibility of the logical language, e.g., to normal clauses [63]. In the past few years there has been a lot of attention to Markov Logic [67], a probabilistic logic used in statistical relational learning [21]. Markov Logic consists of a set of weighted logical formulas each of which can be regarded as a soft constraint. If such a soft constraint is violated by a possible world, the probability of the world does not become zero, as in first order logic, but gracefully decreases. The higher the weight, the harder the constraint becomes, with infinite weights corresponding to the usual logical interpretation. While Markov Logic has been very successful as a framework for statistical relational learning and has been applied in several challenging applications, such as natural language processing and entity resolution, Markov Logic models are hard to interpret and not really suitable as a knowledge representation tool. The reason is that the weights cannot directly be interpreted as probabilities and also that the probability of a formula depends non-linearly on all weights in the theory.

This section contributes a new formalism, called First Order ProbLog, using the Markov Logic idea of soft constraints, but in which each first order formula is annotated with the probability that a grounding of the formula – independently of anything else – holds. This gives a logic very similar to that of Nilson [52]. The questions arise whether inference is feasible and how one can cope with

inconsistency. These questions are explored in this section. Using ideas of Stickel [81], we translate a theory in our logic into a ProbLog program [35] that also includes clauses for inferring inconsistency (with head *false*). We show how to use the ProbLog machinery to assign a minimal and a maximal probability to the truth of the query while taking consistency requirements into account.

7.1.2 First Order ProbLog and its Semantics

A ProbLog program [35] consists of a set of facts annotated with probabilities – called *probabilistic facts* – together with standard definite clauses that can have positive and negative probabilistic facts in their body. The semantics is defined through belief sets which correspond to least Herbrand models of the clauses together with subsets of the probabilistic facts. If fact f_i is annotated with p_i, f_i is included in a belief set with probability p_i and left out with probability $1 - p_i$. The different facts are assumed to be probabilistically independent, however, negative probabilistic facts in clause bodies allow the user to enforce a choice between two clauses.

In this section we define FOProbLog, a language similar to ProbLog, but using full first order logic formulas instead of definite clauses. A FOProbLog statement is of the form $\forall \overline{x}.\Psi_1 : \alpha_1 \vee \ldots \vee \Psi_n : \alpha_n$ where the α_i are non-zero probabilities with sum 1 and the Ψ_i are first order formulas with free variables included in $\overline{x}$. If Ψ_i is *true*, it can be omitted (in which case the sum of the probabilities becomes smaller than 1); if $\alpha_1 = 1$, it can be omitted as well. These formulas express independent beliefs about the world. Each belief is disjunctive: we believe that for each grounding of $\overline{x}$ exactly one of a number of possibilities holds, but we don't know which one, so we attach a probability to each of the disjuncts. We can now have a number of different "complete belief sets", which are formed by believing precisely one disjunct from every grounded disjunction according to its probability.

Ex. 1. *Our running example uses a domain with a single constant $Floris$ and theories consisting of subsets of the following formulas*

(1) $male(Floris) : 0.4 \vee female(Floris) : 0.6$

(2) $\forall x.(cs(x) \rightarrow male(x) : 0.8) \vee (cs(x) \rightarrow female(x) : 0.2)$

(3) $\forall x.\neg(male(x) \wedge female(x))$

(4) $cs(Floris)$

If the theory contains only formula (1), expressing a prior belief about the name Floris being male or female, each complete belief set either includes the fact

$male(Floris)$ (with probability 0.4) or the fact $female(Floris)$ (with probability 0.6).
After adding the second formula, each complete belief set also contains one of $cs(Floris) \rightarrow male(Floris)$ and $cs(Floris) \rightarrow female(Floris)$. One of the resulting four belief sets, with probability $0.8 \cdot 0.6$, contains the formulas $female(Floris)$ and $cs(Floris) \rightarrow male(Floris)$.
Adding the third formula to the theory, the extension of the latter belief set allows one to infer $\neg male(Floris)$ and $\neg cs(Floris)$.

Belief sets in FOProbLog can be inconsistent. One should assign zero probability to such belief sets. With s a total choice and $c(s)$ expressing consistency of s, $P(s|c(s))$, the *normalized* probability of a total choice s, is given by $P(s) \cdot P(c(s)|s)/P(c(s))$ where $P(c(s)|s)$ is either 1 or 0.

Ex. 2. *Using all four formulas of Example 1 makes the belief set with formulas $female(Floris)$, $cs(Floris)$ and $cs(Floris) \rightarrow male(Floris)$ (as well as another one) inconsistent. The probability of an inconsistent belief set is $0.8 \cdot 0.6 + 0.4 \cdot 0.2 = 0.56$. Normalizing the probabilities of the two consistent belief sets, we obtain $0.4 \cdot 0.8/(1 - 0.56) = 0.73$ for the one containing $male(Floris)$ and 0.27 for the one with $female(Floris)$. Using all our independent formulas about maleness, we thus derive that Floris is more likely male than female, contrary to the belief about the name Floris.*

While the above example can be modeled in ProbLog, the probabilities inferred there will have a different meaning, as ProbLog adopts a different view on consistency. In a ProbLog program containing clauses $male(Floris)$. with probability 0.4 and $male(X)$:-$cs(X)$. with probability 0.8, those clauses *together* define the *male* predicate and, by closed world assumption, also $\neg male$, the equivalent of $female$. There are 4 different belief sets, one with both clauses, two with one clause and one without clauses, but none with an inconsistent belief set where $Floris$ is both male and not male. The probability of $male(Floris)$ is thus $0.8 + 0.4 - 0.8 \cdot 0.4 = 0.88$ instead of 0.73.

In the rest of the section, we use a more basic form of FOProbLog which, similarly to ProbLog, distinguishes *probabilistic facts* from the *logical part* of a theory.

Definition 14 (FOProbLog theory)**.** *A* FOProbLog theory $T = (PF, \Phi)$ *consists of a* probabilistic part PF *and a* logical part Φ*. Its predicates are split into sets Σ_P and Σ_L of* probabilistic *and* logical predicates*, respectively. PF contains* probabilistic facts *of the form $pf(\overline{x}) : \alpha$, with α a non-zero probability, $\overline{x}$ n different variables and $pf/n \in \Sigma_P$. Each ground instance $pf(\overline{x})\theta$ of such a fact is true with probability α, and is not a grounding of any other probabilistic fact. Different instances (of the same or of different probabilistic facts) are*

probabilistically independent. An assignment of a truth value to a ground instance of a probabilistic fact is called atomic choice*, an assignment to all of them a* total choice*. The probability prob(s) of a total choice s is the product of all α_i for true instances $pf_i(\overline{x})\theta : \alpha_i$ and all $(1 - \alpha_j)$ for false instances $pf_j(\overline{x})\theta : \alpha_j$. The* logical part Φ *of the theory consists of a set of implications $\forall\overline{x}.P(\overline{x}) \rightarrow F(\overline{x})$, where $F(\overline{x})$ is a first-order formula with free variables $\overline{x}$ using only predicates from Σ_L, and $P(\overline{x})$ is a conjunction of literals with predicates from Σ_P*[1]*.*

Ex. 3. *Formulas (1)-(4) of Example 1 are now written as*

(1a) $pf\_mn(Floris) : 0.4$

(1b) $pf\_mn(Floris) \rightarrow male(Floris)$

(1c) $\neg pf\_mn(Floris) \rightarrow female(Floris)$

(2a) $pf\_cs(x) : 0.8$

(2b) $\forall x.pf\_cs(x) \rightarrow (cs(x) \rightarrow male(x))$

(2c) $\forall x.\neg pf\_cs(x) \rightarrow (cs(x) \rightarrow female(x))$

(3) $\forall x.\neg(male(x) \wedge female(x))$

(4) $cs(Floris)$

The probabilistic part of formulas (3) and (4) is true *and omitted.*

To define the semantics of our logic, we fix a domain D and consider the set $\mathcal{W}_D$ of all possible worlds in D, i.e., each I in $\mathcal{W}_D$ is an interpretation for the vocabulary of the theory with D as its domain; in particular I assigns *true* or *false* to each probabilistic and each logical ground atom.

Notation 1. *In what follows, we use $I \models s$ with I an interpretation and s a total choice to express that I makes the same truth assignments to the probabilistic facts as s. We often say I extends s.*

Definition 15 (Semantics)**.** *Let $T = (PF, \Phi)$ be a theory and D a domain.*

Let Cons be the set of total choices s such that there exists an interpretation I that extends s ($I \models s$) and is a model of Φ ($I \models \Phi$), i.e., the total belief set corresponding to the total choice is consistent.

[1] A formula $\forall\overline{x}.\Psi_1 : \alpha_1 \vee \ldots \vee \Psi_n : \alpha_n$ is split into n formulas $\forall\overline{x}.\ P_i(\overline{x}) \rightarrow \Psi_i(\overline{x})$ and $n-1$ probabilistic facts are introduced, their probabilities are computed and the probabilistic part $P_i(\overline{x})$ of each formula contains a conjunction of probabilistic literals as described in [36].

$\widehat{prob}(s)$ *(the normalized probability of a total choice) is given by* $prob(s) \cdot prob(s \in Cons|s) / \sum_{s \in Cons} prob(s)$ *where* $prob(s \in Cons|s)$ *is 1 when* s *is consistent and 0 otherwise.*

A probability distribution μ *over* $\mathcal{W}_D$ *is a* model *of* T*, denoted* $\mu \models T$*, if and only if (i) for all* $I : I \not\models \Phi$ *implies* $\mu(I) = 0$*; (ii) for each total choice* s*:* $\widehat{prob}(s) = \sum_{I \models s} \mu(I)$*.*

The theory is inconsistent when no consistent total choice exists. In that case, no probability distribution is defined.

Note that the normalization introduced here corresponds to conditioning on total choices with consistent extensions, which will be exploited in Section 7.1.3 to calculate probabilities of queries.

In contrast to the above informal exposition, where belief sets contained logical formulas and hence had interpretations over the logical predicates only, now interpretations assign truth to both probabilistic and logical predicates.

Ex. 4. *In our example, with two probabilistic and three logical predicates and a single constant, we now have 32 possible interpretations. However, as before, only two of the four total choices can be extended in a belief set that is a model of the logical part; moreover, these models are unique. One of them is the belief set* $\{pf\_cs(Floris), pf\_mn(Floris), cs(Floris), male(Floris)\}$*. It extends the total choice* $\{pf\_cs(Floris), pf\_mn(Floris)\}$*, is a model of the logical part of the theory and is assigned probability* 0.73 *in the probability distribution that is a model. The other one,* $\{cs(Floris), female(Floris)\}$*, extends the total choice where both probabilistic atoms are false; it is assigned probability* 0.27*. The total choices that make one probabilistic atom true and the other one false cannot be extended in a model of the logical part; their probability mass has been redistributed over the other ones.*

In the above example, there is at most one belief set that extends a total choice and hence only one probability distribution that is a model. This does not hold in general. FOProbLog theories impose *constraints*, and any distribution satisfying these constraints is a model. Instead of making additional assumptions (such as the principle of indifference) to choose a specific distribution, we will restrict ourselves to sound inference, and will infer probability intervals only.

Ex. 5. *Consider a theory consisting of the probabilistic fact* $pf(x) : 0.7$ *and the logical formula* $\forall x.pf(x) \rightarrow p(x)$ *with a single element domain* $\{A\}$*. The total choice* $pf(A)$ *can be extended into the belief set* $\{pf(A), p(A)\}$ *and hence* $\mu(\{pf(A), p(A)\}) = 0.7$*. The empty total choice can be extended in two ways, namely* $\emptyset$ *and* $\{p(A)\}$*; hence* $\mu(\emptyset) + \mu(\{p(A)\}) = 0.3$*. Any distribution that*

satisfies the latter constraint is a model. The principle of indifference would assign probability 0.15 *to each of the latter two belief sets and hence* 0.85 *to* $p(A)$ *and* 0.15 *to* $\neg p(A)$*. We infer that the probability of* $p(A)$ *is in the interval* $[0.7, 1.0]$ *and that of* $\neg p(A)$ *in* $[0.0, 0.3]$*.*

7.1.3 Inference

We are now interested in deciding what the theory T allows us to conclude over the probability of some query formula Q. For each given domain, T has a non-empty set $\hat{\mathcal{M}}$ of models μ. Each $\mu \in \hat{\mathcal{M}}$ assigns a particular probability $\mu(Q) = \sum_{I \models Q} \mu(I)$ to Q, yielding, in general, a non-empty probability interval $[\min_{\mu \in \hat{\mathcal{M}}} \mu(Q), \max_{\mu \in \hat{\mathcal{M}}} \mu(Q)]$. We are now interested in the inference task of determining this interval. Because $\max_{\mu \in \hat{\mathcal{M}}} \mu(Q)$ must be equal to $1 - \min_{\mu \in \hat{\mathcal{M}}} \mu(\neg Q)$, we can restrict attention to the computation of a lower bound on the probability of a query. As the following theorem shows, we can compute this lower bound without having to consider any specific model μ of T.

Theorem 9. *Let* $\hat{\mathcal{M}}$ *be the non-empty set of models of a consistent theory* T*. Then* $\min_{\mu \in \hat{\mathcal{M}}} \mu(Q) = \sum_{s \models Q} \widehat{prob}(s)$*, where* $s \models Q$ *means that* Q *holds in all* $I \in \mathcal{W}_D$ *such that* $I \models s$*.*

Proof. Let $W_Q = \{I | I \models Q\}$ and $W_{SQ} = \{I | \exists s : I \models s \wedge s \models Q\}$. By definition, $\mu(Q) = \sum_{I \in W_Q} \mu(I)$. Also by definition, $\sum_{s \models Q} \widehat{prob}(s) = \sum_{I \in W_{SQ}} \mu(I)$. Obviously, $\sum_{I \in W_Q} \mu(I) \geq \sum_{I \in W_{SQ}} \mu(I)$. To prove the theorem, it suffices to show that there exists a probability distribution μ' that is a model and for which the equality holds. The equality holds if μ' assigns 0 probability to all $I \in W_Q \setminus W_{SQ}$. For such I, we can distinguish two cases. Either I restricted to the probabilistic atoms gives a total choice that cannot be extended into a consistent belief set, in which case all distributions that are a model assign 0 probability to I. Or I restricted to the probabilistic atoms gives a total choice s with at least one I' extending s such that $I' \not\models Q$. In this case, μ' can be chosen to assign all the probability mass $\widehat{prob}(s)$ to interpretations that are not models of Q and hence set $\mu'(I) = 0$. Hence $\mu'(I) = 0$ for all $I \in W_Q \setminus W_{SQ}$. □

The normalized probability of a total choice that can be extended into a consistent belief set is obtained by dividing its probability by the probability of all such total choices. The latter is the complement of the probability of those total choices where this is not possible, that is, the belief sets containing $false$. Furthermore, recall that $\widehat{prob}(s) = 0$ for total choices without consistent extension. Hence, we also have $\min_{\mu \in \hat{\mathcal{M}}} \mu(Q) =$

$\left(\sum_{s\models Q\wedge\neg false} prob(s)\right) / \left(1 - \sum_{s\models false} prob(s)\right)$, which in fact corresponds to the conditional probability $prob(Q|\neg false)$.

Ex. 6. *Let us reconsider the theory consisting of* $pf(x) : 0.7$ *and* $\forall x.pf(x) \rightarrow p(x)$. *The query* $p(A)$ *can be proven with a total choice that includes the probabilistic fact* $pf(A)$. *As no total choice results in inconsistency, this gives a minimal probability of* 0.7. *No proofs are possible for* $\neg p(A)$, *hence the maximal probability of* $p(A)$ *is* 1.

The number of total choices is exponential in the number of probabilistic facts, hence it is not feasible to evaluate a query for each total choice. More promising is the ProbLog approach that enumerates all proofs in terms of the probabilistic facts they use and encodes this information in a Reduced Ordered Binary Decision Diagram (ROBDD), which can be used to compute the probability of the query [37]. The main difference is that we do not work with Horn clauses but with full first order logic, hence we cannot, as in ProbLog, use the SLD proof procedure to enumerate proofs. However, for quickly building a first prototype and for making maximal use of the ProbLog technology, it is interesting to stay as much as possible within Prolog. Here the work of Stickel [81] that describes how to use Prolog technology for building a first order logic theorem prover comes to the rescue and allows us to convert FOProbLog theories into ProbLog programs.

The basic idea of Stickel's work is to transform formulas into clausal form and to encode each n-literal clause $l_1 \vee \ldots \vee l_n$ by n Horn clauses of the form l_i `:-`$not(l_1), \ldots, not(l_{i-1}), not(l_{i+1}), \ldots, not(l_n)$.. To obtain Horn clauses, negative literals are encoded by positive ones: For each predicate p/n, Stickel introduces a not_p/n and the negation of $p(\overline{t})$ is replaced by $not\_p(\overline{t})$. In the context of FOProbLog, for each formula $\forall\overline{x}.P(\overline{x}) \rightarrow F(\overline{x})$, the logical part $F(\overline{x})$ is converted in clausal form, Stickel's transformation is applied to the resulting clauses, and the conjunction $P(\overline{x})$ is added to the bodies of the final Horn clauses.

Ex. 7. *Applying the transformation to our example gives the following ProbLog program (we switch to Prolog notation for constants and variables):*

(1a) `0.4::pf_mn(floris)`

(1b) `male(floris) :- pf_mn(floris).`

(1c) `female(floris) :- not(pf_mn(floris)).`

(2a) `0.8::pf_cs(X)`

(2b) `male(X) :- cs(X), pf_cs(X).`

(2b) `not_cs(X) :- not_male(X), pf_cs(X).`

(2c) `female(X) :- cs(X), not(pf_cs(X)).`

(2c) `not_cs(X) :- not_female(X), not(pf_cs(X)).`

(3a) `not_female(X) :- male(X).`

(3b) `not_male(X) :- female(X).`

(4) `cs(floris).`

The transformation also has to generate clauses with $false$ in the head, as those are needed during inference to take consistency into account. One way to do so is to use the knowledge that a resolution proof must use at least one clause $pf(\overline{x}) \rightarrow not(l_1) \vee \ldots \vee not(l_n)$ with only negative literals (to have all such clauses as "set of support"). By adding $false$:-$l_1, \ldots l_n, pf(\overline{x})$ for each such clause, we ensure that all proofs of *false* employ such clauses. Alternatively, one can use the positive clauses (clauses with only positive literals) as set of support. Simply adding a clause $false$:-$p(\overline{X}), not\_p(\overline{X})$ for each logical predicate p/n is also possible, but will typically result in a lot of redundant proofs.

Ex. 8. *In our example, using negative clauses as set of support would add:*

```
false :- male(X), female(X).
```

whereas using positive clauses would add:

```
false :- not_cs(floris).
false :- not_male(floris), pf_mn(floris).
false :- not_female(floris), not(pf_mn(floris)).
```

Stickel additionally needs to modify certain parts of the inference mechanism. First, Prolog's input resolution is extended with ancestor resolution: resolution between the current goal and one of the ancestors in the linear chain from query to current goal. A convenient way to do so in the setting of Prolog's depth first left to right execution policy is to keep track of the selected literals with uncompleted proof. If in the uncompleted proof of a literal $p(\overline{t})$ (or $not\_p(\overline{t})$), its negation $not\_p(\overline{s})$ (or $p(\overline{s})$) is selected, the latter literal can be resolved by unifying its atom with the atom of the former (i.e. unifying $\overline{t}$ with $\overline{s}$) [81]. Also, care is required to avoid unsound unification. In general, some unifications may have to be replaced by a sound variant that performs an occur check. When proving queries in knowledge bases, it is unlikely that the occur check is needed. Finally, performing iterative deepening avoids that the search gets trapped in an infinite branch.

Given the transformed theory and a query Q, we can now use ProbLog's machinery to construct two Boolean formulas ψ and ϕ representing the proofs of Q and the proofs of *false*, respectively. The formula $\psi \wedge \neg\phi$ thus

restricts the proofs of Q to consistent belief sets, whereas ϕ is used for normalization, i.e. $\min_{\mu \in \hat{\mathcal{M}}} \mu(Q)$ is obtained by $P(\psi \wedge \neg\phi)/(1 - P(\phi))$, where both intermediate results are calculated using ProbLog's ROBDD algorithm. For the running example, we obtain $\phi = (\neg pf\_cs(floris) \wedge pf\_mn(floris)) \vee (pf\_cs(floris) \wedge \neg pf\_mn(floris))$, and, for the query $? - male(floris)$, $\psi \wedge \neg\phi = pf\_cs(floris) \wedge pf\_mn(floris)$.

7.1.4 A Case Study

To assess performance in practice, we explored an application of Markov Logic in entity resolution [79]. We will first review some details of the application, and then describe how to encode key parts of it directly in ProbLog.

In this application, a fact database containing information about bibliographic entries such as authors and titles is combined with a probabilistic first order theory describing when two database references of the same type are likely the same. The purpose of entity resolution is to compute the probability that two keys or two authors (e.g. `author_william_w_cohen_` and `author_w_w_cohen_`) refer to the same object, i.e., to be able to get probabilities for the queries $sameBib(b1, b2)$ and $sameAuthor(a1, a2)$, as well as for $not\_sameBib(b1, b2)$ and $not\_sameAuthor(a1, a2)$.

Part of the facts database is a binary encoding of a ternary relation between authors, paper title and publication venue. A tuple $(author, paper, venue)$ is encoded as $author(bib, author)$, $title(bib, paper)$ and $venue(bib, venue)$ with bib an arbitrary key identifying a bibliographic entry. As formulas for authors, papers and venues are similar in structure, we will restrict our discussion to those concerning authors and bibliographic entries. Authors are strings (extracted from web pages) that are composed from different words (e.g. `author_william_w_cohen_` and `author_w_w_cohen_` are composed of the words `word_cohen`, `word_w`, and `word_william`. This information is encoded in the database relation $hasWordAuthor(author, word)$.

Figure 7.1 shows the first order logic formulas used in [79] to model a key part of the application. A first simple set of rules states that different strings refer to different authors or different bibliographical entries, respectively. The next formula is an example of rules expressing that the relations about sameness are likely transitive, while the remaining ones link authors to bibliographical entries and words to authors, respectively.

Figure 7.2 shows the result of translating those formulas into ProbLog. We cannot blindly translate the first order theory. The main reason is that the closed world assumption is applied on the large database. Explicitly adding

$\neg sameBib(b1, b2)$

$\neg sameAuthor(a1, a2)$

$\forall\ b1, b2, b3 : sameBib(b1, b2) \wedge sameBib(b2, b3) \rightarrow$
$sameBib(b1, b3)$

$\forall\ b1, b2, a1, a2 : author(b1, a1) \wedge author(b2, a2) \wedge$
$sameAuthor(a1, a2) \rightarrow sameBib(b1, b2)$

$\forall\ a1, a2, w : hasWordAuthor(a1, w) \wedge$
$hasWordAuthor(a2, w) \rightarrow sameAuthor(a1, a2)$

$\forall a1, a2, w : \neg hasWordAuthor(a1, w) \wedge$
$hasWordAuthor(a2, w) \rightarrow \neg sameAuthor(a1, a2)$

Figure 7.1: A key part of the logical theory.

all negative facts would result in an exponential blow-up in the size of the logical theory. Therefore, we instead use negation as failure to encode them and add the first group of rules to our theory. Note that calls to these predicates are correctly executed only when all arguments are ground. This need not be a problem because ProbLog aims at inference of ground queries. Another difficulty – if we stick to the negative clauses as set of support – is that, for each tuple (b, a) not in the database, we would need to add a clause $false$ `:-` $author(b, a)$. As we cannot enumerate all such pairs (b, a), we resort to a different approach to define $false$. Using a positive set of support, we should add $false$ `:-` $not\_author(b, a)$ for each pair in the author relation. As the theory does not contain clauses with $author/2$ in the head, we can use the author relation to generate the pairs. Furthermore, this approach introduces less redundancy in the search space of proofs for $false$ than the approach of simply adding a rule $false$ `:-` $p(\overline{X}), not\_p(\overline{X})$ for each of the predicates $author/2$, $hasWordAuthor/2$, $sameAuthor/2$, and $sameBib/2$. The next block of rules states that identical strings refer to identical entities (with certainty), different strings to different entities (with a rather high probability). As we have added positive clauses, we also have to add the next block of extra rules for proving *false*. Here `domBib/1` and `domAuthor/1` are used to generate the values for the keys and the authors respectively (appropriate ground facts have to be added to the database). Each clause about transitivity gives rise to three ProbLog clauses, which form the next block. Note that loop checking as well as ancestor resolution are important for doing exact inference with these clauses.

```
not_author(B,A) :- not(author(B,A)).
not_hasWordAuthor(A,W) :-
    not(hasWordAuthor(A,W)).

false :- author(B,A), not_author(B,A).
false :- hasWordAuthor(A,W),
    not_hasWordAuthor(A,W).

sameBib(B,B).
sameAuthor(A,A).
not_sameBib(B1,B2) :- B1 \= B2, pf1(B1,B2).
not_sameAuthor(A1,A2) :- A1 \= A2, pf2(A1,A2).

false :- domBib(B), not_sameBib(B,B).
false :- domAuthor(A), not_sameAuthor(A,A).

sameBib(B1,B3) :- sameBib(B1,B2),
    sameBib(B2,B3), pf5(B1,B2,B3).
not_sameBib(B1,B2) :- sameBib(B2,B3),
    not_sameBib(B1,B3), pf5(B1,B2,B3).
not_sameBib(B2,B3) :- sameBib(B1,B2),
    not_sameBib(B1,B3), pf5(B1,B2,B3).

sameBib(B1,B2) :-
    author(B1,A1), author(B2,A2),
    sameAuthor(A1,A2), pf9(B1,B2,A1,A2).
not_sameAuthor(A1,A2) :-
    author(B1,A1), author(B2,A2),
    not_sameBib(B1,B2), pf9(B1,B2,A1,A2).
not_author(B1,A1) :-
    sameAuthor(A1,A2), author(B2,A2),
    not_sameBib(B1,B2), pf9(B1,B2,A1,A2).
not_author(B2,A2) :-
    sameAuthor(A1,A2), author(B1,A1),
    not_sameBib(B1,B2), pf9(B1,B2,A1,A2).
```

Figure 7.2: Key clauses in ProbLog.

The clauses linking authors to bibliographic entries give rise to the last block. Finally, as the rules linking authors and words can be translated in the same way as the previous clauses, we do not further elaborate them.

However, one important observation is that the sharing of words in different author names, paper titles, and venues, results in a densely connected network to the point that almost for every pair $(b1, b2)$ of keys, one can prove $sameBib(b1, b2)$ as well as $not\_sameBib(b1, b2)$. Hence, there are a lot of inconsistent total choices, it is essential to perform normalization and to execute the $false$ query. Clearly, this is very demanding.

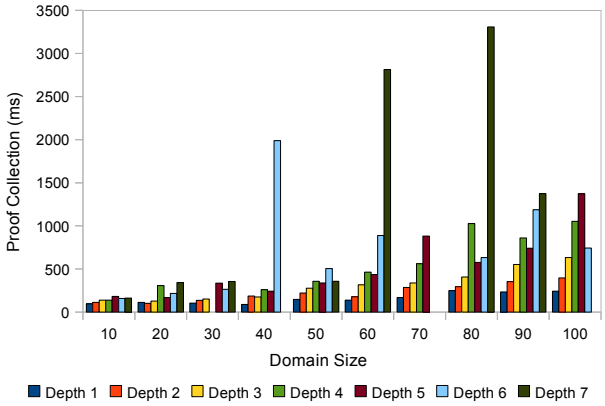


Figure 7.3: Friends experiments.

7.1.5 Experiments

We set up experiments to investigate the feasibility of inference in FOProbLog. We focus on the query *false*, as this can involve all possible queries in a theory and thus is the most challenging query. We used a version of ProbLog with tabling [42, 43] which we extended with ancestor resolution and iterative deepening as suggested by Stickel [81]. Experiments are performed on an Intel Core 2 Duo CPU at 3.00GHz with 2GB RAM running Ubuntu 8.04.2 Linux. In our experiments ProbLog is used to first collect the proofs and construct a Boolean formula for false and to then compute the probability by constructing a ROBDD.

The first experiment uses the following FOProbLog theory:

$$\forall x, y, z.\ pf1(x, y, z) \rightarrow (Fr(x, y) \wedge Fr(y, z) \rightarrow Fr(x, z))$$
$$\forall x.\ pf2(x) \rightarrow (Sm(x) \rightarrow Ca(x))$$
$$\forall x, y.\ pf3(x, y) \rightarrow (Fr(x, y) \rightarrow (Sm(x) \leftrightarrow Sm(y)))$$

and randomly generated databases to investigate the influence of the domain size and the maximum depth used in iterative deepening. The latter limits transitive closure. The runtimes to obtain the Boolean formula representing *false* are presented in Figure 7.3; a missing bar indicates that the experiment failed to properly terminate. We notice that the search space grows exponentially with the depth limit, while the influence of the domain size is less drastic. Assessing the probability of the Boolean formula through ROBDDs is feasible within one minute for most of these experiments.

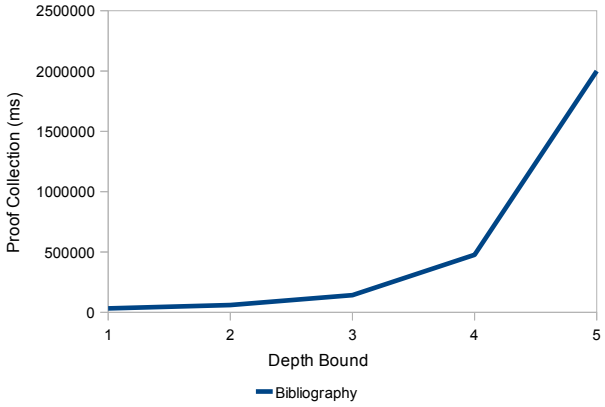


Figure 7.4: Bibliography experiments.

The second experiment uses the entity resolution model of Section 7.1.4 with the full database of [79] containing 1295 bibliographic entries involving roughly 90 authors, 400 venues, 200 titles and 2700 words. Figure 7.4 shows the time to obtain the Boolean formula representing *false* for increasing maximum search depth. The results confirm that search time increases exponentially, as can be expected for such a densly connected problem. The limiting factor of our current prototype is the size of the resulting ROBDDs; in this application, they are too large to be constructed within a time limit of one hour.

It is worth mentioning that in both experiments, the Boolean formulas for other queries are far smaller. However, in most cases, the time still increases exponentially with the depth bound. Furthermore, our current results do not permit general conclusions about the gap between minimal and maximal probabilities obtained for each query.

7.1.6 Discussion

In FOProbLog, probabilities are associated with formulas in first order logic. The assumption that these formulas are independent allows us to define a semantics based on complete belief sets. Inference in FOProbLog is the task of calculating the probability that a query can be proven in a randomly selected complete belief set. Randomly selected belief sets can be inconsistent. The probability of such a selection can be computed. Normalization assigns 0 probability to inconsistent total choices and redistributes their probability mass

over the consistent ones. The probability of the truth of a query is computed with respect to consistent total choices.

One could argue that inconsistency is an indication of the violation of the independence assumption. Interestingly, inconsistent belief sets typically arise after adding factual knowledge (adding *cs(Floris)* or adding the database of bibliographical information). This factual knowledge can be seen as *evidence* that excludes certain belief sets. Inconsistency in the non-factual part is thus not an indication of poor design, as it can make sense to add a formula as evidence to an existing theory. This refines the theory by ruling out certain complete belief sets, and hence causes a redistribution of the probability mass.

A related issue is the presence of redundant formulas. One might consider a situation where two independent experts contribute exactly the same formula to a theory, each of them claiming this formula with probability p_1 and p_2 respectively. The probability that this formula then holds in a complete belief set is $p_1 + p_2 - p_1 \cdot p_2$. This makes sense when the expertise is really based on different knowledge. However, with each additional independent expert coming up with the same formula and assigning some probability p_i to it, the overall probability would further increase (unless the experts explicitly assign a probability to both the formula and its negation). It is plausible that different experts have used the same common knowledge to come up with their formula and hence the independence assumption is violated. In general, the interaction can be more subtle. For example, in the context of our case study, one could add that *sameBib* is a symmetrical relation. This will likely result in different probabilities for queries[2].

One should be aware that the probability annotating a formula is the probability that a ground instance is included in a belief set. To know the minimal probability that the formula can be proven in a randomly selected consistent belief set, one should query for it. This probability can be different for different instances and our current prototype supports querying only for ground instances.

In reality, given a theory and a number of datasets with evidence, one will typically learn probabilities. In that case it is very desirable to have a logical part that avoids inconsistencies and redundancies as much as possible. Indeed, consider again the extreme case that one has two copies of the same formula in the theory. From the evidence, one will derive only the value of $p_1+p_2-p_1\cdot p_2$, so there is a certain randomness in allocating p_1 and p_2. Hence, the logical theory better avoids inconsistencies and redundancies to facilitate the understanding of the learned probabilities.

[2]We have followed as close as possible the original Markov Logic formulation as we only wanted to evaluate the feasibility of inference.

7.1.7 Related Work and Conclusion

We have proposed FOProbLog, a simple but very expressive probabilistic logic and defined its semantics. As Markov Logic, FOProbLog is based on a notion of soft constraint, but formulas are labeled by probabilistic predicates instead of weights. Indeed, the higher the probability of a formula, the lower the probability will be of interpretations not satisfying it. The use of probabilities should make FOProbLog more intuitive from a knowledge representation point of view. Another difference is that the semantics of Markov Logic is defined through the sets of weighted ground instances of the formulas, while FOProbLog's semantics is defined in terms of groundings of probabilistic facts. The difference can be illustrated using the formula $\forall X.pf(X) \rightarrow \exists Y.likes(X,Y)$. In Markov Logic the probability would necessarily depend on the groundings of both X and Y, whereas in FOProbLog it only depends on those of X. Furthermore, the key inference mode in Markov Logic is typically based on the MPE principle approximating a most likely state given some evidence, while in FOProbLog an interval on the probability of queries is computed. In this regard, FOProbLog is also related to the work on Probabilistic Logic Programming (PLP) [51]. Both PLP and FOProbLog combine probabilities of basic random events to assign probabilities to Boolean valued interpretations, subject to constraints given by the program or theory. However, in PLP, basic random events as well as individual atoms in clauses can be annotated with (independent) closed probability intervals. Inference in PLP obtains maximally concise probability bounds for a given query by propagating those intervals using techniques from linear programming. Finally, FOProbLog is also closely related to systems such as PRISM [73] and ICL [64], which are both based on atomic and total choices, but only for definite clauses, as well as to Stochastic Logic Programs [48], an extension of probabilistic context-free grammars, where normalization is required to redistribute the probability mass of failing derivations over successful ones.

We have described how FOProbLog theories can be transformed into ProbLog programs and have identified the bottleneck to do inference in problems similar to the ones tackled by Markov Logic [67]. Much work remains to be done. Tabling makes search for all proofs feasible even for large problems. But calculating the ROBDD needed for normalization becomes too expensive. Hence approximation methods for tabled ProbLog are a first promising direction. Another one is the analysis of independencies in the theory, which may allow to restrict consistency computations to the parts of the database influencing the probability of the current query. Similarly, studying the influence of the depth bound on the probability values obtained may provide insights into the necessary depth of search. So far we only compute probabilities of ground atomic queries. The question arises whether inference for other queries is feasible. Our logic is as

expressive as Nilson's [52], indeed, for a formula F we can write $F : \alpha \vee \neg F : 1-\alpha$, so it is natural that some queries are hard to evaluate. This has to be analysed. On the representation side, the insights obtained from the case study can serve as basis for an automated translation of FOProbLog into ProbLog. Finally, learning parameters for FOProbLog based on corresponding techniques for ProbLog [26] is another interesting line of work.

7.2 Analysing a Publish/Subscribe System for Mobile Ad hoc Networks with ProbLog

This section presents joint work with Koosha Paridel, Gerda Janssens, Yves Vanrompay and Yolande Berbers published in [46]. The main contributions of the author is the implementation of the ProbLog application, and the evaluation of analysis of the Publish/Subscribe system. The paper [46] is directly presented here.

7.2.1 Introduction

We use ProbLog [35], a probabilistic extension of Prolog, to analyse Fadip [54], a Publish/Subscribe protocol for Mobile Ad hoc Networks (MANETs). Publish/Subscribe systems for MANETs are used commonly in disaster recovery, smart city and vehicular networks. We model a MANET as a probabilistic graph representing connections between nodes by ProbLog's probabilistic facts. The Fadip protocol can be seen as a special kind of path finding in such a probabilistic graph. As there can be multiple non-mutually exclusive paths between two nodes, ProbLog is an appropriate probabilistic system to model this application.

Our main contribution is to show how a simple ProbLog program can encode this Fadip application for the case of one publisher and one subscriber, and how simple ProbLog queries can then compute the probabilities of message delivery for different parameter settings. Before, simulations were needed to estimate delivery ratios, while now they are inferred analytically.

We analytically investigate the delivery probability among random node pairs and show the effects of the fading gossip technique. We also evaluate different parameter settings and conclude on what their impact is.

7.2.2 Problem Statement

Publish/Subscribe systems have been intensively studied for wired networks and infrastructured mobile networks [4]. When used in MANETs they suffer from scalability issues. Fadip [54] is a Publish/Subscribe system for MANETs and is designed to be lightweight in terms of the network topology (i.e. fixing routing information and maintaining logical structures of nodes) and the number of messages exchanged for communication. To achieve a reasonable delivery ratio, Fadip uses a hybrid model which propagates subscriptions and publications as bounded as possible and makes matching in intermediary nodes which act as undedicated rendezvous points. In Fadip, the routing is done probabilistically and neither the publishers nor the subscribers have any information about where their publication or subscription might be matched.

7.2.3 ProbLog

ProbLog [37] is a probabilistic framework that extends Prolog with probabilistic facts. A ProbLog program specifies a probability distribution over all possible non-probabilistic subprograms of the ProbLog program. The success probability of a query is defined as the probability that it succeeds in these subprograms. The framework includes different methods to compute probabilities: for Fadip we will use two alternatives, namely exact inference and program sampling as an approximation method.

ProbLog has been motivated by the real-life application of mining large biological networks where edges are labelled with probabilities. An edge represents a probabilistic link between the concepts represented by its nodes. The probabilistic links are mutually independent. ProbLog typically computes the probability of the existence of a path between two nodes [16]. The contribution of common parts in different paths between two nodes to the final probability is dealt according to the inclusion-exclusion principle from set theory.

7.2.4 Fadip Model in ProbLog

We model the MANET as a probabilistic graph. The graph nodes are the nodes of the mobile network. The graph edges model the connectivity between nodes. In a MANET this connectivity is not permanent. To model this, we attach to the edges probabilities which express the fraction of the time the connections are present. These probabilistic edges are represented by probabilistic facts.

We extend the path program to model the Publisher/Subscriber propagation of Fadip as a bounded bidirectional search of a path among two nodes. The parameters $MaxHop_p$ and $MaxHop_s$ are used as bounds when propagating the message of the Publisher and the subscription of the Subscriber, respectively. We used tabling as in [43] to avoid re-computations and to handle loops.

To integrate the fading gossip, we also need to express the fact that sending a message has a probability which decreases with the distance from its source [54]. We model this by flexible probabilistic facts, whose probability is determined at runtime. The delivery of a message between two nodes depends on the connection being present and the distance from the Publisher.

The model results in a relatively simple ProbLog program that allow us to query for the probability of a message delivery from a Publisher to a Subscriber[3].

7.2.5 Analysing the Model

We present how ProbLog can be used to analyse Fadip. The base operation is to calculate delivery probabilities. Fadip aims to reduce network traffic. For this it is beneficial to retain $MaxHop$ parameters as low as possible and to use fading gossip while retaining a good delivery probability. We used the OMNeT++ simulation log of a 150 node WiFi network, moving randomly at $1m/s$ in a playground of $1.5km \times 1km$. For the selection of Publisher and Subscriber, we considered two settings based on the distance between them. For the first, we randomly selected pairs of nodes which have paths with a minimum hop from 4 to 6 and for the second, pairs that have paths with a minimum hop from 8 to 10. We used ProbLog to query the delivery probability for these pairs with multiple values for $MaxHop_p$ and $MaxHop_s$ and activating or not fading gossip. The results of these queries are first used to evaluate the impact of the $MaxHop$ parameters. We observed that increasing $MaxHop_s$ has more effect on improving the delivery probability than increasing $MaxHop_p$. We also used the results to infer optimal values for $MaxHop_p$ and $MaxHop_s$. For example, in the first setting both are 3 and for the second setting are $4, 5$ respectively. Our analysis showed that fading gossip retains a high delivery probability for close distances up to $3 - 4$ hops and the delivery probability rapidly decreases for longer paths.

[3]The ProbLog program and the analysis results can be found at: `https://lirias.kuleuven.be/bitstream/123456789/284687/2/Appendix.pdf`

7.2.6 Conclusions

We presented an application of ProbLog that models and analyses the performance of Fadip. In [10] Bayesian Networks are used to analyse MANETs. By analysing the model one can infer delivery probabilities for different settings and use this information to chose optimal parameter settings and do evaluations. For our example network we concluded that $MaxHop_s$ is more important for obtaining a high delivery probability than $MaxHop_p$ and that the fading gossip technique has a good delivery probability for small hop counts while the message rapidly fades for larger hop counts. In [54] similar results are attained by simulating the behaviour of the network, in this work we attain them analytically. For future work we want to extend the ProbLog program to support multiple publishers and subscribers and study the impact of their interaction.

7.3 Appendix of: Analysing a Publish/Subscribe System for Mobile Ad hoc Networks with ProbLog

Fadip is a Publish/Subscribe system for Mobile Ad hoc Networks which uses probabilistic routing of messages to deal with the volatile nature of the network. It uses controlled propagation of publications and subscriptions, with the fading gossip technique to reduce the number of broadcasts. We present a probabilistic logic program in ProbLog that models Fadip. This allows us to calculate the probabilities that messages are successfully received by subscribers and to analyse the performance of the Fadip system.

7.3.1 Example Network

```
% used to model the nodes of the network
% in future might become probabilistic
node(n1).
node(n2).
node(n3).
node(n4).
% the probabilistic facts representing the connections
0.20::con_pf(n1,n2).
0.40::con_pf(n2,n3).
```

```
0.60::con_pf(n1,n3).
0.80::con_pf(n2,n4).
% to make connection double way
con(N1, N2):-
  node(N1),
  node(N2),
  N1 \== N2,
  are_con(N1, N2).
are_con(N1, N2):-
  con_pf(N1, N2).
are_con(N1, N2):-
  con_pf(N2, N1).
```

7.3.2 Fadip Model in Problog

```
% flexible probabilistic facts
% ID is used to to uniquely identify each msg
P::pf(_ID, P).

% use fading message if fadip, otherwise not
fading_msg(_, _):-
  normal.
fading_msg(_ID, 0):-
  fadip.
fading_msg(ID, HOPS):-
  fadip,
  HOPS > 0,
  P is 0.5 + 0.5 * e ^ (- 1.0 * HOPS),
  pf(ID, P).

% propagate_publication is tabled
:- dynamic propagate_publication/4.
  % propagate a publication from a Publisher to a Subscriber
  % 1st case: Publisher has a subscription & connection to Subscriber
  propagate_publication(Publisher, Subscriber, Message, Parameters):-
    check_subscription(Subscriber, Publisher, message(0), Parameters),
    check_hops(Message, Parameters, 1, 1),
    broadcast(Publisher, Subscriber, Message, 1).
  % 2nd case: Publisher has a connection to Subscriber
  propagate_publication(Publisher, Subscriber, Message, Parameters):-
    check_hops(Message, Parameters, 1, 0),
    broadcast(Publisher, Subscriber, Message, 0).
  % 3rd case: Publisher has a subscription, propagate to neighbours
  propagate_publication(Publisher, Subscriber, Message, Parameters):-
    check_subscription(Subscriber, Publisher, message(0), Parameters),
    check_hops(Message, Parameters, 1, 1),
    broadcast(Publisher, To, Message, 1),
    Subscriber \== To,
```

```
    increment_hops(Message, NewMessage),
    propagate_publication(To, Subscriber, NewMessage, Parameters).
  % 4th case: Publisher has not a subscription use fading message
  propagate_publication(Publisher, Subscriber, Message, Parameters):-
    check_hops(Message, Parameters, 1, 0),
    broadcast(Publisher, To, Message, 0),
    Subscriber \== To,
    increment_hops(Message, NewMessage),
    propagate_publication(To, Subscriber, NewMessage, Parameters).
:- problog_table propagate_publication/4.

:- dynamic check_subscription/4.
  % 1st case: Check if a subscription reached from a direct connection
  check_subscription(Publisher, Subscriber, Message, Parameters):-
    check_hops(Message, Parameters, 0, 1),
    broadcast(Publisher, Subscriber, Message, 0).
  % 2nd case: Check subscriptions through neighbours
  check_subscription(Publisher, Subscriber, Message, Parameters):-
    check_hops(Message, Parameters, 0, 1),
    broadcast(Publisher, To, Message, 0),
    Subscriber \== To,
    increment_hops(Message, NewMessage),
    check_subscription(To, Subscriber, NewMessage, Parameters).
:- problog_table check_subscription/4.

increment_hops(message(HOPS), message(NHOPS)):- NHOPS is HOPS + 1.
check_hops(message(HOPS), parameters(PUBMAXHOP, SUBMAXHOP), PUB, SUB):-
  HOPS < PUB * PUBMAXHOP + SUB * SUBMAXHOP.

broadcast(Sender, Receiver, message(_HOPS), 1):-
  con(Sender, Receiver).
broadcast(Sender, Receiver, message(HOPS), 0):-
  con(Sender, Receiver),
  fading_msg(id(Receiver, Sender), HOPS).

% To be able to swap between the two approachs
:-dynamic normal/0, fadip/0.
fadip.
method(Method):-
  Method, !.
method(normal):-
  retract(fadip),
  assert(normal).
method(fadip):-
  retract(normal),
  assert(fadip).
```

7.3.3 Options Used for Optimization

```
:- use_module(library(problog)).
```

```
:- use_module(library(lists)).
:- set_problog_flag(use_db_trie,true).
:- set_problog_flag(use_old_trie,false).
:- grow_atom_table(2000000), yap_flag(agc_margin, 200000000).
:- set_problog_flag(retain_tables, false).
:- set_problog_flag(refine_anclst, true).
:- set_problog_flag(anclst_represent, integer).
:- set_problog_flag(trie_preprocess, true).
```

7.3.4 Queries

One should query the probability of a message delivered. This is done with:

```
:- problog_exact(propagate_publication(Publisher, Subscriber,
                 message(0), parameters(MaxHopP, MaxHopS)), P,S).
```

Where Publisher, Subscriber can be any node number, the message contains the starting hops (0) and the parameters the Max Hop Publisher and Max Hop Subscriber. To activate/deactivate the fading gossip technique one needs to call method(fadip) or method(normal) respectively.

For example: `problog_exact(propagate_publication(n1,n4,message(0), parameters(2,1)),P,S)` returns: $P = 0.2650531$.

7.3.5 Analysis Results

The network and source code that used for analysis can be found at: `https://lirias.kuleuven.be/bitstream/123456789/284687/4/code.tar.gz`. We used the distributed ProbLog with the developer version of Yap 6[4] configured with `--prefix=$PWD --enable-tabling=yes --with-cudd="CUDDPATH"` together with CUDD-2.4.1[5].

The following tables present the data gathered through the analysis. The query column defines the Publisher to Subscriber nodes analysed for message delivery, the $MaxHop_{Pub}, MaxHop_{Sub}$ columns present the maximum hop parameters used and the normal, fadip columns the probability that the message reaches without fading gossip and with fading gossip technique respectively. The normal column in a real network it would flood the network significantly, thus one is

[4] `http://www.dcc.fc.up.pt/~vsc/Yap/`
[5] `http://vlsi.colorado.edu/~fabio/CUDD/`

Query	$\mathbf{MaxHop_{Pub}}$	$\mathbf{MaxHop_{Sub}}$	Normal	Fadip
50 to 36	0	5	0.93055	0.91667
50 to 36	0	6	0.93508	0.93507
50 to 36	3	3	0.91455	0.79199
50 to 36	3	4	0.92980	0.91448
50 to 36	3	5	0.93511	0.93511
50 to 36	3	6	0.93511	0.93511
50 to 36	4	3	0.91938	0.82786
50 to 36	4	4	0.92980	0.91760
50 to 36	4	5	0.93511	0.93511
50 to 36	4	6	0.93511	0.93511
50 to 36	5	0	0.70662	0.15027
50 to 36	5	3	0.91938	0.84773
50 to 36	5	4	0.92980	0.91760
50 to 36	5	5	0.93511	0.93511
50 to 36	5	6	0.93511	0.93511
50 to 36	6	0	0.81123	0.23443
50 to 36	6	3	0.91938	0.85641
50 to 36	6	4	0.92980	0.91798
50 to 36	6	5	0.93511	0.93511
50 to 36	6	6	0.93511	0.93511
42 to 28	0	5	0.99263	0.97636
42 to 28	0	6	0.99686	0.99680
42 to 28	3	3	0.95143	0.94046
42 to 28	3	4	0.99427	0.99426
42 to 28	3	5	0.99705	0.99705
42 to 28	3	6	0.99705	0.99705
42 to 28	4	3	0.95154	0.94428
42 to 28	4	4	0.99427	0.99427
42 to 28	4	5	0.99705	0.99705
42 to 28	4	6	0.99705	0.99705
42 to 28	5	0	0.84314	0.27184
42 to 28	5	3	0.95154	0.94544
42 to 28	5	4	0.99427	0.99427
42 to 28	5	5	0.99705	0.99705
42 to 28	5	6	0.99705	0.99705
42 to 28	6	0	0.84727	0.37159
42 to 28	6	3	0.95154	0.94618
42 to 28	6	4	0.99427	0.99427
42 to 28	6	5	0.99705	0.99705
42 to 28	6	6	0.99705	0.99705

Table 7.1: Results from nodes of 4 to 6 hop distance (1).

looking for the best trade in minimizing the MaxHop parameters and retaining high probability.

Query	**MaxHop$_{\mathbf{Pub}}$**	**MaxHop$_{\mathbf{Sub}}$**	**Normal**	**Fadip**
34 to 57	0	5	0.99989	0.99977
34 to 57	0	6	0.99989	0.99989
34 to 57	3	3	0.99826	0.99124
34 to 57	3	4	0.99989	0.99987
34 to 57	3	5	0.99989	0.99989
34 to 57	3	6	0.99989	0.99989
34 to 57	4	3	0.99826	0.99226
34 to 57	4	4	0.99989	0.99987
34 to 57	4	5	0.99989	0.99989
34 to 57	4	6	0.99989	0.99989
34 to 57	5	0	0.98985	0.66224
34 to 57	5	3	0.99826	0.99277
34 to 57	5	4	0.99989	0.99987
34 to 57	5	5	0.99989	0.99989
34 to 57	5	6	0.99989	0.99989
34 to 57	6	0	0.99013	0.79284
34 to 57	6	3	0.99826	0.99290
34 to 57	6	4	0.99989	0.99987
34 to 57	6	5	0.99989	0.99989
34 to 57	6	6	0.99989	0.99989
19 to 17	0	5	0.99951	0.99920
19 to 17	0	6	0.99990	0.99991
19 to 17	3	3	0.95969	0.90643
19 to 17	3	4	0.99880	0.99575
19 to 17	3	5	0.99992	0.99992
19 to 17	3	6	0.99992	0.99992
19 to 17	4	3	0.98306	0.93398
19 to 17	4	4	0.99888	0.99652
19 to 17	4	5	0.99992	0.99992
19 to 17	4	6	0.99992	0.99992
19 to 17	5	0	0.51387	0.13597
19 to 17	5	3	0.98356	0.94495
19 to 17	5	4	0.99888	0.99705
19 to 17	5	5	0.99992	0.99992
19 to 17	5	6	0.99992	0.99992
19 to 17	6	0	0.53996	0.18752
19 to 17	6	3	0.98356	0.95331
19 to 17	6	4	0.99888	0.99729
19 to 17	6	5	0.99992	0.99992
19 to 17	6	6	0.99992	0.99992

Table 7.2: Results from nodes of 4 to 6 hop distance (2).

Query	$\mathbf{MaxHop_{Pub}}$	$\mathbf{MaxHop_{Sub}}$	Normal	Fadip
96 to 91	4	5	0.31091	0.11317
96 to 91	4	6	0.75804	0.48629
96 to 91	5	4	0.29762	0.05732
96 to 91	5	5	0.62376	0.24765
91 to 0	4	5	0.09333	0.03549
91 to 0	4	6	0.03549	0.77500
91 to 0	4	5	0.77500	0.09333
135 to 91	4	5	0.32120	0.11716
135 to 91	4	6	0.77059	0.49556
135 to 91	5	4	0.30744	0.05933
135 to 91	5	5	0.63394	0.25327
102 to 85	4	5	0.73073	0.26262
102 to 85	4	6	0.93231	0.83127
102 to 85	5	4	0.68023	0.14909
87 to 86	4	5	0.61940	0.24165
87 to 86	4	6	0.84652	0.58963
87 to 86	5	4	0.60336	0.12513
87 to 86	5	5	0.70304	0.31845
87 to 86	5	6	0.84833	0.65106
87 to 86	6	4	0.64601	0.17245
87 to 86	6	5	0.70411	0.38441
87 to 86	6	6	0.84833	0.68492
63 to 51	4	5	0.97482	0.65070
63 to 51	4	6	0.20526	0.93928
76 to 2	4	5	0.61283	0.52868
76 to 2	4	6	0.05476	0.89417
76 to 2	5	4	0.20830	0.25000
76 to 2	5	5	0.69991	0.52875
76 to 2	5	6	0.05931	0.89417
76 to 2	6	4	0.26636	0.25006
76 to 2	6	5	0.70025	0.52875
76 to 2	6	6	0.37030	0.89417
86 to 68	4	5	0.90069	0.78221
86 to 68	4	6	0.27176	0.96060
86 to 68	5	4	0.53345	0.77712
86 to 68	5	5	0.93215	0.80056
86 to 68	5	6	0.41493	0.96066
86 to 68	6	4	0.63340	0.79534
86 to 68	6	5	0.94239	0.80061
86 to 68	6	6	0.73740	0.96066
117 to 85	4	5	0.93597	0.93050
117 to 85	4	6	0.34620	0.94227
117 to 85	5	4	0.00000	0.82474

Table 7.3: Results from nodes of 8 to 10 hop distance.

Chapter 8

Conclusion

We have presented ProbLog a probabilistic extension of Prolog. This new programming language is both a simple and very expressive tool useful for modelling data mining and machine learning applications. Together with its expressibility power several performance issues appear.

In this thesis we have made a clear distinction among the several steps that ProbLog takes to perform inference: **SLD resolution**, **Boolean formulae preprocessing** and **ROBDD compilation**. We have presented different ways to optimise the performance of each step separately in order to achieve more tractable inference for different ProbLog programs.

We presented how ProbLog and other similar probabilistic logic programming systems can be tabled. The benefits from tabling ProbLog programs are both in performance and expressive power. Furthermore, tabled programs are more declarative than their non-tabled equivalents. We have shown how tabled ProbLog programs achieve significant performance improvements over the **SLD resolution** step. We also have presented how tabling for a class of problems affected negatively the next steps. To achieve tabling in ProbLog we have introduced a new data structure, namely nested tries. This newly introduced data structure has similarities with the support graph of PRISM.

Furthermore, we introduced three different approaches to perform Boolean formulae preprocessing for DNF Boolean formulae, namely naive, decomposition and recursive node merging. We compared them experimentally and explained them. Unfortunately, we shown that there is "no one best for all" programs method. Besides the three different preprocessing methods, we also presented a new data structure, namely, depth breadth trie which can be used to further

optimise preprocessing. We also presented three additive optimisations that use this data structure. Finally, we presented a generalised algorithm that performs preprocessing over nested tries by using any of the preprocessing methods available. As nested tries are produced from tabling, it is very important that the preprocessing methods perform well together with nested tries. To improve their performance we introduced several different optimisations that target at remedying the negative performance effects of tabling.

In order to improve the performance over the last step of inference, ROBDD compilation, we presented an approach that relies on finding patterns of Boolean variables in DNF Boolean formulae, namely AND/OR-clusters. We used these patterns to reduce the number of Boolean variables and simplify the DNF that is needed to be processed. In this thesis we explained how AND/OR-clusters can be taken in advantage for probabilistic inference and we also mentioned that these patterns could have similar benefits in advantage in other fields that use DNF formulae. We also presented two polynomial algorithms that detect this patterns in DNF formulae. In addition we showed how the two patterns can be used interchangeable to reduce the DNF Boolean formulae. We finally, experimentally verified the benefits of AND-clusters.

Besides improving the performance of inference methods, our work also included the extension of ProbLog's functionality. We presented how a ProbLog negation that abides to the well found semantics can be realised. Furthermore, we also extended ProbLog with meta-calls and higher order functionality. In order to achieve efficiently these tasks we introduced the abstract ProbLog engine which we used to implement several different inference methods. Through nesting ProbLog engines we achieved to implement meta-calls and higher order functions such as ProbLog answers and ProbLog findall predicates.

Finally, we presented two ProbLog applications that use several of the mentioned features in order to achieve their tasks. We see that tabling is an irreplaceable feature of ProbLog both for expressibility and performance benefits.

The ProbLog2010 and ProbLog2011 systems equipped with the presented improvements and optimisations have shown a significant improvement compared with the initial implementation[1] of ProbLog. From our experience for SLD resolution, tabling is indispensable and heavily used. For the preprocessing step the most used method is recursive node merging in combination with depth breadth trie. While decomposition is often advantageous, the lack of an implementation that combines decomposition with nested tries implies its use is restricted to non tabled programs. Finally the AND/OR-clusters are a very promising approach, unfortunately the implementation currently only computes

[1] With initial implementation we refer to ProbLog before any of our contributions.

AND-clusters and only on DNF Boolean formulae without negated literals, so it is only useful for specific programs.

We propose the following settings for ProbLog 2010 implementation:

- for non cyclic applications that benefit from tabling, such as (Hidden) Markov Models, Bayesian Networks, we suggest the use of: **Tabling**, preprocessing by **Recursive Node Merging** together with the use of **Depth Breadth Trie** with optimisation level three;
- for cyclic applications that benefit from tabling, such as reachability in probabilistic graphs, we suggest the use of: **Tabling**, preprocessing by **Recursive Node Merging** together with the use of **Depth Breadth Trie** and with the optimisations: **ancestor check**, **ancestor list refine** or **pre-process** active;
- for applications where tabling does not grand a significant benefit, such as bounded reachability in probabilistic graphs, we suggest the use of: **Decomposition** together with **AND-clusters compression**.

8.1 Future Work

The work presented in this thesis, leaves several interesting points for further research. On the part of performance the most challenging open matter is the improvement of the preprocessing method for nested tries. Furthermore, while the presented preprocessing approaches are theoretically compatible among them due to technical issues we have not investigated their behaviour when combined. An outline to that direction would be the investigation of using the decomposition method in conjunction with the depth breadth trie data structure.

Further research is been made related to the replacement of ROBDDs by arithmetic circuits [12, 85] or other knowledge compilation data structure. Related to this matter, future work also should focus on avoiding compiling a Boolean formula in a data structure as much as possible. Easily, several different patterns could be detected in DNF or CNF Boolean formulae. These patterns could then be used to compute the probability for parts of the problem reducing the size of the Boolean formula to compile. In some cases these patterns could signify mutual exclusiveness of separate parts of the Boolean formula, one could then divide the Boolean formula in several smaller ones and after compiling each smaller re-combine the result. In general, approaches such as that could ensure polynomial inference for several classes of programs.

Future work, also could be focused on several aspects not related with performance. Our initial approach for achieving meta-calls introduced several new functionalities for ProbLog, research to extend the functionality through meta-calls could be interesting. A second form of probabilistic findall could be developed that would return possible worlds instead of answers.

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[2] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Faster SAT and smaller BDDs via common function structure. In *Proceedings of International Conference on Computer Aided Design*, pages 443–448, 2001.

[3] H. R. Andersen. An introduction to binary decision diagrams. In *Lecture Notes,* `http://www.cs.auc.dk/~kgl/VERIFICATION99/mm4.html`, 1997.

[4] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. *DIS,Universita di Roma"La Sapienza", Tech.Rep*, 2005.

[5] R. S. Bird. Tabulation techniques for recursive programs. *ACM Comput. Surv.*, 12(4):403–417, December 1980.

[6] H. Blockeel. *Top-Down Induction of First Order Logical Decision Trees*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998. `http://www.cs.kuleuven.ac.be/~ml/PS/blockeel98:phd.ps.gz`.

[7] H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artif. Intell.*, 101(1-2):285–297, 1998.

[8] M. Bruynooghe, T. Mantadelis, A. Kimmig, B. Gutmann, J. Vennekens, G. Janssens, and L. De Raedt. ProbLog technology for inference in a probabilistic first order logic. In H. Coelho, R. Studer, and M. Wooldridge, editors, *Proceedings of European Conference on Artificial Intelligence*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 719–724. IOS Press, 2010.

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

[10] S. Buchegger and J.-Y. L. Boudec. The effect of rumor spreading in reputation systems for mobile Ad-Hoc networks. In *Proceedings of WiOpt*, 2003.

[11] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[12] A. Darwiche and P. Marquis. A knowledge compilation map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264, 2002.

[13] L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors. *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*. Springer, 2008.

[14] L. De Raedt and K. Kersting. Probabilistic logic learning. *SIGKDD Explorations*, 5(1):31–48, 2003.

[15] L. De Raedt and K. Kersting. Probabilistic inductive logic programming. In S. Ben-David, J. Case, and A. Maruoka, editors, *Proceedings of Algorithmic Learning Theory*, volume 3244 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2004.

[16] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In M. M. Veloso, editor, *Proceedings of International Joint Conferences on Artificial Intelligence*, pages 2462–2467, 2007.

[17] J. Eisner and N. W. Filardo. Dyna: Extending datalog for modern AI. In O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, editors, *Datalog*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer, 2010.

[18] D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, and L. De Raedt. Inference in probabilistic logic programs using weighted CNF's. In F. G. Cozman and A. Pfeffer, editors, *Proceeding of Uncertainty in Artificial Intelligence*, pages 211–220. AUAI Press, 2011.

[19] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, Sept. 1960.

[20] A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Communications of the ACM*, 38(3):620–650, 1991.

[21] L. Getoor, N. Friedman, D. Koller, A. Pfeffer, and B. Taskar. Probabilistic relational models. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

[22] L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[23] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and D. Tarlow. Church: a language for generative models. *Computing Research Repository*, abs/1206.3255, 2012.

[24] H.-F. Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *Proceedings of the 17th International Conference on Logic Programming*, pages 150–165, London, UK, 2001. Springer-Verlag.

[25] B. Gutmann, M. Jaeger, and L. De Raedt. Extending ProbLog with continuous distributions. In P. Frasconi and F. A. Lisi, editors, *Proceedings of Inductive Logic Programming*, volume 6489 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2010.

[26] B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. Parameter learning in probabilistic databases: A least squares approach. In W. Daelemans, B. Goethals, and K. Morik, editors, *ECML/PKDD (1)*, volume 5211 of *Lecture Notes in Computer Science*, pages 473–488. Springer, 2008.

[27] B. Gutmann, I. Thon, and L. De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, editors, *ECML/PKDD (1)*, volume 6911 of *Lecture Notes in Computer Science*, pages 581–596. Springer, 2011.

[28] P. Hintsanen. The most reliable subgraph problem. In *Proceedings of Principles and Practice of Knowledge Discovery in Databases*, pages 471–478, 2007.

[29] Intelligent Systems Laboratory. Quintus Prolog user's manual, 2003. `http://www.sics.se/quintus/`.

[30] T. Janhunen. Representing normal programs with clauses. In R. L. de Mántaras and L. Saitta, editors, *ECAI*, pages 358–362. IOS Press, 2004.

[31] G. Janssens, B. Demoen, R. Tronçon, and H. Vandecasteele. hipP User's Manual, 2006. `/cw/prolog/hipp/hipp-yes.linux/share/doc/hipp/manual.pdf`.

[32] R. M. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *SFCS '83: Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 56–64, Washington, DC, USA, 1983. IEEE Computer Society.

[33] K. Kersting and L. De Raedt. Basic principles of learning Bayesian logic programs. In De Raedt et al. [13], pages 189–221.

[34] A. Kimmig. *A Probabilistic Prolog and its Applications*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, Nov. 2010. Luc De Raedt (supervisor).

[35] 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*, 11(2-3):235–262, 2011.

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

[37] A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt. On the efficient execution of ProbLog programs. In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of International Conference on Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2008.

[38] A. Kimmig, G. Van den Broeck, and L. De Raedt. An algebraic Prolog for reasoning about possible worlds. In W. Burgard and D. Roth, editors, *Proceedings of AAAI Conference on Artificial Intelligence*. AAAI Press, 2011.

[39] D. Maier. The complexity of some problems on subsequences and supersequences. *Communications of the ACM*, 25(2):322–336, 1978.

[40] T. Mantadelis. SimpleCUDD package, technical report and manual, 2008. `https://lirias.kuleuven.be/handle/123456789/253405`.

[41] T. Mantadelis, B. Demoen, and G. Janssens. A simplified fast interface for the use of CUDD for Binary Decision Diagrams, September 2008. Computer Intelligence and Learning PhD student day (collocated with ECML PKDD 2008), Antwerp, 15-19 September 2008.

[42] T. Mantadelis and G. Janssens. Tabling relevant parts of SLD proofs for ground goals in a probabilistic setting. In P. Tarau, P. Moura, and N. Zhou, editors, *International Colloquium on Implementation of Constraint and*

LOgic Programming Systems (CICLOPS), Pasadena, California, USA, 14-17 July 2009, July 2009.

[43] T. Mantadelis and G. Janssens. Dedicated tabling for a probabilistic setting. In M. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 124–133, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[44] T. Mantadelis and G. Janssens. Variable compression in ProbLog. In C. G. Fermüller and A. Voronkov, editors, *Proceedings of International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2010.

[45] T. Mantadelis and G. Janssens. Nesting probabilistic inference. *Computing Research Repository*, abs/1112.3785, 2011.

[46] T. Mantadelis, K. Paridel, G. Janssens, Y. Vanrompay, and Y. Berbers. Analysing a Publish/Subscribe System for Mobile Ad Hoc Networks with ProbLog. In R. Rocha and J. Launchbury, editors, *Proceedings of Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*, pages 34–37. Springer, 2011.

[47] T. Mantadelis, R. Rocha, A. Kimmig, and G. Janssens. Preprocessing Boolean Formulae for BDDs in a Probabilistic Context. In T. Janhunen and I. Niemelä, editors, *Proceedings of European Conference on Logics in Artificial Intelligence*, volume 6341 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2010.

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

[49] C. Muise, S. A. Mcilraith, J. C. Beck, and E. Hsu. DSHARP: Fast d-DNNF compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*, Canadian Conference on Artificial Intelligence, 2012.

[50] N. Narodytska and T. Walsh. Constraint and variable ordering heuristics for compiling configuration problems. In *Proceedings of International Joint Conferences on Artificial Intelligence*, pages 149–154, 2007.

[51] R. T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information Computing*, 101(2):150–201, 1992.

[52] N. J. Nilson. Probabilistic logic. *AI*, 28:71–87, 1986.

[53] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of International Conference on Computer Aided Design*, pages 74–77, 1995.

[54] K. Paridel, Y. Vanrompay, and Y. Berbers. Fadip: Lightweight Publish/Subscribe for Mobile Ad Hoc Networks. In R. Meersman, T. S. Dillon, and P. Herrero, editors, *OTM Conferences (2)*, volume 6427 of *Lecture Notes in Computer Science*, pages 798–810. Springer, 2010.

[55] G. Pemmasani, H.-F. Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online justification for tabled logic programs. In *Lecture Notes in Computer Science: Logic Programming*, pages 500–501, London, UK, 2003. Springer-Berlin.

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

[57] A. Pfeffer. Ibal: A probabilistic rational programming language. In B. Nebel, editor, *Proceedings of International Joint Conferences on Artificial Intelligence*, pages 733–740. Morgan Kaufmann, 2001.

[58] A. Pfeffer. CTPPL: A continuous time probabilistic programming language. In C. Boutilier, editor, *Proceedings of International Joint Conferences on Artificial Intelligence*, pages 1943–1950, 2009.

[59] A. Pfeffer. Figaro: An object-oriented probabilistic programming language, 2009.

[60] D. Poole. Logic programming, abduction and probability. In *Proceedings of Future Generation Computer Systems*, pages 530–538, 1992.

[61] D. Poole. Probabilistic Horn abduction and bayesian networks. *Artif. Intell.*, 64(1):81–129, 1993.

[62] D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

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

[64] D. Poole. The independent choice logic and beyond. In De Raedt et al. [13], pages 222–243.

[65] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for effciently computing WFS. In *Proceedings of LPNMR*, pages 431–441, 1997.

[66] A. Rauzy, E. Châtelet, Y. Dutuit, and C. Bérenguer. A practical comparison of methods to assess sum-of-products. *Reliability Engineering & System Safety*, 79(1):33–42, 2003.

[67] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[68] F. Riguzzi and T. Swift. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In M. V. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 162–171. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[69] F. Riguzzi and T. Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Computing Research Repository*, abs/1107.4747, 2011.

[70] R. Rocha, F. Silva, R. R. Fern, and V. Santos Costa. A tabling engine for the YAP Prolog system. In *Proceedings of the APPIA-GULP-PRODE Joint Conference on Declarative Programming, AGP'2000*, La Habana, Cuba, December 2000.

[71] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. YAP user's manual, 2002. http://www.ncc.up.pt/˜vsc/Yap.

[72] T. Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of International Conference on Logic Programming*, pages 715–729, 1995.

[73] T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of International Joint Conferences on Artificial Intelligence*, pages 1330–1339. Morgan Kaufmann, 1997.

[74] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. Artif. Intell. Res. (JAIR)*, 15:391–454, 2001.

[75] T. Sato and Y. Kameya. Statistical abduction with tabulation. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 567–587. Springer, 2002.

[76] P. Schachte. Global variables in logic programming. In *Proceedings of International Conference on Logic Programming*, pages 3–17, 1997.

[77] P. Sevon, L. Eronen, P. Hintsanen, K. Kulovesi, and H. Toivonen. Link discovery in graphs derived from biological databases. In U. Leser, F. Naumann, and B. A. Eckman, editors, *DILS*, volume 4075 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2006.

[78] D. S. Shterionov, A. Kimmig, T. Mantadelis, and G. Janssens. DNF Sampling for ProbLog Inference. *Computing Research Repository*, abs/1009.3798, 2010.

[79] P. Singla and P. Domingos. Entity resolution with Markov logic. In *Proceedings of International Conference on Data Mining*, pages 572–582. IEEE Computer Society, 2006.

[80] F. Somenzi. CUDD: Colorado University Decision Diagram package, programmer's manual, 2005. `http://vlsi.colorado.edu~fabio/CUDD/`.

[81] M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Automated Reasoning*, 4:353–380, 1988.

[82] T. Swift and D. S. Warren. XSB: Extending Prolog with tabled logic programming. *CoRR*, abs/1012.5123, 2010.

[83] T. Swift, D. S. Warren, and et al. The XSB system, programmer's manual, 2009.

[84] H. Tamaki and T. Sato. OLD resolution with tabulation. In E. Shapiro, editor, *Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin / Heidelberg, 1986.

[85] L. G. Valiant. Why is Boolean complexity theory difficult? In *Poceedings of the London Mathematical Society symposium on Boolean function complexity*, pages 84–94, New York, NY, USA, 1992. Cambridge University Press.

[86] G. Van den Broeck, I. Thon, M. van Otterlo, and L. De Raedt. DTProbLog: A decision-theoretic probabilistic Prolog. In M. Fox and D. Poole, editors, *Proceedings of AAAI Conference on Artificial Intelligence*. AAAI Press, 2010.

[87] J. Vennekens, M. Denecker, and M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3):245–308, 2009.

[88] J. Vennekens, A. Kimmig, T. Mantadelis, B. Gutmann, M. Bruynooghe, and L. De Raedt. From ProbLog to first order logic: A first exploration. In

P. Domingos and K. Kersting, editors, *International Workshop on Statistical Relational Learning, Leuven, Belgium, 2-4 July 2009*, July 2009.

List of Publications

Journal Articles

- Koosha Paridel, Theofrastos Mantadelis, Ansar-Ul-Haque Yasar, Davy Preuveneers, Gerda Janssens, Yves Vanrompay, Yolande Berbers. *Analyzing the efficiency of context-based grouping on collaboration in VANETs with large-scale simulation.* Journal of Ambient Intelligence and Humanized Computing, volume 3, pages 1-16, 2012.

Conference Papers

- Theofrastos Mantadelis, Koosha Paridel, Gerda Janssens, Yves Vanrompay, Yolande Berbers. *Analysing a publish/subscribe system for mobile ad hoc networks with ProbLog.* Ricardo Rocha, John Launchbury (eds.), PADL, Austin, 24-25 January 2011, LNCS Series, volume 6539, pages 34-37, Springer 2011.

- Theofrastos Mantadelis, Gerda Janssens. *Variable compression in ProbLog.* Christian Fermüller, Andrei Voronkov (eds.), Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Yogyakarta, Indonesia, 10-15 October 2010, Lecture Notes in Computer Science, volume 6397, pages 504-518, Springer-Verlag's 2010.

- Theofrastos Mantadelis, Ricardo Rocha, Angelika Kimmig, Gerda Janssens. *Preprocessing Boolean formulae for BDDs in a probabilistic context.* Tomi Janhunen, Ilkka Niemelä (eds.), The European Conference on Logics in Artificial Intelligence, Helsinki, 13-15 September 2010, Logics in Artificial Intelligence, 12th European Conference, JELIA 2010, Proceedings, volume 6341, issue 12, pages 260-272, Springer 2010.

- Maurice Bruynooghe, Theofrastos Mantadelis, Angelika Kimmig, Bernd Gutmann, Joost Vennekens, Gerda Janssens, Luc De Raedt. *ProbLog technology for inference in a probabilistic first order logic.* Helder Coelho, Rudi Studer, Michael Woolridge (eds.), European Conference on Artificial Intelligence, Lisbon, Portugal, 16-20 August 2010, ECAI 2010 - 19th European Conference on Artificial Intelligence, pages 719-724, IOS Press 2010.

- Theofrastos Mantadelis, Gerda Janssens. *Dedicated tabling for a probabilistic setting.* Manuel Hermenegildo, Torsten Schaub (eds.), International Conference on Logic Programming, Edinburgh, Scotland, July 16-19, Technical Communications of the 26th International Conference on Logic Programming, volume 7, pages 124-133, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

Workshop Papers

- Theofrastos Mantadelis, Gerda Janssens. *Nesting probabilistic inference.* Salvador Abreu, Vitor Santos Costa (eds.), International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), Lexington, Kentucky, USA, 10 July 2011, CICLOPS, pages 1-16, 2011.

- Dimitar Shterionov, Angelika Kimmig, Theofrastos Mantadelis, Gerda Janssens. *DNF sampling for ProbLog inference.*, International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), Edinburgh, Scotland, 15 July 2010, Proceedings International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), 15 pages, 2010.

- Mantadelis, Theofrastos; Janssens, Gerda. Tabling relevant parts of SLD proofs for ground goals in a probabilistic setting, Tarau, Paul; Moura, Paulo; Zhou, Neng-Fa (eds.), International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), Pasadena, California, USA, 14-17 July 2009.

- Joost Vennekens, Angelika Kimmig, Theofrastos Mantadelis, Bernd Gutmann, Maurice Bruynooghe, Luc De Raedt. *From ProbLog to first order logic: A first exploration.* Pedro Domingos, Kristian Kersting (eds.), International Workshop on Statistical Relational Learning, Leuven, Belgium, 2-4 July 2009.

Technical Reports & Manuals

- Theofrastos Mantadelis, Gerda Janssens. *Variable compression in ProbLog (technical report).* CW Reports, volume CW586, 11 pages, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2010.
- Theofrastos Mantadelis, Bart Demoen, Gerda Janssens. *A simplified fast interface for the use of CUDD for Binary Decision Diagrams. A BDD Tool for Statistical Relational Learning: SimpleCUDD (poster & manual).* Computer Intelligence and Learning PhD student day (collocated with ECML PKDD 2008), Antwerp, 15-19 September 2008.

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
Scientific Computing Group
Celestijnenlaan 200A box 2402
B-3001 Heverlee